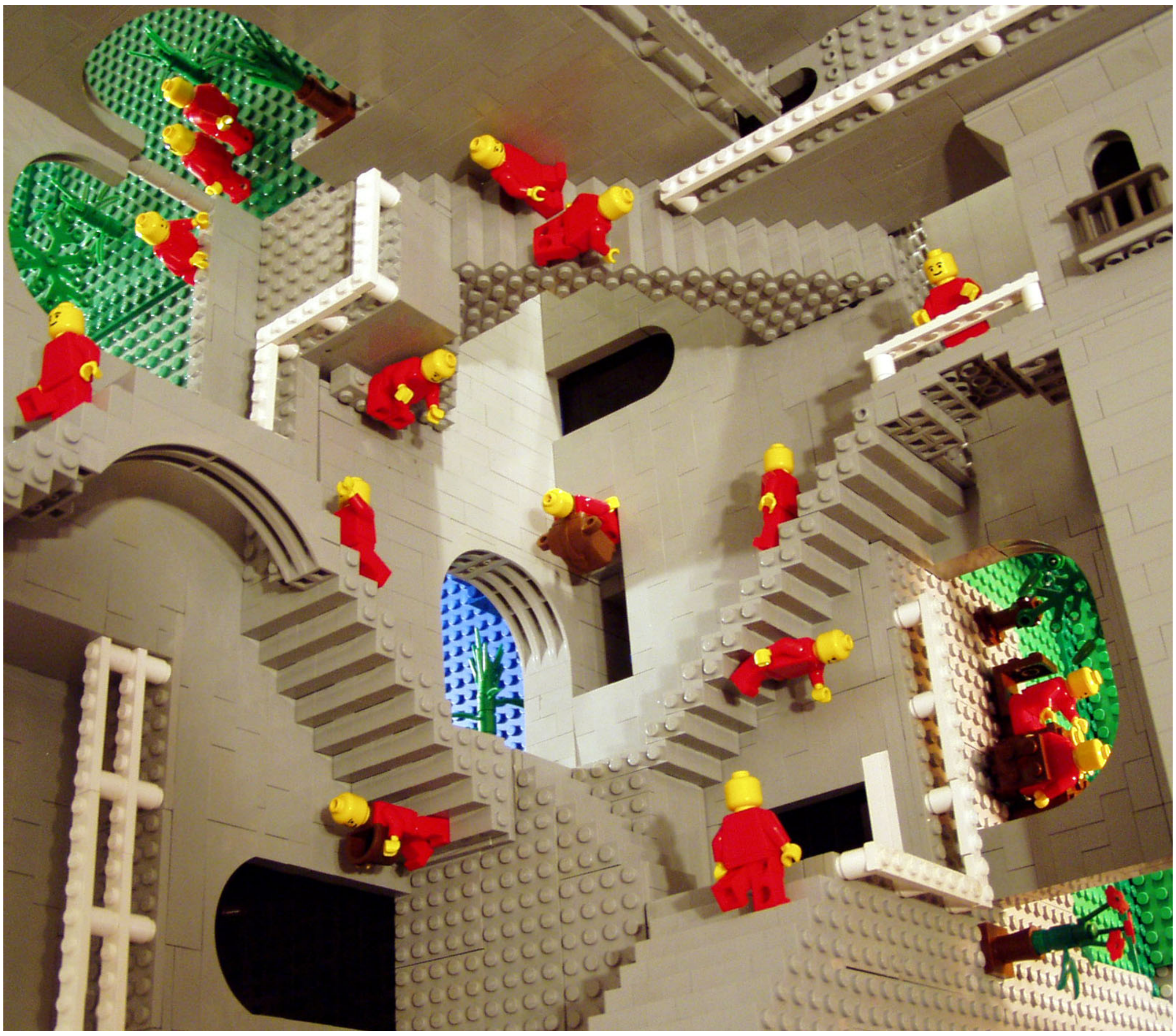
IMPLICIT COMPLEXITY IN THEORY AND PRACTICE

Disputats for the degree of *Doctor Scientiarum* at the University of Copenhagen

JEAN-YVES MOYEN



København, 2017

Contents

Introduction

Abstract:

Implicit Computational Complexity aims at finding syntactical criteria on programs that guarantee a runtime behaviour (usually, complexity). Ideally, ICC should analyse any program to determine its complexity, possibly even suggesting optimisations. While it is impossible to precisely determine the complexity of all programs, ICC still manages to prove that a wide range of programs run in polynomial time.
This dissertation presents my past and present works in the field of ICC.
The first part groups several journal articles showing incremental advances in ICC: the notion of *Quasi-Interpretation* and how it can be used for proving various complexity bounds; the notion of *Resource Control Graphs* and how they can be used for expressing several ICC results in a common language; and the notion of *transverse criteria* and how they can be used for comparing ICC results.
The second part groups current works studying the set of equivalences between programs, a new direction in ICC that builds on twenty years of research in the field. The *extensional equivalence* (two programs are equivalent if they compute the same function) plays a central role in Computability. Are there any other equivalences which could play a similarly important role?
The third part shows a possible application of ICC "in real life". After twenty years and many results, it seems that ideas from ICC are ready to be used in actual compilers. Indeed, it is possible to add some new analysis or optimisations as compiler passes. This opens ways for a fruitful collaboration between two separate research communities.

Dansk Resume:

I feltet *implicit beregningskompleksitet* (ICC) søger man at finde syntaktiske kriterier, som kan garantere, at programmer udviser bestemte egenskaber (typisk kompleksitet) på kørselstidpunktet. Ideelt burde ICC kunne anvendes til at analysere ethvert program med det formål at bestemme dets kompleksitet, og måske endda foreslå programoptimeringer. Medens det er umuligt præcist at bestemme alle programmers kompleksitet, er det stadigt muligt via metoder fra ICC at bevise, at en bred vifte af programmer kører i polynomiel tid.
I denne disputats præsenteres mine hidtidige resultater inden for ICC.
Disputatsens første del indeholder en række tidsskriftartikler, som præsenterer inkrementelle fremskridt inden for ICC: *Kvasi-fortolkninger* (*Quasi-interpretations*) og hvordan de kan bruges til at bevise forskellige kompleksitetsgrænser; *Ressourcekontrolgrafer* (*Resource Control Graphs*) og hvordan de kan anvendes til at udtrykke adskillige resultater i ICC under anvendelse af en fælles formalisme; samt *transverse kriterier* (*transverse criteria*) og hvordan de kan anvendes til at sammenligne resultater i ICC.
Disputatsens anden del indeholder nylig forskning, som undersøger mængden af ækvivalensrelationer mellem programmer, en ny retning inden for ICC som bygger på tyve års studier i feltet. Den såkaldte *ekstensionelle ækvivalensrelation* (to programmer er ækvivalente hvis de beregner den samme afbildning) spiller en central rolle i beregnelighedsteori. Findes der andre ækvivalensrelationer, som kunne spille en lignende, vigtig rolle?
Disputatsens tredje del omhandler en mulig anvendelse af ICC "i den virkelige verden". Efter tyve års arbejde og mange resultater ser det ud som om, at ideer fra ICC er klar til at kunne bruges i faktiske oversættere: det er muligt at tilføje passende nye analyser og optimeringer som oversætter-gennemløb. Dette åbner mulighed for et frugtbart samarbejde mellem to hidtil adskilte forskningsmiljøer.

Acknowledgements:

First and foremost, I wish to thank my co-authors, who had a direct contribution to this dissertation: James E. Avery, Patrick Baillot, Guillaume Bonfante, Ugo dal Lago, Jean-Yves Marion, Thomas Rubiano, Pavel Ruzicka, Thomas Seiller, and Jakob G. Simonsen. Many colleagues in Paris and elsewhere also had a strong influence on my work through many discussions on these subjects: Pierre Boudes, Christophe Fouqueré, Stefano Guerrini, Paulin Jacobé de Naurois, Lars Kristiansen, Jean-Vincent Loddo, Guilio Manzonetto, Micaela Mayero, Damiano Mazza, Virgil Mogbil, and Michele Pagani.

As often, thanking everybody personally is not possible, so my thoughts go:

- among family, to my father, for the intellectual curiosity he gave us, and for everything else;
- among colleagues, to Neil D. Jones, for first welcoming me in Denmark, and for the constant inspiration;
- among friends, to everybody at Bastard Café, for being a family away from family.

If your name is not here and you believe it should, you can try and convince me to buy you a beer.

Contributions of the dissertation

About this document

This document is a *doktordisputats* – a dissertation within the Danish academic system required to obtain the degree of *Doctor Scientiarum*, equivalent to the French *Habilitation à Diriger des Recherches.* It contains my work in the field of *Implicit Computational Complexity* from 2003 to 2016 and consists in a collection of published papers as well as some yet unpublished work.

The articles in this dissertation are grouped in three parts. In addition to the thematic grouping, this can also be viewed as a temporal grouping: my past in ICC; my present work and ideas; and a possible future for ICC. This is also reflected in the choice of material presented in each part: my past research is represented by journal articles; my present one by articles under revision, or unpublished long versions of workshop papers; and the possible future by short articles only.

1 Background

1.1 Complexity

The classical study of Complexity concerns the amount of resources (usually, time or space) that a program needs in order to run. It extends to the study of problems (or functions) by studying the *minimal* amount of resources needed to solve a given problem.

The core dichotomy of Computability, between programs and functions, is all the more present in Complexity. While it is (relatively) easy to define the complexity of a single program (in short, its running time), there are many different ways to solve a problem, thus many different programs that compute the same function. And each of these programs has a different complexity: anyone who studied a bit of Algorithmics knows that sorting can be done in time $\mathcal{O}(n^2)$ with insertion sort or $\mathcal{O}(n\log(n))$ with merge sort (among other possibilities).

Thus, the complexity of a function is defined as being the minimal complexity of any program computing it. Since there are infinitely many programs computing any given function, it is usually difficult to find out that complexity. The definition, however, provides a very easy way to show *upper bounds* on the complexity of functions: the existence of the insertion sort algorithm shows that the complexity of sorting is *at most* $\mathcal{O}(n^2)$. On the other hand, *lower bounds* on the complexity of functions are usually extremely hard to prove (sorting being an exception).

Knowing the complexity of a program is obviously interesting: it makes it possible to determine which parts of the program are efficient and which are not, and to predict the amount of resources that the program will need to be executed, thus making sure that it is actually possible to run it on a given computer. Knowing the complexity of a function is even better: it makes it possible to determine which parts of the program have too high a complexity and should be optimised or rewritten.

Finding the complexity of any program is in general uncomputable. However, in many specific cases, it is possible to find the complexity of individual programs. Unfortunately, computing the complexity of a given program is not an easy task.

Firstly, it requires some non-trivial computations, even more so when the program is complicated, or uses some advanced features of programming languages. Most bachelors in Computer Science had lessons in Complexity and it is usually not an easy subject to teach and to learn. Even without digging into advanced complexity analysis, solving recursive equations can be difficult.

Secondly, complexity is formally defined in terms of a computational model (usually Turing Machines, sometimes λ-calculus). Because of their lack of expressivity, computational models are not used to actually program anything. This makes the analysis of actual programs even harder.

So, ideally, we would like to have a tool which is able to perform a clever analysis of actual programs and determine their complexity, and maybe even analyse the *function* computed by the program and suggest some optimisations or drastic rewrites of the program to improve its complexity. Unfortunately, all of these dreams are impossible (that is, not computable).

1.2 Implicit Computational Complexity

The field of *Implicit Computational Complexity* (ICC) was launched by the early works of Cobham on *Bounded Recursion*, and the breakthrough of Bellantoni and Cook on *Safe Recursion.* They designed restricted programming languages in which it is only possible to write programs that run in polynomial time.

Their works 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 charge of devising a proof has been pushed upstream from the programmer. As soon as a program is written in one of these languages, it is guaranteed to run in polynomial time. The programmer does not need anymore to know about complexity, 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.

2 Advances in Implicit Complexity

The first part of this dissertation contains various works I have done in the field of Implicit Computational Complexity. It is composed of three journal articles that are long versions of works presented in conferences and workshops at the time.

Quasi-interpretations: a way to control resources is a gathering of many works around the notion of *Quasi-Interpretations* (QI) that I introduced during my PhD. While Safe Recursion was a somewhat monolithic analysis, we have split it in two parts by separating the *termination* from the *bound* itself. Termination of programs is ensured by usual termination tools, in this case termination orderings, while the polynomial bound is enforced by the QI. We show how the choice of termination ordering as well as the kind of QI used ensures various complexity bounds on the programs.

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 explicit complexity (the complexity of the program itself), it tries to analyse the implicit complexity (the complexity of the function). 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 polytime 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.

Resource Control Graphs (RCG) is a somewhat generic framework in which to express several different ICC analysis. 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.

I show this way that, notably, both *Non Size Increasing* programs and the *Size Change Principle* can be expressed as RCGs despite being apparently extremely different.

This work shows my early efforts towards unifying ICC systems. 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.

For a long time, I have been convinced that ICC systems are like the proverbial blind men approaching an elephant: each touching a different part and describing it in terms that let the others think this is a different beast (the legs are like pillars, the trunk like a branch, the belly like a wall, the tail like a rope, the tusks like pipes, and the ears like paper). RCG is an attempt to take a step back and use a common language to describe the ICC elephant as a single beast.

On quasi-interpretations, blind abstractions and implicit complexity is another take at some sort of unification of ICC. Rather than trying to express everything with a common language, we focus here on how it is possible to compare results that are expressed in seemingly incomparable formalisms.

Usually, the power of various ICC methods is compared by "battles of examples": one shows that the new method is able to do everything that the old one could do, plus a few more examples. Hopefully, the examples are sufficiently well chosen to convince the reader that they are part of a large group of new programs that can now be analysed. This is, however, totally ad hoc and not really scientific.

In this work, we argue that we should define broad classes of programs, dubbed *transverse criteria*, and study how each ICC method interacts with them. Typically, if a method A is only able to analyse programs admitting a given criterion, while a method B can also analyse other programs, then we can say that B is more powerful than A more convincingly than by way of a handful of examples.

As a case study, we provide the transverse criterion of *blindly polynomial* programs and show that QI are only able to analyse these programs (and not all of them). Thus, any ICC method able to analyse both these and non-blindly polynomial programs would be more powerful than QI.

3 Equivalences between Programs

The second part of this dissertation deals with my current work and ideas related to ICC. The core idea started from a rereading of Rice's Theorem and revolves around the dichotomy between programs and functions that underlies Computability and Complexity.

While a program computes a single function, a given function can be computed by many different programs. Semantics studies the links between programs and functions and usually introduces a *semantic function* which, to each program, associates the function it computes. The semantic function is non-computable but is nonetheless a useful tool to study programs and programming languages.

Rather than looking a the semantics, at the correspondence between programs and functions, we choose to look at sets of programs computing the same function, that is, the inverse images of the semantic function. In this view, it appears immediately that "computing the same function" is an equivalence between programs, which we call *extensional equivalence*, or *Rice's equivalence* to emphasise the role it plays in Rice's Theorem.

Now, (computable) functions are just a convenient way of representing one class of the extensional equivalence. That is, for example, $x \mapsto x^2$ is a nice way to refer to all the programs that compute the squaring function.

Focusing on equivalences naturally raises questions around other equivalences between programs. It is well known that the set of all equivalences between programs (or natural numbers) admits a complete lattice structure. In this lattice, Rice's Theorem states that everything in the principal filter at the extensional equivalence is undecidable. Because Rice's Theorem is so nicely expressed in the language of lattice and equivalences, this view can provide good insights on computability.

In *Chains, Antichains, and Complements in Infinite Partition Lattices*, we study the structure of the lattice of equivalences itself. This is a purely order-theoretical study, and we consider equivalences over sets of any cardinality (not just countable ones such as the set of programs).

We give complete characterisations of the possible cardinals of maximal chains; maximal antichains; and sets of complements in such a lattice, with more precise results under the Generalized Continuum Hypothesis.

In *More intensional versions of Rice's Theorem*, we go back to Rice's Theorem, using the equivalence view to improve it. We mostly focus on two questions: how necessary is the "extensional set" restriction in the original Theorem? and what would happen if we replaced Rice's equivalence (*i.e.*, extensionality) by another equivalence?

We obtain generalisations of Rice's Theorem that are now able to take into account the behaviour of programs, and not just the functions they compute. Notably, we show that any decidable set of programs that contains all the programs running in polynomial time must also contain programs of arbitrarily high runtime complexity.

In *Computability in the Lattice of Equivalence Relations*, we study interactions between the lattice structure and Computability. We know that most equivalences between programs are undecidable, so we look at sets of equivalences that are defined by computability or complexity properties (*e.g.*, the set of decidable equivalences; the set of polytime equivalences; the set of recursively enumerable equivalences). A possible outcome would be to find a subset of equivalences that is manageable and correctly approximates the whole set, in a way similar to the rationals approximating the reals.

We show that sets of subrecursive equivalences are not sublattices; and that sets of equivalences in the arithmetical hierarchy are not complete sublattices. This shows that notions from Computability do not interact well with the order-theoretic structure of equivalences. Thus, we may want to use other mathematical tools to study the set of equivalences.

4 Experiments in Implicit Complexity

The third part of this dissertation focuses on the work done with my PhD student, Thomas Rubiano. We are bringing ICC results and methods into real-life compilers.

Correctly analysing a "real world" programming language is very difficult because of the large number of constructions allowed in it. For this reason, most research in ICC focuses on "toy" languages such as Term Rewriting Systems or the LOOP language. While such languages arguably capture the essence of functional or imperative programming languages, they are still far from real languages. That makes ICC analysis hard to get out of the academic spheres, and at the same time makes it hard to have a large number of examples to feed the analysis in order to study its power.

At the other end of the gap, compilers work with many different programming languages, perform complex optimisations on them, and produce machine code for various architectures. In order to cope with the specificities both of the source (programming) and target (assembly) languages, modern compilers are usually split in three parts: a front-end that transforms source code (from various languages) into an internal *Intermediate Representation* (IR); a middle-end that optimises the IR; and a back-end that transforms the IR into assembly language (for various architectures).

The Intermediate Representation, where all the work happens, is normally a simplified assembly-like language where information on program structure (types, loops, tests, . . .) is kept. It needs to be simple enough, otherwise the optimisations are too hard to write, and thus it is at a correct level of expressivity to perform ICC analysis. It needs to keep some structure, because the optimisations may use it, and thus we can also rely on that structure for our analysis.

Therefore, it appears that the IR is a good language with which to express ICC analysis and spread the ideas to a wider community. The experiments we have done, mostly in LLVM, show that it is indeed quite possible to implement these ideas and that they can provide some analysis or optimisations not already present.

In *Detection of Non-Size Increasing Programs in Compilers*, we have implemented a pass in LLVM that detects *Non Size Increasing* programs, that is programs that compute without the need of extra memory (and can reuse their own pointers rather than freeing and allocating memory all the time).

The analysis is currently not used to fuel further optimisations but nonetheless shows that Intermediate Representation is a good level to implement ICC related analysis.

In *Loop Quasi-Invariant Chunk Motion by peeling with statement composition*, we have implemented a program optimisation that is able to move large pieces of invariant code out of loops (instead of recomputing the same thing several times). The optimisation is fueled by a data-flow analysis reminiscent of the Size Change Principle or the *mwp*-polynomials.

Usual loop-invariant transformations are able to move single statements out of loops, but not more complex (and composed) ones. This transformation is notably able to move an inner loop out of the outer loop, thus actually reducing the complexity of the program.

Closing words

The articles grouped in this dissertation show a clear trend in my research towards a maturation and dissemination of ICC. Early ICC has been plagued with strong differences in the languages analysed (from the type systems of Linear Logic to the imperative Loop language). This made it hard to effectively adapt results from one team of researchers to another and slowed down the progression of ideas and techniques.

Since my PhD, I have been thinking that the various ICC techniques should be somehow merged, and I have long been working towards that goal. Unifying ICC will allow it to be used by the broader community without requiring people to learn about many different systems that seem to be so close but yet so far.

My early works were direct attempts at such a unification; my current works on equivalences form an attempt at finding "something bigger" than the individual ICC systems; my current works in compilation form an attempt to spread the ideas to a completely different community.

Being a broad selection of my works, this dissertation also mirrors my career from a young PhD to an accomplished Associate Professor. It contains works that I have done alone, with many colleagues from several countries whom I had the pleasure to meet, and with my PhD student.

Part I

Advances in Implicit Complexity

Quasi-interpretations: a way to control resources

Guillaume Bonfante, Jean-Yves Marion, Jean-Yves Moyen

Theoretical Computer Science 412(25), pp. 2776–2796, 2011

Abstract:

This paper presents in a reasoned way our works on resource analysis by quasi-interpretations. The controlled resources are typically the runtime, the runspace or the size of a result in a program execution.
Quasi-interpretations allow analyzing system complexity. A quasi-interpretation is a numerical assignment, which provides an upper bound on computed functions and which is compatible with the program operational semantics. Quasi-interpretation method offers several advantages: (i) It provides hints in order to optimize an execution, (ii) it gives resource certificates, and (iii) finding quasi-interpretations is decidable for a broad class which is relevant for feasible computations.
By combining the quasi-interpretation method with termination tools (here term orderings), we obtained several characterizations of complexity classes starting from Ptime and Pspace.

1 Introduction

This paper is part of a general investigation on program complexity analysis. We present the quasi-interpretation method which applies potentially to any formalism that can be reduced to transition systems. A quasi-interpretation gives a kind of measure by assigning to each symbol of a system a monotonic numerical function over $\mathbb{R}^+$. A quasi-interpretation possesses two main properties. First, the quasi-interpretation of a constructor term is a real which bounds its size. Second, a quasi-interpretation weakly decreases when a term is reduced.

From a practical point of view, the quasi-interpretation method is a tool to perform complexity analysis in a static way. Quasi-interpretations allow to establish an upper bound on the size of intermediate values which occur in a computation. This was used for a resource byte-code verifier in [ACGZJ04]. Moreover in the context of mobile-code or of secured application, a resource certificate can be sent which consists in the (partial) proof of the fact that a program admits a quasi-interpretation.

We restrict our study to quasi-interpretations over $\mathbb{R}^+$ which are bounded by some polynomials. A consequence of Tarski's Theorem [Tar51] is that it is decidable whether or not a program admits a **Max-Poly** quasi-interpretation which are built by combining max operator of fixed arity and polynomials of bounded degrees. This leads to an automatic synthesis procedure of a meaningful class of quasi-interpretations.

From a theoretical point of view, we combine quasi-interpretations with termination tools. We focus on simplification orderings and we consider in particular Recursive Path Orderings introduced by Dershowitz [Der82]. It turns out that we characterize the class Ptime of functions computable in polynomial time [MM00] and the class Pspace of functions computable in polynomial space [BMM01].

This work is related to Cobham [Cob62], Bellantoni and Cook [BC92], Leivant [Lei94] and Leivant-Marion [LM93a] ideas to delineate complexity classes. Note that most of the machine-independent characterizations of complexity classes have an extensional point of view. They study functions and do not pay too much attention to the algorithmic aspects. In this paper, we try an alternative way of looking at complexity classes by focusing on algorithms. In this long-term research program, the completeness problematic has moved and the nature of the problem has changed. Indeed, the class of algorithms (with respect to some encoding), say which run in polynomial time, is not recursively enumerable. So we cannot expect to characterize all Ptime algorithms. But we think that this question could shed light on the nature of computations and contribute to an intentional computability theory. Similar questions have been brought up by Caseiro [Cas97], Hofmann [Hof00a] and Jones [Jon99]. It is also worth mentioning the studies on intentionality of Colson, see for example [Col98], and of Moschovakis, as well as Gurevich.

Lastly, Marion and Péchoux suggest a new method, called sup-interpretation [MP06], which is closely related to quasi-interpretations. Sup-interpretations allow to capture more algorithms, and to characterize small parallel complexity classes [BMP06]. On the other hand, sup-interpretations do not have the nice properties of quasi-interpretations.

The paper organization is the following. The next Section introduces the first order functional programming language. The quasi-interpretations are defined next in Section 3. We suggest a classification of quasi-interpretations which induces a natural complexity hierarchy. Then, we study quasi-interpretation properties. Section 4 establishes that it is decidable if a program admits a quasi-interpretation with respect to a broad class of polynomially bounded assignments. Section 5 defines recursive path orderings used to prove termination of programs and some properties that we shall use later on. After these three sections, we state the main results at the beginning of Sections 6 and 7. Roughly speaking, the first result says that programs which terminate by product or lexicographic orderings are computable in polynomial-space. The second result means that programs that terminate by product ordering or that are tail recursive are computable in polynomial time. It is worth noticing that we have to compute the program by call by value semantics with a cache in order to have an exponential speed-up. The last Section 8 is devoted to simulations of both space and time bounded computations.

2 First order functional programming

Term rewriting systems underpin first order functional programming, that is why we refer to Dershowitz and Jouannaud survey [DJ90]. Throughout the following discussion, we consider three finite disjoint sets $\mathcal{X}, \mathcal{F}, \mathcal{C}$ of variables, function symbols and constructors.

2.1 Syntax of programs

Definition 1. The sets of terms and the rules are defined in the following way:

$$\begin{array}{llcl}
\textit{(Constructor terms)} & \mathcal{T}(\mathcal{C}) \ni v & ::= & \mathbf{c} \mid \mathbf{c}(v_1, \cdots, v_n) \\
\textit{(terms)} & \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t & ::= & \mathbf{c} \mid x \mid \mathbf{c}(t_1, \cdots, t_n) \mid \mathtt{f}(t_1, \cdots, t_n) \\
\textit{(patterns)} & \mathcal{P} \ni p & ::= & \mathbf{c} \mid x \mid \mathbf{c}(p_1, \cdots, p_n) \\
\textit{(rules)} & \mathcal{D} \ni d & ::= & \mathtt{f}(p_1, \cdots, p_n) \to t
\end{array}$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow w_i}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(w_1, \cdots, w_n)} \textit{(Constructor)}$$

$$\frac{t_i \downarrow w_i \quad \mathtt{f}(p_1, \cdots, p_n) \rightarrow r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = w_i \quad r\sigma \downarrow w}{\mathtt{f}(t_1, \cdots, t_n) \downarrow w} \textit{(Function)}$$

Figure 1: Call by value semantics with respect to a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$.

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$. We shall use a type writer font for function symbols and a bold face font for constructors.

Definition 2. A program is a quadruplet $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ such that $\mathcal{E}$ is a finite set of $\mathcal{D}$-rules. Each variable in the right-hand side of a rule also appears in the left hand side of the same rule. We distinguish among $\mathcal{F}$ a main function symbol whose name is given by the program name $\mathtt{f}$.

The set of rules induces a rewriting relation $\rightarrow$. The relation $\overset{*}{\rightarrow}$ is the reflexive and transitive closure of $\rightarrow$. Throughout, we consider orthogonal programs which is a sufficient condition in order to be confluent. Following Huet [Hue80], the program rules satisfy both conditions: (i) Each rule $\mathtt{f}(p_1, \cdots, p_n) \rightarrow t$ is left-linear, that is a variable appears only once in $\mathtt{f}(p_1, \cdots, p_n)$, and (ii) there are no two left hand-sides which are overlapping. Lastly, a ground term is a term with no variables.

2.2 Semantics

Orthogonal programs define a class of deterministic first order functional programs. The domain of the computed functions is the constructor term algebra $\mathcal{T}(\mathcal{C})$.

A substitution σ is a mapping from variables to terms. We say that it is a constructor substitution when the range of σ is $\mathcal{T}(\mathcal{C})$. We note $\mathfrak{S}$ the set of these constructor substitutions.

We consider a call by value semantics which is displayed in Figure 1. The meaning of $t \downarrow w$ is that t evaluates to a constructor term w. The program $\mathtt{f}$ computes a partial function $[\![\mathtt{f}]\!] : \mathcal{T}(\mathcal{C})^n \rightarrow \mathcal{T}(\mathcal{C})$ defined as follows. For every $v_1, \cdots, v_n \in \mathcal{T}(\mathcal{C})$, $[\![\mathtt{f}]\!](v_1, \cdots, v_n) = w$ iff $\mathtt{f}(v_1, \cdots, v_n) \downarrow w$. Otherwise, it is undefined and $[\![\mathtt{f}]\!](v_1, \cdots, v_n) = \bot$.

Notice that if $t \downarrow w$ then $t \overset{*}{\rightarrow} w$, because programs are confluent.

3 Quasi-interpretations

3.1 Quasi-interpretation definition

To approach the resource control problem, we suggest the concept of quasi-interpretation which plays the main role in this study. Quasi-interpretations have been introduced by Marion [Mar00, Mar03], Bonfante [Bon00], and Marion-Moyen [MM00]. There are related to interpretation to prove termination, an in particular to [BCMT01].

The fundamental property of a quasi-interpretation is that it is a numerical approximation from above of the size of each intermediate values (that is constructor terms), which appears in a reduction process. However, a quasi-interpretation does not give an upper bound on a term size which appears in a reduction process. A typical example is the `lcs` example in 6 whose reduction involved terms of exponential size, but the `lcs` program admits an additive quasi-interpretation, as we shall see later.

Let $\mathbb{R}^+$ be the set of non-negative real numbers. An assignment $(\!|-|\!)$ is a mapping from constructors and function symbols, that is $\mathcal{C} \bigcup \mathcal{F}$, such that for each symbol $\mathtt{f}$ of arity n it yields

1. An non-negative real number $(\!|c|\!)$ of $\mathbb{R}^+$, for every symbol c of arity 0.
2. a n-ary function $(\!|b|\!) : (\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$ for every symbol b of arity is $n > 0$.

Take a denumerable sequence $X_1, \ldots, X_n, \ldots$ We extend an assignment $(\!|-|\!)$ to terms canonically. Given a term t with n variables $x_1, \ldots, x_n$, the assignment $(\!|t|\!)$ denotes a function from $(\mathbb{R}^+)^n$ to $\mathbb{R}^+$ and is defined as follows:

$$(\!|x_i|\!) = X_i \qquad\qquad x_i \in \mathcal{X}$$
$$(\!|b(t_1, \cdots, t_n)|\!) = (\!|b|\!)((\!|t_1|\!), \cdots, (\!|t_n|\!))$$

An assignment satisfies the *subterm property* if for any $i = 1, n$ and any $X_1, \cdots, X_n$ in $\mathbb{R}^+$, we have

$$(\!| b |\!)(X_1, \cdots, X_n) \geq X_i$$

A direct consequence of the subterm property is that for any ground term s and any subterm t of s, $(\!| s |\!) \geq (\!| t |\!)$.

An assignment is *weakly monotone* if for any symbol b, $(\!| b |\!)$ is an increasing (not necessarily strictly) function with respect to each variable. That is, for every symbol b and for all $i = 1, n$ if $X_i \geq Y_i$, we have $(\!| b |\!)(X_1, \cdots, X_n) \geq (\!| b |\!)(Y_1, \cdots, Y_n)$.

A substitution σ is defined over a term t, if the domain of σ contains all variables of t. Given two terms t and s, we say that $(\!| t |\!) \geq (\!| s |\!)$ if for every constructor substitution σ defined over t and s, we have $(\!| t\sigma |\!) \geq (\!| s\sigma |\!)$.

Definition 3 (Quasi-interpretation)**.** A quasi-interpretation $(\!| - |\!)$ of a program `f` is a weakly monotonic assignment satisfying the subterm property such that for each rule $l \to r$

$$(\!| l |\!) \geq (\!| r |\!)$$

Throughout, when we shall write "quasi-interpretation", we always mean "quasi-interpretation of a program $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$".

It is worth noticing that the inequalities that defines a quasi-interpretation are not strict which differs from the notion of interpretation used to prove termination.

Proposition 4. *Assume that $(\!| - |\!)$ is a quasi-interpretation of a program $\mathtt{f}$. For any ground terms s and v such that $s \xrightarrow{*} v$, we have $(\!| s |\!) \geq (\!| v |\!)$*

Proof. A context is a particular term that we write $\mathsf{C}[\diamond]$ where $\diamond$ is a new variable. The substitution of $\diamond$ in $\mathsf{C}[\diamond]$ by a term t is noted $\mathsf{C}[t]$.

The proof goes by induction on the derivation length n. For this, suppose that $s = s_0 \to \ldots \to s_n = v$. If $n = 0$ then the result is immediate. Otherwise $n > 0$ and in this case, there is a rule $\mathtt{f}(p_1, \cdots, p_n) \to t$ and a constructor substitution σ such that $s_0 = \mathsf{C}[\mathtt{f}(p_1, \cdots, p_n)\sigma]$ and $s_1 = \mathsf{C}[t\sigma]$. Since $(\!| - |\!)$ is a quasi-interpretation, we have $(\!| t\sigma |\!) \leq (\!| \mathtt{f}(p_1, \cdots, p_n)\sigma |\!)$. The weak monotonicity property implies that $(\!| \mathsf{C}[t\sigma] |\!) \leq (\!| \mathsf{C}[\mathtt{f}(p_1, \cdots, p_n)\sigma] |\!)$. We conclude by induction hypothesis. □ □

Example 5. Given a list l of tally natural numbers, $\mathtt{sort}(l)$ sorts the elements of l by insertion. The constructor set is $\mathcal{C} = \{\mathbf{tt}, \mathbf{ff}, \mathbf{0}, \mathbf{suc}, \mathbf{nil}, \mathbf{cons}\}$.

$$\begin{aligned}
\mathtt{if}\ \mathbf{tt}\ \mathtt{then}\ x\ \mathtt{else}\ y &\to x \\
\mathtt{if}\ \mathbf{ff}\ \mathtt{then}\ x\ \mathtt{else}\ y &\to y \\
\mathbf{0} < \mathbf{suc}(y) &\to \mathbf{tt} \\
x < \mathbf{0} &\to \mathbf{ff} \\
\mathbf{suc}(x) < \mathbf{suc}(y) &\to x < y \\
\mathtt{insert}(a, \mathbf{nil}) &\to \mathbf{cons}(a, \mathbf{nil}) \\
\mathtt{insert}(a, \mathbf{cons}(b, l)) &\to \mathtt{if}\ a < b\ \mathtt{then}\ \mathbf{cons}(a, \mathbf{cons}(b, l)) \\
&\qquad \mathtt{else}\ \mathbf{cons}(b, \mathtt{insert}(a, l)) \\
\mathtt{sort}(\mathbf{nil}) &\to \mathbf{nil} \\
\mathtt{sort}(\mathbf{cons}(a, l)) &\to \mathtt{insert}(a, \mathtt{sort}(l))
\end{aligned}$$

Constructors admit the following quasi-interpretation.

$$\begin{aligned}
(\!| \mathbf{tt} |\!) = (\!| \mathbf{ff} |\!) = (\!| \mathbf{0} |\!) = (\!| \mathbf{nil} |\!) &= 0 \\
(\!| \mathbf{suc} |\!)(X) &= X + 1 \\
(\!| \mathbf{cons} |\!)(X, Y) &= X + Y + 1
\end{aligned}$$

And function symbols

$$\begin{aligned}
(\!| \mathtt{if\ then\ else} |\!)(X, Y, Z) &= \max(X, Y, Z) \\
(\!| < |\!)(X, Y) &= \max(X, Y) \\
(\!| \mathtt{insert} |\!)(X, Y) &= X + Y + 1 \\
(\!| \mathtt{sort} |\!)(X) &= X
\end{aligned}$$

This example illustrates two important facts. Quasi-interpretations can be max-functions like in the case of $<$. And, the quasi-interpretations of both sides of a rule can be the same. For example take the last rule. We see that

$$(\!| \mathtt{sort}(\mathbf{cons}(a, l)) |\!) = A + L + 1 = (\!| \mathtt{insert}(a, \mathtt{sort}(l)) |\!)$$

Example 6. Given two binary words u and v over the constructor set $\{\mathbf{a}, \mathbf{b}, \boldsymbol{\epsilon}\}$, $\mathtt{lcs}(u, v)$ returns the the length of the longest common subsequence of u and v. The expression $\mathtt{lcs}(\mathbf{ababa}, \mathbf{baaba})$ evaluates to $\mathbf{suc}^4(\mathbf{0})$ because the length longest common subsequence is 4 (take **baba**).

$$\begin{aligned}
\mathtt{max}(n, \mathbf{0}) &\to n \\
\mathtt{max}(\mathbf{0}, m) &\to m \\
\mathtt{max}(\mathbf{suc}(n), \mathbf{suc}(m)) &\to \mathbf{suc}(\mathtt{max}(n, m)) \\
\mathtt{lcs}(\boldsymbol{\epsilon}, y) &\to \mathbf{0} \\
\mathtt{lcs}(x, \boldsymbol{\epsilon}) &\to \mathbf{0} \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{i}(y)) &\to \mathbf{suc}(\mathtt{lcs}(x, y)) && \mathbf{i} \in \{\mathbf{a}, \mathbf{b}\} \\
\mathtt{lcs}(\mathbf{i}(x), \mathbf{j}(y)) &\to \mathtt{max}(\mathtt{lcs}(x, \mathbf{j}(y)), \mathtt{lcs}(\mathbf{i}(x), y)) && \mathbf{i} \neq \mathbf{j}, \mathbf{j} \in \{\mathbf{a}, \mathbf{b}\}
\end{aligned}$$

It admits the following quasi-interpretation:

- $(\!|\boldsymbol{\epsilon}|\!) = (\!|\mathbf{0}|\!) = 0$
- $(\!|\mathbf{a}|\!)(X) = (\!|\mathbf{b}|\!)(X) = (\!|\mathbf{suc}|\!)(X) = X + 1$
- $(\!|\mathtt{lcs}|\!)(X, Y) = (\!|\mathtt{max}|\!)(X, Y) = \max(X, Y)$

3.2 Taxonomy of Quasi-interpretations

Our aim is to study feasible computations. That is why we confine ourselves to programs admitting quasi-interpretations which are bounded by polynomials. We insist that assignments are bounded by polynomials, but are not necessarily polynomials.

Definition 7. An assignment $(\!|-|\!)$ is polynomial if for each symbol $b \in \mathcal{F} \bigcup \mathcal{C}$, $(\!|b|\!)$ is a function *bounded* by a polynomial.

Next, we classify polynomial assignment according to the rate of growth of constructor assignments.

Definition 8. Let $\mathbf{c}$ be a constructor of arity $n > 0$.

- An assignment of $\mathbf{c}$ is *additive* (or of kind 0) if
$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha$$
where $\alpha \geq 1$.
- An assignment of $\mathbf{c}$ is *affine* (or of kind 1) if
$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} \beta_i X_i + \alpha \qquad \alpha \geq 1$$
where the β_i's are constants of $\mathbb{R}^+$ and $\alpha > 1$.
- An assignment $\mathbf{c}$ is *multiplicative* (or of kind 2) if
$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = Q(X_1, \cdots, X_n) + \alpha \qquad \alpha \geq 1$$
where Q is a polynomial.

We classify *polynomial* assignments by the kind of assignments given to constructors, and not to function symbols. If each constructor assignment is additive (resp. affine, multiplicative) then the assignment is additive (resp. affine, multiplicative) assignment.

A program `f` admits an additive quasi-interpretation $(\!|\_|\!)$ if it is an additive assignment. We shall also just say that `f` is additive, without explicitly mentioning the additive quasi-interpretation tied to it.

Similarly, A program `f` admits an affine (resp. a multiplicative) quasi-interpretation $(\!|\_|\!)$ if it is an affine (resp. a multiplicative). We shall also just say that `f` is affine (resp. multiplicative).

In both previous examples, programs admit a polynomial quasi-interpretation because each quasi-interpretation is bounded by a polynomial. In example 5, the quasi interpretation of the function symbol $<$ is not a polynomial. The insertion sort program admits an additive quasi-interpretation because each constructor (that is the symbol in $\{\mathbf{tt}, \mathbf{ff}, \mathbf{0}, \mathbf{suc}, \mathbf{nil}, \mathbf{cons}\}$) admits an additive assignment. On the other hand, the assignment of the function symbol $<$ is not additive but it is does not matter because it is not a constructor. For the same reason, the `lcs` example admits also an additive quasi-interpretation.

Example 9. We give three programs which illustrate the three kinds of program classes delineated by quasi-interpretations.

$$\begin{aligned}
\mathtt{add}(\mathbf{0}, y) &\to y\\
\mathtt{add}(\mathbf{suc}(x), y) &\to \mathbf{suc}(\mathtt{add}(x, y))\\
\mathtt{mult}(\mathbf{0}, y) &\to \mathbf{0}\\
\mathtt{mult}(\mathbf{suc}(x), y) &\to \mathtt{add}(y, \mathtt{mult}(x, y))
\end{aligned}$$

These rules define the addition and the multiplication. They admit the following additive quasi-interpretation.

$$\begin{aligned}
(\!|\mathbf{0}|\!) &= 0\\
(\!|\mathbf{suc}|\!)(X) &= X + 1\\
(\!|\mathtt{add}|\!)(X, Y) &= X + Y\\
(\!|\mathtt{mult}|\!)(X, Y) &= X \times Y
\end{aligned}$$

So, addition and multiplication are additive programs (additivity only refers to the interpretation of constructors). Now, in order to define the exponential, we introduce another successor $\mathbf{s}$ which has an affine assignment.

$$\begin{aligned}
\mathtt{exp}(\mathbf{0}) &\to \mathbf{suc}(\mathbf{0})\\
\mathtt{exp}(\mathtt{s}(x)) &\to \mathtt{add}(\mathtt{exp}(x), \mathtt{exp}(x))
\end{aligned}$$

The quasi-interpretations of the new symbols are:

$$\begin{aligned}
(\!|\mathtt{s}|\!)(X) &= 2X + 1\\
(\!|\mathtt{exp}|\!)(X) &= X + 1
\end{aligned}$$

The above program which defines the exponential admits an affine quasi-interpretation. We see that the domain of `exp` and its co-domain are not the same. Indeed, the domain is generated by $\{\mathbf{0}, \mathbf{s}\}$ whose quasi-interpretation is affine and the co-domain is generated by $\{\mathbf{0}, \mathbf{suc}\}$ whose quasi-interpretation is additive. We shall see later on that it is necessary to have two successors with different kinds of quasi-interpretations. Similar observations have be done in [BCMT01] and on tiering system in which an argument of tier 1 produces an output of tier 0. We think that it is worth to investigate this analogy in order to interpret tiering concepts by quasi-interpretations.

We define the doubly-exponential function, i.e. $n \mapsto 2^{2^n}$, as follows (here we need yet another successor, $\mathbf{s}'$).

$$\begin{aligned}
\mathtt{dexp}(\mathbf{0}) &\to \mathbf{suc}(\mathbf{suc}(\mathbf{0}))\\
\mathtt{dexp}(\mathtt{s}'(x)) &\to \mathtt{mult}(\mathtt{dexp}(x), \mathtt{dexp}(x))
\end{aligned}$$

and

$$\begin{aligned}
(\!|\mathtt{s}'|\!)(X) &= (X + 2)^2\\
(\!|\mathtt{dexp}|\!)(X) &= X + 2
\end{aligned}$$

Again we see that the domain and co-domain are not the same. The domain admits a multiplicative quasi-interpretation and the co-domain has an additive one.

Other classes of assignments could be introduced such as elementary or primitive recursive assignments, but we will not discuss about them in this paper. This type of extensions is related to Lescanne's paper [Les92] about interpretation for termination proofs.

3.3 Elementary properties of assignments

We study now some quantitative properties of assignments when they are of the kind mentioned above. The size $|t|$ of a term t is defined by

$$|t| = \begin{cases} 0 & \text{if } t \text{ is 0-ary symbol}\\ 1 + \sum_{i=1,n} |t_i| & \text{if } t = \mathtt{f}(t_1, \ldots, t_n)\end{cases}$$

Proposition 10. *Assume that $(\!|\_|\!)$ is an additive, an affine or a multiplicative assignment. For any constructor term t in $\mathcal{T}(\mathcal{C})$, we have $|t| \leq (\!|t|\!)$.*

Proof. The proof goes by induction on the size of t. □ □

Proposition 11. *Assume that $(\!| \_ |\!)$ is an additive, an affine or a multiplicative quasi-interpretation of a program* $\mathtt{f}$*. For any term u and any constructor term $t \in \mathcal{T}(\mathcal{C})$, if $u \xrightarrow{*} t$, we have $|t| \leq (\!| u |\!)$.*

Proof. Proposition 4 implies that $(\!| u |\!) \geq (\!| t |\!)$. Then from Proposition 10, we have $|t| \leq (\!| t |\!)$. So, $|t| \leq (\!| u |\!)$. □ □

Proposition 12.

- *If $(\!| \_ |\!)$ is an additive assignment, for any constructor term t in $\mathcal{T}(\mathcal{C})$, we have $(\!| t |\!) \leq k \times |t|$.*
- *If $(\!| \_ |\!)$ is an affine assignment, for any constructor term t in $\mathcal{T}(\mathcal{C})$, we have $(\!| t |\!) \leq 2^{k \times |t|}$.*
- *If $\mathtt{f}$ is a multiplicative program, for any constructor term t in $\mathcal{T}(\mathcal{C})$, we have $(\!| t |\!) \leq 2^{2^{k \times |t|}}$.*

where in each case k is a constant which depends on the assignment $(\!| \_ |\!)$ given to constructors.

Proof. The proof goes by induction on the size of t. □ □

It is worth noticing that the above Proposition illustrates a general phenomenon that we shall see all along this paper. Roughly speaking, the complexity increases by an exponential when we jump from additive to affine quasi-interpretations, or from affine to multiplicative ones.

3.4 Call-trees

We present now call-trees which are a tool that we shall use all along. A call-tree gives a static view of an execution which captures all function calls. Hence, we can study dependencies between function calls without taking care of the extra details provided by the underlying rewriting relation.

Take a program $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$. A *state* of a program $\mathtt{f}$ is a tuple $\langle \mathtt{h}, v_1, \cdots, v_p \rangle$ where $\mathtt{h}$ is a function symbol of $\mathcal{F}$ of arity p and $v_1, \ldots, v_p$ are constructor terms of $\mathcal{T}(\mathcal{C})$. Throughout, we may omit to mention the program $\mathtt{f}$ when the context is clear.

Assume that $\eta_1 = \langle \mathtt{h}, v_1, \cdots, v_p \rangle$ and $\eta_2 = \langle \mathtt{g}, s_1, \cdots, s_m \rangle$ are two states. A *transition* is a triplet $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ such that:

(i) e is a rule $\mathtt{h}(q_1, \cdots, q_p) \to t$ of $\mathcal{E}$,

(ii) there is a constructor substitution σ such that $q_i\sigma = v_i$ for all $1 \leq i \leq p$,

(iii) there is a subterm $\mathtt{g}(s_1, \cdots, s_m)$ of t such that for any $1 \leq i \leq m$, $s_i\sigma \xrightarrow{*} s_i$ and $s_i \in \mathcal{T}(\mathcal{C})$.

The reflexive transitive closure of $\cup_{e \in \mathcal{E}} \overset{e}{\rightsquigarrow}$ is $\overset{*}{\rightsquigarrow}$.

Proposition 13. *Let $(\!| - |\!)$ be a quasi-interpretation of a program $\mathtt{f}$. Assume that $\langle \mathtt{h}, v_1, \cdots, v_p \rangle$ and $\langle \mathtt{g}, s_1, \cdots, s_m \rangle$ are two states such that*

$$\langle \mathtt{h}, v_1, \cdots, v_p \rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, s_1, \cdots, s_m \rangle$$

Then we have $(\!| \mathtt{g}(s_1, \cdots, s_m) |\!) \leq (\!| \mathtt{h}(v_1, \cdots, v_p) |\!)$ and also $(\!| s_i |\!) \leq (\!| \mathtt{h}(v_1, \cdots, v_p) |\!)$ for all $1 \leq i \leq m$.

Proof. The hypothesis $\langle \mathtt{h}, v_1, \cdots, v_p \rangle \overset{*}{\rightsquigarrow} \langle \mathtt{g}, s_1, \cdots, s_m \rangle$ means that there is a term t such that $\mathtt{h}(v_1, \cdots, v_p) \xrightarrow{*} t$ and that $\mathtt{g}(s_1, \cdots, s_m)$ is a subterm of t. Proposition 4 states $(\!| \mathtt{h}(v_1, \cdots, v_p) |\!) \geq (\!| t |\!)$. Since a quasi-interpretation satisfies the subterm property, we have $(\!| s_i |\!) \leq (\!| \mathtt{g}(s_1, \cdots, s_m) |\!) \leq (\!| \mathtt{h}(v_1, \cdots, v_p) |\!)$. □ □

Next, we define the $\langle \mathtt{g}, s_1, \cdots, s_m \rangle$-call tree as a tree where (i) $\langle \mathtt{g}, s_1, \cdots, s_m \rangle$ is the root. (ii) the set of nodes is $\{\eta \mid \langle \mathtt{h}, v_1, \cdots, v_p \rangle \overset{*}{\rightsquigarrow} \eta\}$ and (iii) there is an edge between the state η_1 and the state η_2 if $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$.

Some state may actually appear several time in the tree. This happen typically in two cases: when there is a loop in the computation or when there are two different sequences of call leading to the same one (such as with the Fibonacci function). We do not merge identical states in the call-tree, hence nodes are occurrences of a state rather that a state alone.

Keeping several occurrences of the same state is useful because here we need to mimic the call by value semantics of Figure 1. This semantics actually performs identical calls several times. In Section 7 identical nodes in a call tree will be merged and the tree will thus turn into a directed acyclic graph.

The size of a state $\langle \mathtt{g}, s_1, \cdots, s_m \rangle$ is $\sum_{i=1}^{m} |s_i|$.

Lemma 14. *Let $(\!| - |\!)$ be an additive (or affine, or multiplicative) quasi-interpretation of a program f. The size of each node of the $\langle f, t_1, \cdots, t_n\rangle$-call graph is bounded by $d \times (\!| f(t_1, \cdots, t_n) |\!)$ where d is the maximal arity of a function symbol in f.*

Proof. Suppose that $\langle \mathtt{g}, s_1, \cdots, s_m\rangle$ is a state of the $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$-call graph. It follows from Proposition 13 that $(\!| s_i |\!) \leq (\!| \mathtt{f}(t_1, \cdots, t_n) |\!)$. As each s_i is a constructor term, Proposition 10 entails that $|s_i| \leq (\!| s_i |\!)$. Therefore

$$|\langle \mathtt{g}, s_1, \cdots, s_m\rangle| = \sum_{i=1,n} |s_i| \leq d \times (\!| \mathtt{f}(t_1, \cdots, t_n) |\!)$$

where d is the maximal arity of a function symbol. □ □

3.5 Upper bound on the complexity

It turns out that we can now state a quite important practical point. Indeed, consider an additive program $\mathtt{f}$. Then, the quasi-interpretation of $(\!| \mathtt{f}(t_1, \cdots, t_n) |\!)$ is bounded by a polynomial in the input size, that is in $\sum_{i=1,n}(|t_i|)$. Next, by combining Lemma 14, we deduce that the size of each state of the $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$-call tree is bounded by a polynomial in the input size. Because, the size of each state is bounded by the quasi-interpretation of the root.

Theorem 15. *Assume that f is a program. For any constructor terms $t_1, \ldots, t_n$,*

- *If f is an additive program, the size of each state of the $\langle f, t_1, \cdots, t_n\rangle$-call tree is bounded by $P(m)$ where P is some polynomial.*
- *If f is an affine program, the size of each state of the $\langle f, t_1, \cdots, t_n\rangle$-call tree is bounded by $2^{k\times m}$ where k is some constant.*
- *If f is a multiplicative program, the size of each state of the $\langle f, t_1, \cdots, t_n\rangle$-call tree is bounded by $2^{2^{k\times m}}$ where k is some constant.*

where $m = \max_{i=1,n} |t_i|$.

Proof. It is a consequence of Lemma 14 and Proposition 12. □ □

From this result, we can see that the halting problem on a given input is decidable, thus leading to a potential runtime detection of non-termination. In [Ama03], Amadio wrote a first proof of the result above.

Theorem 16. *There is an evaluation procedure which, given an additive program f and given n constructor terms $t_1, \cdots, t_n$, computes the value w if $f(t_1, \cdots, t_n) \downarrow w$ and otherwise returns $\perp$, that is if the evaluation does not terminate. This evaluation procedure runs in exponential time, i.e. in $2^{P(\max_{i=1}^{n} |t_i|)}$, where P is a polynomial.*

Proof. We build a call by value evaluator with a deterministic Turing machine with an extra tape which behaves as a stack in order to evaluate $\mathtt{f}(t_1, \cdots, t_n)$. The stack is used for the recursive calls and the normal tapes contain the current context. Actually, a call by value procedure computes the value of each state of $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$-call graph and for this we perform breadth-first exploration. A context corresponds to a state and the number of states that we have to memorize is bounded by the width of the call-graph. And the width is bounded by te maximal arity d of a function symbol. So, the space use on the space is bounded by $k' \times d \times (\!| \mathtt{f}(t_1, \cdots, t_n) |\!)$ for some constant k'. (Notice that here, we do not take into consideration the size of the stack.) Cook's theorem [Coo71] implies that the call by value evaluator can be then simulated in time $2^{K\times (\!| \mathtt{f}(t_1,\cdots,t_n) |\!)}$ for some constant K (which depends on k' and d). Since the program admits a polynomial quasi-interpretation, the time is bounded by $2^{P(\max_{i=1}^{n} |t_i|)}$, where $P(X) = K \times (\!| \mathtt{f} |\!)(kX, \ldots, kX)$ by Proposition 12. □ □

Corollary 17. *There is an evaluation procedure which given an affine (resp. multiplicative) program f and n constructor terms $t_1, \cdots, t_n$, computes the value w if $t \downarrow w$ and otherwise returns $\perp$ in double exponential time, i.e. in $2^{2^{K'\times \max_{i=1}^{n} |t_i|}}$ (resp. in triple exponential time, i.e. in $2^{2^{2^{K''\times \max_{i=1}^{n} |t_i|}}}$), where K' and K'' are two constants.*

3.6 Uniform Termination is undecidable

Quasi-interpretations do not ensure termination. Indeed, the rule $\mathtt{f}(x) \to \mathtt{f}(x)$ admits the quasi-interpretation $(\!| \mathtt{f} |\!)(X) = X$ but does not terminate. Moreover, quasi-interpretations do not give enough information to decide uniform termination as stated in the following theorem.

Theorem 18. *It is undecidable to know whether a program which admits a polynomial quasi-interpretation, terminates or not on all inputs.*

Proof. Senizergues proved in [Sen95] that the uniform termination of non-increasing semi-Thue systems is undecidable. These semi-Thue systems are a particular case of rewriting systems with a quasi-interpretation (simply take the identity polynomial for the unary symbols and 1 for the unique constant ϵ) . The conclusion follows immediately. □ □

4 Synthesis of Quasi-interpretations

We consider now the problem of finding program quasi-interpretations, which is an important practical question. For this, we restrict assignments to the class **Max-Poly**. The class of **Max-Poly** functions contains constant functions ranging over non-negative rationals and is closed by projections, maximum, addition, multiplication and composition.

We establish as a direct consequence of Tarski's Theorem [Tar51] that finding a program quasi-interpretation, which belongs to the class **Max-Poly** over non-negative real numbers, is decidable, when degrees are fixed. Indeed, Tarski demonstrated that the first-order theory for reals containing the addition $+$, the multiplication $\times$, the equality $=$, the order $>$ with variables over reals and rational constants is decidable. On the other hand, the same question over natural numbers is undecidable because it is a consequence of Matiyasevich's Theorem [Mat93].

We consider two related problems. The first one is the *verification problem*:

inputs: A program `f` and an assignment $(\!|-|\!)$.

problem: Is $(\!|-|\!)$ a quasi-interpretation for `f`?

The second one is the *synthesis problem*:

input: A program `f`.

problem: Is there an assignment $(\!|-|\!)$ which is a quasi-interpretation for `f`?

Before proceeding to the main discussion, it is convenient to have a normal representation of function in **Max-Poly**.

Proposition 19 (Normalization)**.** *A **Max-Poly** function Q can always be expressed as:*

$$Q(X_1, \dots, X_n) = \max(P_1(X_1, \dots, X_n), \dots, P_k(X_1, \dots, X_n))$$

where each P_i is a polynomial. We say that the max*-degree of Q is k and the degree of Q is the maximum degree of the polynomials $P_1, \dots, P_k$.*

Proof. This is due to the fact that max is distributive with $+$ and $\times$ over the non-negative reals. □ □

Now consider a **Max-Poly** assignment $(\!|-|\!)$ of a program `f`. Take a rule $l \to r$ and define

$$S_{l\to r} = \forall X_1, \dots X_p \geq 0 : \bigvee_{i=1..n} \bigwedge_{j=1..m} P_i(X_1, \dots, X_p) \geq Q_j(X_1, \dots, X_p)$$

where $(\!|l|\!) = \max(P_1, \dots, P_n)$, $(\!|r|\!) = \max(Q_1, \dots, Q_m)$ and $X_1, \dots, X_p$ are all the variables of $(\!|l|\!)$. (Recall that the variables of $(\!|r|\!)$ are also variables of $(\!|l|\!)$.)

We see that the first order formula $S_{l\to r}$ is true iff $(\!|l|\!) \geq (\!|r|\!)$.

Theorem 20. *The verification problem for **Max-Poly** assignments is decidable in exponential time in the size of the program.*

Proof. In order to solve the verification problem, we have to decide whether or not the following first order formula is true.

$$S_{\mathcal{E}} = \bigwedge_{l\to r \in \mathcal{E}} S_{l\to r} \qquad \text{for a given assignment}$$

This is performed by Tarski's decision procedure. Basu, Pollack and Roy [BPR96] established that such procedure is at most exponential in the number of quantifiers. In our case, it corresponds to the maximum arity of symbols. □ □

Theorem 21. *The synthesis problem for **Max-Poly** assignment of bounded degree and bounded* max*-degree is decidable in doubly exponential time in the size of the program.*

Proof. Without loss of generality, we restrict ourselves to unary functions. Functions with many variables are handled in the same way but with more coefficients and indexes. By Theorem hypothesis, we assume that the degree is d and the max-degree is k.

Suppose that there are n symbols, constructors or functions, $b_1, \dots, b_n$. The assignment of b_i is of the form

$$(\!|b_i|\!)(X) = \max(P_1^{b_i}(X), \dots, P_k^{b_i}(X)) \qquad \text{where } P_m^{b_i} = \sum_{j=0}^{d} a_{b_i,m,j} X^j$$

Now, we have to guess polynomial coefficients by proving the validity of the formula:

$$\exists a_{b_1,1,0} \dots a_{b_1,k,d}, \dots, a_{b_n,1,0}, \dots, a_{b_n,k,d} : S_{\mathcal{E}}$$

where $S_{\mathcal{E}}$ is defined in the previous proof. Lastly, we need to verify that the subterm and the weak monotonicity properties and the fact that the coefficient of degree 0 for constructors is ≥ 1.

The total number of quantifiers is $k \times (d+1) \times n$. So, the decision procedure is doubly exponential in the size of the program. □ □

Remark 22. The quasi interpretations of all examples belong to the class **Max-Poly**. Actually, it appears that the class of **Max-Poly** quasi-interpretations is sufficient for daily programs. In practice, each program appears to admit a **Max-Poly** quasi-interpretation with low degrees, usually no more than 2 for both the degree of polynomials and the arity of max.

Although a solution of the decision of **Max-Poly** synthesis problem is presented above, yet the procedure for carrying out the decision is complex. There is need of specific methods for finding quasi-interpretations which are in a smaller class but which are relevant. For this reason, Amadio [Ama03] considered the max-plus algebra over rational numbers. A program which admits a quasi-interpretation over the max-plus algebra are related to non-size increasing according to Hofmann [Hof99]. Amadio established that the synthesis of max-plus quasi-interpretation is in NPtime-hard and NPtime-complete in the case of multi-linear assignments.

5 Termination

We now focus on termination which plays the role of a mold capturing certain algorithm patterns. We obtain a finer control resource by the combination of termination tools and quasi-interpretations. Here, we consider Recursive Path Orderings which are simplification orderings and so well-founded. Among the pioneers of this subject, there are Plaisted [Pla78], Dershowitz [Der82], Kamin and Lévy [KL80]. Finally, Krishnamoorthy and Narendran in [KN85] have proved that deciding whether a program terminates by Recursive Path Orderings is a NP-complete problem.

5.1 Extension of an ordering to sequences

Suppose that $\preceq$ is a partial ordering and $\prec$ its strict part. We describe two extensions of $\prec$ to sequences of the same length.

Definition 23. The product extension[1] of $\prec$ over sequences, noted $\prec^p$, is defined as follows.
We have $(m_1, \cdots, m_k) \prec^p (n_1, \cdots, n_k)$ if and only if (i) $\forall i \leq p, m_i \preceq n_i$ and (ii) $\exists j \leq k$ such that $m_j \prec n_j$.

Definition 24. The lexicographic extension of $\prec$, noted $\prec^l$, is defined as follows.
We have $(m_1, \cdots, m_k) \prec^l (n_1, \cdots, n_k)$ if and only if there exists an index j such that (i) $\forall i < j, m_i \preceq n_i$ and (ii) $m_j \prec n_j$.

The product ordering of sequences in a restriction of the more usual multi-set ordering of sequences. We do not need here the full power of the multi-set ordering mainly because we only compare sequences of the same length will the multi-set ordering works on sequences of any length.

Notice that the product ordering of sequences is also a restriction of the lexicographic ordering, that is two sequences ordered by the product extension are also ordered lexicographically.

5.2 Recursive path ordering with status

Let $\prec_{\mathcal{F}}$ be an ordering on $\mathcal{F}$ and $\approx_{\mathcal{F}}$ be a compatible equivalence relation such that if $\texttt{f} \approx_{\mathcal{F}} \texttt{g}$ then $\texttt{f}$ and $\texttt{g}$ have the same arity. The quasi-ordering $\preceq_{\mathcal{F}} = \prec_{\mathcal{F}} \cup \approx_{\mathcal{F}}$ is a precedence over $\mathcal{F}$.

Definition 25. A status *st* is a mapping which associates to each function symbol $\texttt{f}$ of $\mathcal{F}$ a status $st(\texttt{f})$ in $\{p, l\}$. A status is compatible with a precedence $\preceq_{\mathcal{F}}$ if it satisfies the fact that if $\texttt{f} \approx_{\mathcal{F}} \texttt{g}$ then $st(\texttt{f}) = st(\texttt{g})$.

Throughout, we assume that status are compatible with precedences.

Definition 26. Given a precedence $\preceq_{\mathcal{F}}$ and a status *st*, the recursive path ordering $\prec_{rpo}$ is defined in Figure 2.

When $st(\texttt{f}) = p$, the status of $\texttt{f}$ is said to be product. In that case, the arguments are compared with the product extension of $\prec_{rpo}$. Otherwise, the status is said to be lexicographic.

A program is ordered by $\prec_{rpo}$ if there is a precedence on $\mathcal{F}$ and a status *st* such that for each rule is decreasing, that is each rule $l \to r$, we have $r \prec_{rpo} l$.

[1] Unlike [MM00], we have decided to present the product extension instead of the permutation extension. This simplifies the presentation without loss of generality. Actually, there is a tedious procedure to transform the rules in order to prove termination by product ordering.

$$\frac{s = t_i \ or \ s \prec_{rpo} t_i}{s \prec_{rpo} \mathtt{f}(\ldots, t_i, \ldots)} \ \mathtt{f} \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{\forall i \ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathbf{c}(s_1, \cdots, s_m) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \ \mathtt{f} \in \mathcal{F}, \mathbf{c} \in \mathcal{C}$$

$$\frac{\forall i \ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n) \qquad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}}{g(s_1, \cdots, s_m) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \ \mathtt{f}, \mathtt{g} \in \mathcal{F}$$

$$\frac{(s_1, \cdots, s_n) \prec_{rpo}^{st(\mathtt{f})} (t_1, \cdots, t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \qquad \forall i \ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(s_1, \cdots, s_n) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)} \ \mathtt{f}, \mathtt{g} \in \mathcal{F}$$

Figure 2: Definition of $\prec_{rpo}$

Theorem 27 (Dershowitz [Der82]). *Each program which is ordered by $\prec_{rpo}$ terminates on all inputs.*

Example 28.

1. The `shuffle` program rearranges two words. It terminates with a product status.

$$\begin{aligned} \mathtt{shuffle}(\boldsymbol{\epsilon}, y) &\to y \\ \mathtt{shuffle}(x, \boldsymbol{\epsilon}) &\to x \\ \mathtt{shuffle}(\mathbf{i}(x), \mathbf{j}(y)) &\to \mathbf{i}(\mathbf{j}(\mathtt{shuffle}(x, y))) \qquad \mathbf{i}, \mathbf{j} \in \{\mathbf{0}, \_\} \end{aligned}$$

2. The following program reverses a word by tail-recursion. It terminates with a lexicographic status.

$$\begin{aligned} \mathtt{reverse}(\boldsymbol{\epsilon}, y) &\to y \\ \mathtt{reverse}(\mathbf{i}(x), y) &\to \mathtt{reverse}(x, \mathbf{i}(y)) \qquad \mathbf{i} \in \{\mathbf{0}, \_\} \end{aligned}$$

3. The program `sort` of Example 5 terminates if each function symbol has a product status and by setting the precedence `if then else` $\prec_{\mathcal{F}}$ `insert` $\prec_{\mathcal{F}}$ `sort`

4. The `lcs` program of Example 6 is ordered by taking `max` $\prec_{\mathcal{F}}$ `lcs`, and both symbols have a product status.

5.3 Extensional Characterization

The orderings considered are special cases of more general ones and in particular of *Multiset Path Ordering* and *Lexicographic Path Ordering.* Nevertheless, they characterize the same set of functions. Say that a $\mathrm{RPO}_{\mathrm{Pro}}$-program is a program in which each function symbol has a product status. Following the result of Hofbauer [Hof92], we have[2]

Theorem 29. *The set of functions computed by RPO_{Pro}-programs is exactly the set of primitive recursive functions.*

Now, say that a $\mathrm{RPO}_{\mathrm{Lex}}$-program is a program in which each function symbol has a lexicographic status. Weiermann [Wei95] has established that[3]

Theorem 30. *The set of functions computed by RPO_{Lex}-programs is exactly the set of multiple-recursive functions.*

5.4 Consequences of termination proofs

We write $s \trianglelefteq t$ to say that s is a subterm of t.

Proposition 31.

1. *For each constructor term t and s, $s \prec_{rpo} t$ iff $s \triangleleft t$.*

[2]Since the product ordering is a restriction of the multiset ordering, any $\mathrm{RPO}_{\mathrm{Pro}}$-program is also terminating by the more usual MPO termination ordering. Conversely, as stated above, there is a (somewhat tedious) procedure to turn MPO programs into $\mathrm{RPO}_{\mathrm{Pro}}$-programs.

[3]here, the $\mathrm{RPO}_{\mathrm{Lex}}$-programs are exactly the programs terminating by the usual LPO termination ordering.

2. *For each constructor term $s_1, \cdots, s_n$ and $t_1, \cdots, t_n$,*
   $(s_1, \cdots, s_n) \prec^x_{rpo} (t_1, \cdots, t_n)$ implies $(s_1, \cdots, s_n) \lhd^x (t_1, \cdots, t_n)$, where x is a status p or l and $\lhd^x$ is the corresponding extension based on the subterm relation.

3. *For each constructor term $s_1, \cdots, s_n$ and $t_1, \cdots, t_n$,*
   $(s_1, \cdots, s_n) \prec^x_{rpo} (t_1, \cdots, t_n)$ implies $(|s_1|, \cdots, |s_n|) <^x (|t_1|, \cdots, |t_n|)$, where x is a status p or l and $<^x$ is the corresponding extension of the ordering over natural numbers.

Proof. The proofs go by induction on the size of terms. □ □

Remark 32. The ordering $\prec_{rpo}$ is not stable for constructor contexts. Indeed, we have $\underline{(}\epsilon) \prec_{rpo} \mathbf{0}(\underline{(}\epsilon))$, but $\underline{(}\underline{(}\epsilon)) \prec_{rpo} \underline{(}\mathbf{0}(\underline{(}\epsilon)))$ does not hold. So, $\prec_{rpo}$ is not a reduction ordering but there is no rewriting inside constructor terms.

Lemma 33. *Let f be a program which is ordered by $\prec_{rpo}$, α be the number of function symbols and d be the maximal arity of function symbols. Assume that the size of each state of the $\langle f, t_1, \cdots, t_n\rangle$-call tree is strictly bounded by A. Then the following facts hold:*

1. *If $\langle f, t_1, \cdots, t_n\rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \cdots, s_m\rangle$ then*
   (a) $g \prec_{\mathcal{F}} f$ or
   (b) $g \approx_{\mathcal{F}} f$ and $(s_1, \cdots, s_m) \prec^{st(f)}_{rpo} (t_1, \cdots, t_n)$.
2. *If $\langle f, t_1, \cdots, t_n\rangle \overset{*}{\rightsquigarrow} \langle g, s_1, \cdots, s_m\rangle$ and $g \approx_{\mathcal{F}} f$ then the number of states between the states $\langle f, t_1, \cdots, t_n\rangle$ and $\langle g, s_1, \cdots, s_m\rangle$ is bounded by A^d.*
3. *The length of each branch of the call-tree is bounded by $\alpha \times A^d$.*

Proof.

1. Because the rules of the program decrease by $\prec_{rpo}$.
2. Suppose that $\langle \mathtt{h}, v_1, \cdots, v_p\rangle$ is a state between $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$ and $\langle \mathtt{g}, s_1, \cdots, s_m\rangle$. Due to the first point of this lemma, we have $\mathtt{h} \approx_{\mathcal{F}} \mathtt{f}$ and $(v_1, \cdots, v_p) \prec^{st(\mathtt{f})}_{rpo} (t_1, \cdots, t_n)$. So, by proposition 31(3), we have $(|v_1|, \cdots, |v_p|) <^{st(\mathtt{f})} (|t_1|, \cdots, |t_n|)$. Since the size of each component is bounded by A and $n \leq d$, the length of the decreasing chain is bounded by A^d.
3. In each branch, the previous point of the Lemma claims that there are at most A^d states whose function symbols have the same precedence. Next, there are A^d states whose function symbols have the precedence immediately below, and so on. As there are only α function symbols, the length of the branch is bounded by $\alpha \times A^d$.

□ □

6 Characterizing space bounded computation

6.1 Polynomial space computation

Definition 34. A RPO^{QI}-program is a program that (i) admits a quasi-interpretation and (ii) which terminates by $\prec_{rpo}$.

Theorem 35. *The set of functions computed by* additive *RPO^{QI}-programs is exactly the set of functions computable in* polynomial space.

The upper-bound on space-usage is established by Theorem 38. The completeness of this characterization is established by Theorem 55.

Example 36. The Quantified Boolean Formula (QBF) problem is PSPACE complete. It consists in determining the validity of a boolean formula with quantifiers over propositional variables. Without loss of generality, we restrict formulae to $\neg, \vee, \exists$. QBF problem is solved by the following program.

$$\begin{aligned}
\mathtt{not}(\mathbf{tt}) &\to \mathbf{ff} & \mathtt{not}(\mathbf{ff}) &\to \mathbf{tt}\\
\mathtt{or}(\mathbf{tt}, x) &\to \mathbf{tt} & \mathtt{or}(\mathbf{ff}, x) &\to x\\
\mathbf{0} = \mathbf{0} &\to \mathbf{tt} & \mathbf{suc}(x) = \mathbf{0} &\to \mathbf{ff}\\
\mathbf{0} = \mathbf{suc}(y) &\to \mathbf{ff} & \mathbf{suc}(x) = \mathbf{suc}(y) &\to x = y\\
\mathtt{in}(x, \mathbf{nil}) &\to \mathbf{ff} & \mathtt{in}(x, \mathbf{cons}(a, l)) &\to \mathtt{or}(x = a, \mathtt{in}(x, l))
\end{aligned}$$

$$\begin{aligned}
\mathtt{verify}(\mathbf{Var}(x), t) &\to \mathtt{in}(x, t)\\
\mathtt{verify}(\mathbf{Not}(\phi), t) &\to \mathtt{not}(\mathtt{verify}(\phi, t))\\
\mathtt{verify}(\mathbf{Or}(\phi_1, \phi_2), t) &\to \mathtt{or}(\mathtt{verify}(\phi_1, t), \mathtt{verify}(\phi_2, t))\\
\mathtt{verify}(\mathbf{Exists}(n, \phi), t) &\to \mathtt{or}(\mathtt{verify}(\phi, \mathbf{cons}(n, t)), \mathtt{verify}(\phi, t))\\
\mathtt{qbf}(\phi) &\to \mathtt{verify}(\phi, \mathbf{nil})
\end{aligned}$$

Booleans are encoded by $\{\mathbf{tt}, \mathbf{ff}\}$, variables are encoded by unary integers which are generated by $\{\mathbf{0}, \mathbf{suc}\}$. Formulae are built from $\{\mathbf{Var}, \mathbf{Not}, \mathbf{Or}, \mathbf{Exists}\}$. All these symbols are constructors. The main function symbol is `qbf`.

Rules are ordered by $\prec_{rpo}$ by putting

$$\{\mathtt{not}, \mathtt{or}, \mathtt{\_=\_}\} \prec_{\mathcal{F}} \mathtt{in} \prec_{\mathcal{F}} \mathtt{verify} \prec_{\mathcal{F}} \mathtt{qbf}$$

and each function symbol has a product status except `verify` which has a lexicographic status.

They admit the following additive quasi-interpretations :

$$\begin{aligned}
(\!|\mathbf{c}|\!) &= 0 && \text{where } \mathbf{c} \text{ is a constructor of arity } 0\\
(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) &= 1 + \sum_{i=1}^{n} X_i && \text{where } \mathbf{c} \text{ is a constructor of arity} > 0\\
(\!|\mathtt{verify}|\!)(\Phi, T) &= \Phi + T\\
(\!|\mathtt{qbf}|\!)(\Phi) &= \Phi + 1\\
(\!|\mathtt{g}|\!)(X_1, \cdots, X_n) &= \max_{i=1}^{n} X_i && \text{for other function symbols}
\end{aligned}$$

6.2 RPO^{QI}-programs are Pspace computable

We are now establishing that a RPO^{QI}-program `f` is computable in polynomial space.

Lemma 37. *Let f be a RPO^{QI}-program. For each constructor term $t_1, \cdots, t_n$, the space used by a call by value interpreter to compute $f(t_1, \cdots, t_n)$ is bounded by a polynomial in $(\!|f(t_1, \cdots, t_n)|\!)$.*

Proof. Take an innermost call by value interpreter, like the one of Figure 1. It builds recursively in a depth first manner the $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$-call tree, evaluates nodes and backtracks. Put $A = (\!|\mathtt{f}(t_1, \cdots, t_n)|\!)$. The interpreter only needs to store states along a branch of the call-tree. Each state as well as the intermediate results are bounded by $O(A)$. The maximal length of a branch is bounded by $\alpha \times A^d$ by Lemma 33(3). The number of states and results to memorize for the depth first search is bounded by $\alpha \times A^{d+1} \times \beta$ where β is the maximal size of a rule. In other words, β is an upper bound on the width of the call-tree. Therefore, the space used by the interpreter is bounded by $O(A^{d+1})$. □ □

Theorem 38. *Let f be an additive RPO^{QI}-program. For each constructor term $t_1, \cdots, t_n$, the space used by a call by value interpreter to compute $f(t_1, \cdots, t_n)$ is bounded by a polynomial in $\max_{i=1}^{n} |t_i|$.*

Proof. By Proposition 12, we have $(\!|t_i|\!) \leq O(|t_i|)$. Because quasi-interpretations are polynomially bounded, we have $(\!|\mathtt{f}(t_1, \cdots, t_n)|\!) \leq P(\max_{i=1}^{n} |t_i|)$, for some polynomial P. So the space is bounded by $O(P(\max_{i=1}^{n} |t_i|)^{d+1})$ following Lemma 37. □ □

6.3 Beyond polynomial space

The kind of constructor quasi-interpretations provides an upper bound on the space required to evaluate a program.

Theorem 39.

- *The set of functions computed by* affine *RPO^{QI}-programs is exactly the set of functions computable in* linear exponential space*, that is in space bounded by $2^{O(n)}$ where n is the size of the inputs.*
- *The set of functions computed by* multiplicative *RPO^{QI}-programs is exactly the set of functions computable in* linear double exponential space*, that is in space bounded by $2^{2^{O(n)}}$ where n is the size of the inputs..*

Proofs are very similar to the one of Theorem 38. The kind of quasi-interpretation gives the different upper-bounds on the space-usage as established in Proposition 12. The converse is established by Theorems 58 and 61.

7 Characterizing time bounded computation

7.1 Polynomial time computation

Definition 40. A function symbol f is linear in a program terminating by $\prec_{rpo}$ if for each rule $\mathtt{f}(p_1, \cdots, p_n) \to r$, then there is at most one occurrence in r of a function symbol g with the same precedence than f, that is $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$.

Definition 41.

1. A $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program is a program that (i) admits a quasi-interpretation, (ii) which terminates by $\prec_{rpo}$ and (iii) each function symbol has a product status.
2. A $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program is a program that (i) admits a quasi-interpretation, (ii) which terminates by $\prec_{rpo}$, and (iii) each function symbol is linear and has a lexicographic status.
3. A $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro+Lin}}$-program is a program that (i) admits a quasi-interpretation, (ii) which terminates by $\prec_{rpo}$ and (iii) each function symbol which has a lexicographic status is linear, and others have a product status.

Tail recursive programs are $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-programs as it is illustrated by the `reverse` program in Example 28. On the other hand, the program that solves QBF in Example 36, is not a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program, because of the definition of `verify` (in the case of $\mathbf{Exists}(n, \phi)$) which leads to two recursive calls with substitution of parameters. Note that lexicographic ordering captures the template of recursion with parameter substitutions which was the key ingredient of the characterization of polynomial space functions [LM95] by tiering discipline.

Theorem 42. *The set of functions computed by* additive *RPO^{QI}_{Lin}-programs (resp. RPO^{QI}_{Pro}-programs and $RPO^{QI}_{Pro+Lin}$-programs) is exactly the set of functions computable in* polynomial time.

The upper-bound on time-usage is established by Theorem 47 below. The completeness of this characterization is established by Theorem 54.

Example 43. The `lcs` example 6 is quite interesting and is an illustration of an important observation. Indeed, if one applies the rules of the program following a call by value strategy, one gets an exponentially long derivation chain. But the theorem states that the `lcs` function is computable in polynomial time. Actually, one should be careful not to confuse the algorithm and the function it computes. This function (length of the longest common subsequence) is a classical textbook example of so called "dynamic programming" (see chapter 16 of [CLR90]) and can in this way be computed in polynomial time.

So, the theorem does not characterize the complexity of the algorithm, which we should call its *explicit* complexity but the complexity of the function computed by this algorithm, which we should dub its *implicit* complexity.

7.2 $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro+Lin}}$-programs are PTIME computable

In order to avoid an exponential explosion like, for instance, in the `lcs` case, we switch from the call-by-value semantics previously defined to a call-by-value semantics with cache, see Figure 4. Hence, we simulate dynamic programming techniques, which consist in storing each result of a function call in a table and avoiding to recompute the same function call if it is already in the table. This technique is inspired from Andersen and Jones' rereading ([AJ94]) of Cook simulation technique over 2 way push-down automata ([Coo71]) and is called memoization.

The expression $\langle C, t\rangle \Downarrow \langle C', w\rangle$ means that the computation of t is w given a program f and an initial cache C. The final cache C' contains C and each call which has been necessary to complete the computation.

More precisely, say that a configuration is a list such as $(\mathtt{g}, w_1, \cdots, w_m, w)$ where $\langle \mathtt{g}, w_1, \cdots, w_m\rangle$ is a state, and $[\![\mathtt{g}]\!](w_1, \cdots, w_m) = w$. When a term $\mathtt{g}(w_1, \cdots, w_m)$ is considered, we search for a configuration $(\mathtt{g}, w_1, \cdots, w_m, w)$ in the current cache C. If such configuration exists, we use it to short-cut the computation and so we return w. Otherwise, we apply a program equation, say $l \to r$, by matching $\mathtt{g}(w_1, \cdots, w_m)$ with l. Then, we update C by adding the configuration $(\mathtt{g}, w_1, \cdots, w_m, w)$ to the current cache C.

Figure 3 shows what happens to the $\langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{baaba}\rangle$-call tree when memoization is applied. Notice that identical subtrees are merged and the call-tree becomes a directed acyclic graph.

The key point for additive programs, is to establish that the size of a cache C is polynomially bounded in the size of the input arguments.

Lemma 44. *Suppose that $\langle C_0, f(t_1, \cdots, t_n)\rangle \Downarrow \langle C, w\rangle$. The size of the final cache C is bounded by a polynomial in $(\!| f(t_1, \cdots, t_n) |\!)$.*

Proof. Define $C_{\mathtt{g}}$ as the set of m-uplets of $\mathcal{T}(\mathcal{C})$-terms which are the arguments of states of g. That is, $(u_1, \cdots, u_m) \in C_{\mathtt{g}}$ iff $(\mathtt{g}, u_1, \cdots, u_m, v) \in C$. We have

$$\#C = \sum_{\mathtt{g} \in \mathcal{F}} \#C_{\mathtt{g}}$$

where we write $\#S$ for the cardinal of a set S.

To give an upper-bound on the cardinality of $C_{\mathtt{g}}$, we define two sets $C^{\vee}_{\mathtt{g}}$ and $C^{\wedge}_{\mathtt{g}}$. The idea is to separate the calls which come from functions of strictly higher precedence and the ones which come from functions of the same precedence. Consider the $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$-call tree. Say that the covering graph of $\mathtt{g}$ is the subgraph of the $\langle \mathtt{f}, v_1, \cdots, v_n\rangle$-call tree obtained by removing all states which are not labeled by functions $\mathtt{h}$ which has precedence equivalent to $\mathtt{g}$, that $\mathtt{h} \approx_{\mathcal{F}} \mathtt{g}$. Define two sets $C^{\vee}_{\mathtt{g}}$ and $C^{\wedge}_{\mathtt{g}}$ as follows. $C^{\vee}_{\mathtt{g}}$ contains all the roots of the covering graph of $\mathtt{g}$ labeled by $\mathtt{g}$, and $C^{\wedge}_{\mathtt{g}}$ contains all the other nodes of the covering graph labeled by $\mathtt{g}$.

- We consider the $C^{\vee}_{\mathtt{g}}$'s. Suppose that $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, but $\mathtt{f} \neq \mathtt{g}$. For the special case where $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$, we have $\#C^{\vee}_{\mathtt{f}} = 1$. By definition, we have $\#C^{\vee}_{\mathtt{g}} = 0$.
  Suppose that $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$. Then, $(u_1, \cdots, u_m) \in C^{\vee}_{\mathtt{g}}$. It follows that the cardinality of $C^{\vee}_{\mathtt{g}}$ is bounded by
  $$\#C^{\vee}_{\mathtt{g}} \leq \sum_{\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}} \#C_{\mathtt{f}}$$
- We consider $C^{\wedge}_{\mathtt{g}}$.
  1. The status of $\mathtt{g}$ is product. Proposition 31 states that sub-calls of the same rank starting from $\mathtt{f}(v_1, \cdots, v_n)$ have arguments which are subterms of the v_i's. Therefore there are at most $\prod_{i \leq n}(|v_i| + 1)$ such sub-calls. It follows from Lemma 14 that the number of sub-calls is bounded
     $$\prod_{i \leq n}(|v_i| + 1) \leq (d \times (\!| \mathtt{f}(t_1, \cdots, t_n) |\!))^d$$
     where d is the maximal arity of a function symbol.
  2. The status of $\mathtt{g}$ is lexicographic. But by definition of $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-programs, there is at most one recursive call starting from $\mathtt{g}$ for each rule application. Lemma 33(3) entails that the maximal length of a branch is $(\!| \mathtt{f}(t_1, \cdots, t_n) |\!)^d$ which is also a bound on the number of successive calls initiated by $\mathtt{f}$.

From both previous points, we obtain that
$$\#C^{\wedge}_{\mathtt{g}} \leq (\#C^{\vee}_{\mathtt{g}} + 1) \times d^d \times (\!| \mathtt{f}(t_1, \cdots, t_n) |\!)^d$$
Finally, we have
$$\#C_{\mathtt{g}} \leq \#C^{\vee}_{\mathtt{g}} + \#C^{\wedge}_{\mathtt{g}}$$
By combining previous inequalities, we see that the cardinality of C is polynomially bounded in $(\!| \mathtt{f}(t_1, \cdots, t_n) |\!)$. ☐ ☐

Example 45. Figures 5 and 6 show the covering graphs of $\mathtt{lcs}$ and $\mathtt{max}$ in the $\langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{baaba}\rangle$-call tree. Nodes in $C^{\vee}_{\mathtt{g}}$ have been squared while nodes of $C^{\wedge}_{\mathtt{g}}$ have been circled ($\mathtt{g} \in \{\mathtt{lcs}, \mathtt{max}\}$).

Lemma 46. *Let f be a $RPO^{QI}_{Pro+Lin}$-program. For each constructor term $t_1, \cdots, t_n$, the runtime of the call by value interpreter with cache to compute $f(t_1, \cdots, t_n)$ is bounded by a polynomial in $(\!| f(t_1, \cdots, t_n) |\!)$.*

$$\begin{tikzcd}
 & & & & \langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{baaba}\rangle \arrow[dll] \arrow[drr] \arrow[drrrrr] & & & & & \\
 & & \langle \mathtt{lcs}, \mathbf{baba}, \mathbf{baaba}\rangle \arrow[d] & & & & \langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{aaba}\rangle \arrow[d] & & & \langle \mathtt{max}, 4, 4\rangle \arrow[d] \\
 & & \langle \mathtt{lcs}, \mathbf{aba}, \mathbf{aaba}\rangle \arrow[d] & & & & \langle \mathtt{lcs}, \mathbf{baba}, \mathbf{aba}\rangle \arrow[dl] \arrow[dr] \arrow[drr] & & & \langle \mathtt{max}, 3, 3\rangle \arrow[d] \\
 & & \langle \mathtt{lcs}, \mathbf{ba}, \mathbf{aba}\rangle \arrow[dll] \arrow[dl] \arrow[dr] & & & \langle \mathtt{lcs}, \mathbf{aba}, \mathbf{aba}\rangle \arrow[dll] & & \langle \mathtt{lcs}, \mathbf{baba}, \mathbf{ba}\rangle \arrow[d] & \langle \mathtt{max}, 3, 2\rangle \arrow[d] & \langle \mathtt{max}, 2, 2\rangle \arrow[d] \\
\langle \mathtt{max}, 1, 2\rangle \arrow[d] & \langle \mathtt{lcs}, \mathbf{a}, \mathbf{aba}\rangle \arrow[d] & & \langle \mathtt{lcs}, \mathbf{ba}, \mathbf{ba}\rangle \arrow[d] & & & & \langle \mathtt{lcs}, \mathbf{aba}, \mathbf{a}\rangle \arrow[d] & \langle \mathtt{max}, 2, 1\rangle \arrow[d] & \langle \mathtt{max}, 1, 1\rangle \arrow[d] \\
\langle \mathtt{max}, 0, 1\rangle & \langle \mathtt{lcs}, \epsilon, \mathbf{ba}\rangle & & \langle \mathtt{lcs}, \mathbf{a}, \mathbf{a}\rangle \arrow[d] & & & & \langle \mathtt{lcs}, \mathbf{ba}, \epsilon\rangle & \langle \mathtt{max}, 1, 0\rangle & \langle \mathtt{max}, 0, 0\rangle \\
 & & & \langle \mathtt{lcs}, \epsilon, \epsilon\rangle & & & & & &
\end{tikzcd}$$

Figure 3: The $\langle \mathtt{lcs}, \mathbf{ababa}, \mathbf{baaba}\rangle$-call tree with memoization.

(Constructor)

$$\frac{\mathbf{c} \in \mathcal{C} \quad \langle C_{i-1}, t_i \rangle \Downarrow \langle C_i, w_i \rangle}{\langle C_0, \mathbf{c}(t_1, \cdots, t_n) \rangle \Downarrow \langle C_n, \mathbf{c}(w_1, \cdots, w_n) \rangle}$$

(Read)

$$\frac{\langle C_{i-1}, t_i \rangle \Downarrow \langle C_i, w_i \rangle \quad (\mathtt{f}, w_1, \cdots, w_n, w) \in C_n}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n) \rangle \Downarrow \langle C_n, w \rangle}$$

(Update)

$$\frac{\langle C_{i-1}, t_i \rangle \Downarrow \langle C_i, w_i \rangle \quad \mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = w_i \quad \langle C_n, r\sigma \rangle \Downarrow \langle C, w \rangle}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n) \rangle \Downarrow \langle C \cup (\mathtt{f}, w_1, \cdots, w_n, w), w \rangle}$$

Figure 4: Call-by-value interpreter with Cache of $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$.

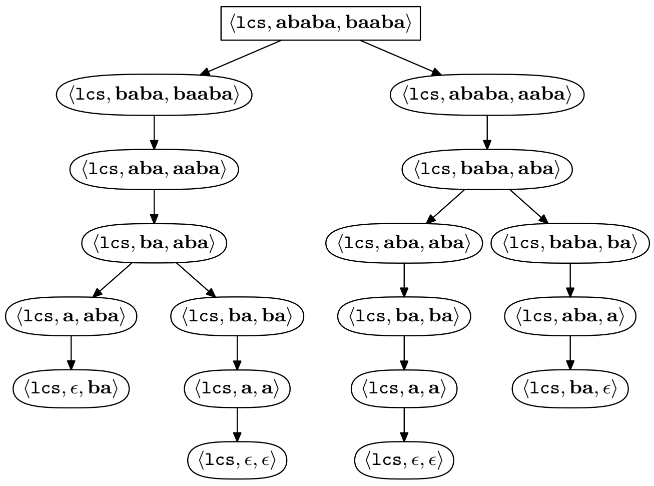


Figure 5: The `lcs`-covering graph of the call-tree.

Proof. Since an evaluation procedure memorizes all necessary configurations, the runtime is at most quadratic in the size of the cache. Note that the exact runtime depends on the implementation strategy and in particular on the cache management. □ □

Theorem 47. *Let f be an additive $RPO^{QI}_{Pro+Lin}$-program (resp. RPO^{QI}_{Pro}-program and RPO^{QI}_{Lin}-program). For each constructor term $t_1, \cdots, t_n$, the runtime to compute $f(t_1, \cdots, t_n)$ is bounded by a polynomial in $\max_{i=1}^n |t_i|$.*

Proof. By Proposition 12, we have $(\!| t_i |\!) \leq O(|t_i|)$. For some polynomial P we have $(\!| \mathtt{f}(t_1, \cdots, t_n) |\!) \leq P(\max_{i=1}^n |t_i|)$, because quasi-interpretations are polynomially bounded. Lemma 46 implies that the time is bounded by a polynomial in $\max_{i=1}^n |t_i|$. □ □

In the general case, memoization is not used because one cannot decide which results will be reused and the cache

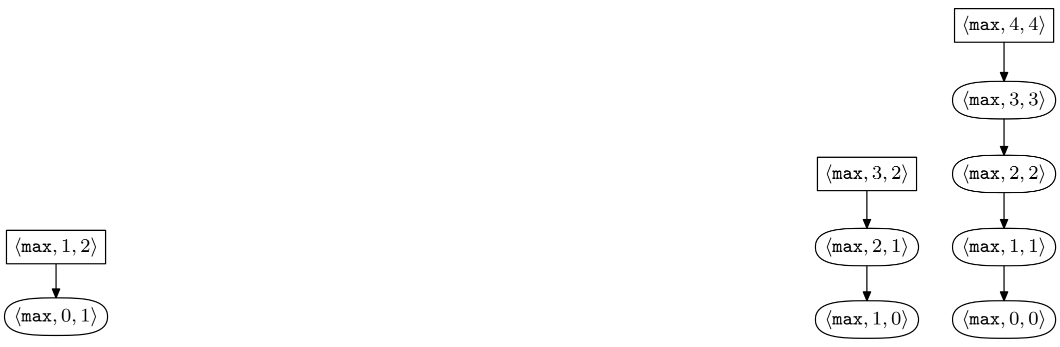


Figure 6: The `max`-covering graph of the call-tree.

may become too big to be really useful. In our particular case, the termination ordering gives enough information on the structure of the program to minimize the cache [Mar00].

We see that if a function symbol is linear, see Definition 40, then no result needs to be recorded. More generally, consider the $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$-call tree. When evaluating $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$, the call by value semantics with cache stores all values. Say that a separation set N is a set of states such that each chain starting from the root state $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$ meets a state of N. If we know the value of each state of N, then values of the states below N-states are useless in order to determine $\langle \mathtt{f}, t_1, \cdots, t_n\rangle$. And, we can forget them. Therefore, it is sufficient to store in a cache a separation set N for each function symbol. Now say that a separation set N is minimal if for each state $s \in N$, $N \setminus \{s\}$ is not a separation set. We can require an implementation to keep a minimal separation set. To perform dynamically, we have to compare configurations in the cache. Take two configurations $(\mathtt{f}, t_1, \cdots, t_n, t)$ and $(\mathtt{g}, s_1, \cdots, s_m, s)$. If $\mathtt{f}(t_1, \cdots, t_n) \prec_{rpo} \mathtt{g}(s_1, \cdots, s_m)$ then we do not need anymore the configuration $(\mathtt{f}, t_1, \cdots, t_n, t)$ and we can erase it from the cache.

7.3 Beyond polynomial time

Theorem 48.

- *The set of functions computed by* affine $RPO^{QI}_{Pro+Lin}$*-programs (resp.* RPO^{QI}_{Pro}*-programs and* RPO^{QI}_{Lin}*-programs) is exactly the set of functions computable in* linear exponential time*, that is in time bounded by* $2^{O(n)}$.
- *The set of functions computed by* multiplicative $RPO^{QI}_{Pro+Lin}$*-programs (resp.* RPO^{QI}_{Pro}*-programs and* RPO^{QI}_{Lin}*-programs) is exactly the set of functions computable in* linear double exponential time*, that is in time bounded by* $2^{2^{O(n)}}$.

Proofs are very similar to the one of Theorem 47. The kind of quasi-interpretation gives the different upper-bounds on the time-usage as established in Proposition 12. The converse is again a consequence of Theorems 57 and 60.

8 Simulation of Parallel Register Machines

8.1 Parallel Register Machines

Following [LM95], we introduce *Parallel Register Machines (PRM)* which are able to model the essential features of both traditional sequential computing like Turing Machines and alternating computations like Alternating Turing Machines.

A PRM M works over the word algebra $\mathbb{W}$ generated by the constructors $\{\mathbf{0}, \underline{\ }, \boldsymbol{\epsilon}\}$ and consists in

1. a finite set $S = \{s_0, s_1, \ldots, s_k\}$ of *states*, including a distinct state BEGIN.
2. a finite list $\Pi = \{\pi_1, \ldots, \pi_m\}$ of *registers*; we write OUTPUT for π_m; Registers will only store values in $\mathbb{W}$;
3. a function *com* mapping states to *commands* which are
   $[\mathbf{Succ}(\pi = \mathbf{i}(\pi), s')]$, $[\mathbf{Pred}(\pi = \mathbf{p}(\pi), s')]$, $[\mathbf{Branch}(\pi, s', s'')]$,
   $[\mathbf{Fork}_{\min}(s', s'')]$, $[\mathbf{Fork}_{\max}(s', s'')]$, $[\mathbf{End}]$.

A *configuration* of a PRM M is given by a pair (s, F) where $s \in S$ and F is a function $\Pi \to \mathbb{W}$ which stores register values. We note $\{\pi \leftarrow \pi'\}F$ to mean that the value of the register π is the content of π', the other registers stay unchanged.

In order to have a choice mechanism to simulate alternation by the fork operation, we define an ordering $\blacktriangleleft$ on $\mathbb{W}$: $\boldsymbol{\epsilon} \blacktriangleleft y$, $\mathbf{0}(x) \blacktriangleleft \underline{\ }(y)$, $\mathbf{i}(x) \blacktriangleleft \mathbf{i}(y)$ if and only if $x \blacktriangleleft y$. We define the operations $\min_\blacktriangleleft$ and $\max_\blacktriangleleft$ wrt $\blacktriangleleft$.

Next, we define a semantic partial-function $\mathtt{eval} : \mathbb{N} \times S \times \mathbb{W}^m \mapsto \mathbb{W}$, that maps the result of the machine in a time bound given by the first argument.

- $\mathtt{eval}(0, s, F)$ is undefined.
- If $com(s)$ is $\mathbf{Succ}(\pi = \mathbf{i}(\pi), s')$ then $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, s', \{\pi \leftarrow \mathbf{i}(\pi)\}F)$.
- If $com(s)$ is $\mathbf{Pred}(\pi = \mathbf{p}(\pi), s')]$, then $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, s', \{\pi \leftarrow \mathbf{p}(\pi)\}F)$ where $\mathbf{p}$ is the predecessor function on $\mathbb{W}$;
- If $com(s)$ is $\mathbf{Branch}(\pi, s', s'')$ then $\mathtt{eval}(t+1, s, F) = \mathtt{eval}(t, r, F)$, where $r = s'$ if $\pi = \mathbf{0}(w)$ and $r = s''$ if $\pi = \underline{\ }(w)$;
- If $com(s)$ is $\mathbf{Fork}_{\min}(s', s'')$ then
  $\mathtt{eval}(t+1, s, F) = \min_\blacktriangleleft(\mathtt{eval}(t, s', F), \mathtt{eval}(t, s'', F))$;

- If $com(s)$ is $\mathbf{Fork}_{\max}(s', s'')$ then
  $\mathtt{eval}(t+1, s, F) = \max_{\blacktriangleleft}(\mathtt{eval}(t, s', F), \mathtt{eval}(t, s'', F))$;
- If $com(s)$ is **End** then $\mathtt{eval}(t+1, s, F) = F(\text{OUTPUT})$.

Let $T : \mathbb{N} \to \mathbb{N}$ be a function. A function $\phi : \mathbb{W}^k \to \mathbb{W}$ is PRM-computable in time T if there is a PRM M such that for each $(w_1, \cdots, w_k) \in \mathbb{W}^k$, we have

$$\mathtt{eval}(T(\max_{i=1}^{k} |w_i|), \text{BEGIN}, F_0) = \phi(w_1, \cdots, w_k)$$

where $F_0(\pi_i) = w_i$ for $i = 1..k$ and otherwise $F_0(\pi_j) = \boldsymbol{\epsilon}$.

8.2 Space and Time bounded computation

A *Register machines (RM)* is a PRM without fork commands. A Turing machine can be simulated linearly in time by a RM.

Proposition 49. *A function ϕ is computable in polynomial (respectively exponential, doubly exponential) time iff ϕ is RM-computable in polynomial time (resp. exponential, doubly exponential).*

There are pleasingly well-known tight connections between space used by a Turing machine and time used by PRM. The essence of the translation comes from the work of Savitch [Sav70] and Chandra, Kozen, Stockmeyer [CKS81].

Theorem 50. *A function ϕ is computable in polynomial (resp. exponential, doubly exponential) space iff ϕ is PRM-computable in polynomial time (resp. exponential, doubly exponential).*

8.3 Time bounded simulation with lexicographic termination

Without loss of generality, we consider only unary functions in the following. It would be laborious to specify this simulation in full details otherwise.

Lemma 51 (Lexicographic Plug and play lemma)**.** *Assume that $\phi : \mathbb{W} \to \mathbb{W}$ is a PRM-computable function in time bounded by T. Define f by*

$$\begin{array}{rrcll} f: & \mathbb{N} \times \mathbb{W} & \to & \mathbb{W} & \\ & (n, w) & \mapsto & \phi(w) & \textit{if } n > T(|w|) \\ & (n, w) & \mapsto & \perp & \textit{otherwise} \end{array}$$

Then,

1. *the function f is computed by an additive RPO^{QI}-program,*
2. *and, if f is computed by a RM, then f is computable by an additive RPO$^{QI}_{Lin}$-program.*

Proof. Suppose that f is computed by a PRM M. The simulation of the PRM M is done by following the rules of the semantic partial function $\mathtt{eval}$. For this, the set of constructors is $\mathcal{C} = \{\mathbf{0}, \underline{\,}, \mathbf{s}, \diamond, \boldsymbol{\epsilon}\} \cup S$ where S is the set of states.

We show first that $\min_{\blacktriangleleft}$ and $\max_{\blacktriangleleft}$ are additive RPO^{QI}-program.

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathtt{min}(\boldsymbol{\epsilon}, w) & \to & \boldsymbol{\epsilon} & \mathtt{max}(\boldsymbol{\epsilon}, w) & \to & w \\ \mathtt{min}(w, \boldsymbol{\epsilon}) & \to & \boldsymbol{\epsilon} & \mathtt{max}(w, \boldsymbol{\epsilon}) & \to & w \\ \mathtt{min}(\mathbf{0}(w), \underline{\,}(w')) & \to & \mathbf{0}(w) & \mathtt{max}(\mathbf{0}(w), \underline{\,}(w')) & \to & \underline{\,}(w') \\ \mathtt{min}(\underline{\,}(w), \mathbf{0}(w')) & \to & \mathbf{0}(w') & \mathtt{max}(\underline{\,}(w), \mathbf{0}(w')) & \to & \underline{\,}(w) \\ \mathtt{min}(\bar{\mathbf{i}}(w), \mathbf{i}(w')) & \to & \mathbf{i}(\mathtt{min}(w, w')) & \mathtt{max}(\bar{\mathbf{i}}(w), \mathbf{i}(w')) & \to & \bar{\mathbf{i}}(\mathtt{max}(w, w')) \end{array}$$

with $\mathbf{i} \in \{\mathbf{0}, \underline{\,}\}$.

We associate the following quasi-interpretations:

$$\begin{array}{ccc} (\!|\boldsymbol{\epsilon}|\!) = 0 & (\!|\diamond|\!) = 0 & \forall q \in S, (\!|q|\!) = 0 \\ (\!|\mathbf{0}|\!)(X) = X + 1 & (\!|\underline{\,}|\!)(X) = X + 1 & (\!|\mathbf{s}|\!)(X) = X + 1 \\ (\!|\mathtt{min}|\!)(W, W') = \max(W, W') & (\!|\mathtt{max}|\!)(W, W') = \max(W, W') & \end{array}$$

Next, we write a program to compute the semantic partial function $\mathtt{eval}$.

(a) $\mathtt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \pi_m) \to \mathtt{Eval}(t, s', \pi_1 \cdots, \mathbf{i}(\pi_j), \cdots, \pi_m)$
if $com(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_j), s')$,

(b) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \mathbf{i}(\pi_j), \cdots, \pi_m) \rightarrow \texttt{Eval}(t, s', \pi_1, \cdots, \pi_j, \cdots, \pi_m)$
 if $\mathit{com}(s) = \mathbf{Pred}(\pi_j = \mathbf{p}(\pi_j), s')$,

(c) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \mathbf{0}(\pi_j), \cdots, \pi_m) \rightarrow \texttt{Eval}(t, s', \pi_1, \cdots, \pi_m)$
 if $\mathit{com}(s) = \mathbf{Branch}(\pi_j, s', s'')$,

(d) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \underline{\ }(\pi_j), \cdots, \pi_m) \rightarrow \texttt{Eval}(t, s'', \pi_1, \cdots, \pi_m)$
 if $\mathit{com}(s) = \mathbf{Branch}(\pi_j, s', s'')$,

(e) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \pi_m) \rightarrow \texttt{min}(\texttt{Eval}(t, s', \pi_1, \cdots, \pi_m), \texttt{Eval}(t, s'', \pi_1, \cdots, \pi_m))$
 if $\mathit{com}(s) = \mathbf{Fork}_{\min}(s', s'')$,

(f) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \cdots, \pi_m) \rightarrow \texttt{max}(\texttt{Eval}(t, s', \pi_1, \cdots, \pi_m), \texttt{Eval}(t, s'', \pi_1, \cdots, \pi_m))$
 if $\mathit{com}(s) = \mathbf{Fork}_{\max}(s', s'')$,

(g) $\texttt{Eval}(\mathbf{s}(t), s, \pi_1, \ldots, \pi_m) \rightarrow \pi_m$, if $\mathit{com}(s) = \mathbf{End}$,

Finally, put $\texttt{f}(t, w) \rightarrow \texttt{Eval}(t, \text{BEGIN}, w, \boldsymbol{\epsilon}, \ldots, \boldsymbol{\epsilon})$. It is routine to check that $f = [\![\texttt{f}]\!]$.
These programs admit the following quasi-interpretations:

$$(\!|\texttt{Eval}|\!)(T, S, \Pi_1, \cdots, \Pi_m) = T + S + \sum_{i=1}^{m} \Pi_i$$
$$(\!|\texttt{f}|\!)(T, X) = T + X$$

The status of each function symbol is lexicographic. The precedence satisfies $\{\texttt{min}, \texttt{max}\} \prec_{\mathcal{F}} \texttt{Eval} \prec_{\mathcal{F}} \texttt{f}$. We see that each rule is decreasing by $\prec_{rpo}$. Therefore, $\texttt{f}$ is a RPO^{QI}-program.

Now, observe that $\texttt{Eval}$ has always one occurrence in the right hand side of the rules except in the fork cases. So, $\texttt{f}$ is a RPO$^{\text{QI}}_{\text{Lin}}$-program if f is computed by a RM. ☐ ☐

8.4 Time bounded simulation with product termination

In [MM00], the simulation of RM is performed by a RPO$^{\text{QI}}_{\text{Pro}}$-program in a different manner because the status of function symbols is product and not lexicographic as in the above result. For this reason, we give details of the simulation of a time bounded function in the case where symbols have a product status.

Lemma 52 (Product Plug and play lemma). *Assume that $\phi : \mathbb{W} \rightarrow \mathbb{W}$ is a RM-computable function in time bounded by T. Define f by*

$$\begin{array}{rlcll} f: & \mathbb{N} \times \mathbb{W} & \rightarrow & \mathbb{W} & \\ & (n, w) & \mapsto & \phi(w) & \textit{if } n > T(|w|) \\ & (n, w) & \mapsto & \bot & \textit{otherwise} \end{array}$$

Then, f is computable by an additive RPO$^{QI}_{Pro}$-program.

Proof. Compared with the previous proof, the simulation is performed in bottom-up way. For this, we use an extra constructor $\mathbf{c}$ to encode tuples. And we define $\texttt{Step}$ which gives the next configuration.

(a) $\texttt{Step}(\mathbf{c}(s, \pi_1, \cdots, \pi_m)) \rightarrow \mathbf{c}(s', \pi_1 \cdots, \mathbf{i}(\pi_j), \cdots, \pi_m)$
 if $\mathit{com}(s) = \mathbf{Succ}(\pi_j = \mathbf{i}(\pi_j), s')$

(b) $\texttt{Step}(\mathbf{c}(s, \pi_1, \cdots, \mathbf{i}(\pi_j), \cdots, \pi_m)) \rightarrow \mathbf{c}(s', \pi_1, \cdots, \pi_j, \cdots, \pi_m)$
 if $\mathit{com}(s) = \mathbf{Pred}(\pi_j = \mathbf{p}(\pi_j), s')$

(c) $\texttt{Step}(\mathbf{c}(s, \pi_1, \cdots, \mathbf{0}(\pi_j), \cdots, \pi_m) \rightarrow \mathbf{c}(s', \pi_1, \cdots, \pi_m)$
 if $\mathit{com}(s) = \mathbf{Branch}(\pi_j, s', s'')$

(d) $\texttt{Step}(\mathbf{c}(s, \pi_1, \cdots, \underline{\ }(\pi_j), \cdots, \pi_m) \rightarrow \mathbf{c}(s'', \pi_1, \cdots, \pi_m)$
 if $\mathit{com}(s) = \mathbf{Branch}(\pi_j, s', s'')$

(e) $\texttt{Step}(\mathbf{c}(s, \pi_1, \ldots, \pi_m) \rightarrow \mathbf{c}(s, \pi_1, \ldots, \pi_m)$
 if $\mathit{com}(s) = \mathbf{End}$

The simulation is made by

$$\begin{aligned}\mathtt{Eval}(\boldsymbol{\epsilon}, x) &\to x\\ \mathtt{Eval}(\mathtt{s}(t), x) &\to \mathtt{Step}(\mathtt{Eval}(t, x))\\ \mathtt{f}(t, w) &= \mathtt{Eval}(t, \mathbf{c}(\text{BEGIN}, w, \boldsymbol{\epsilon}, \ldots, \boldsymbol{\epsilon}))\end{aligned}$$

The rules are ordered by putting $\mathtt{Step} \prec_{\mathcal{F}} \mathtt{Eval}$ where each symbol has now a product status. A quasi-interpretation of the rules is

$$\begin{aligned}(\!|\mathbf{c}|\!)(S, \Pi_1, \cdots, \Pi_m) &= S + \sum_i \Pi_i + 1\\ (\!|\mathtt{Step}|\!)(X) &= X + 1\\ (\!|\mathtt{Eval}|\!)(T, X) &= T + X\end{aligned}$$

The others constructor assignments are identical to the ones in the previous simulation described in the proof of Lemma 51

$$\begin{array}{ccc}(\!|\boldsymbol{\epsilon}|\!) = 0 & (\!|\diamond|\!) = 0 & \forall q \in S, (\!|q|\!) = 0\\ (\!|\mathbf{0}|\!)(X) = X + 1 & (\!|\mathbf{1}|\!)(X) = X + 1 & (\!|\mathbf{s}|\!)(X) = X + 1\end{array}$$

□ □

Unlike the previous proof, this simulation can not be extended in order to capture parallel computation.

Now, it remains to compute the clock, that is the length of the iteration of the main loop of the simulation in order to complete the simulation.

8.5 Simulation of Polynomial computations

Proposition 53. *Any polynomial is computed by an additive RPO^{QI}_{Lin}-programs (resp. RPO^{QI}_{Pro}-programs).*

Proof. We define any polynomial by composition from the additive programs for the addition add and the multiplication *mult*.

$$\begin{aligned}\mathtt{add}(\diamond, y) &\to y\\ \mathtt{add}(\mathtt{s}(x), y) &\to \mathtt{s}(\mathtt{add}(x, y))\\ \mathtt{mult}(\diamond, y) &\to \diamond\\ \mathtt{mult}(\mathtt{s}(x), y) &\to \mathtt{add}(y, \mathtt{mult}(x, y))\end{aligned}$$

The programs add and mult admits the following quasi-interpretations

$$\begin{aligned}(\!|\mathtt{add}|\!)(X, Y) &= X + Y\\ (\!|\mathtt{mult}|\!)(X, Y) &= X \times Y\end{aligned}$$

where constructors have the quasi-interpretation

$$\begin{aligned}(\!|\diamond|\!) &= 0\\ (\!|\mathbf{s}|\!)(X) &= X + 1\end{aligned}$$

The programs add and mult terminates by $\prec_{rpo}$ by putting $\mathtt{add} \prec_{\mathcal{F}} \mathtt{mult}$ with a product status. So, any polynomial is a $\text{RPO}^{\text{QI}}_{\text{Pro}}$-program. On the other hand, add and mult are linear and thus any polynomial is a also $\text{RPO}^{\text{QI}}_{\text{Lin}}$-program. □ □

Theorem 54.

- *A polynomial time function is computed by an additive RPO^{QI}_{Lin}-program.*
- *A polynomial time function is computed by an additive RPO^{QI}_{Pro}-program.*
- *A polynomial time function is computed by an additive $RPO^{QI}_{Pro+Lin}$-program.*

Proof. Let ϕ be a function which is computed by a RM in time bounded by a polynomial P.

For the first case, the time bound P is computed by an additive $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program following Proposition 53. We compose with Lemma 51, we conclude that ϕ is computed by an additive $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program.

For the second case, the time bound P is also an additive $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program following Proposition 53. We compose with Lemma 52, we conclude that ϕ is computed by an additive $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program.

The last case is an immediate consequence of the previous constructions. □ □

Theorem 55. *A polynomial space function is computed by an additive RPO^{QI}-program.*

Proof. Let ϕ be a function which is computed by a PRM in time bounded by a polynomial P. Since P is also computed by a RPO^{QI}-program following Proposition 53 and by composing with Lemma 51, we conclude that ϕ is computed by an additive RPO^{QI}-program. □ □

8.6 Simulation of exponential computations

Proposition 56. *Let $\gamma > 0$ be a constant. The function $\lambda n.2^{\gamma n}$ is computed by an affine RPO^{QI}_{Lin}-program (resp. RPO^{QI}_{Pro}-program).*

Proof. The function $\lambda n.2^{\gamma n}$ is computed by

$$\begin{aligned}
\mathtt{mk}_\gamma(\diamond) &\to \diamond \\
\mathtt{mk}_\gamma(\mathtt{s}'(x)) &\to \tilde{\mathtt{s}}'(\ldots(\tilde{\mathtt{s}}'(\mathtt{mk}_\gamma(x)))\ldots) && \gamma \text{ times} \\
\mathtt{d}(x) &\to \mathtt{add}(x,x) && \mathtt{add} \text{ is defined in Prop 53} \\
\mathtt{exp}(\diamond) &\to \mathtt{s}(\diamond) \\
\mathtt{exp}(\tilde{\mathtt{s}}'(x)) &\to \mathtt{d}(\mathtt{exp}(x)) \\
\mathtt{exp}_\gamma(x) &\to \mathtt{exp}(\mathtt{mk}_\gamma(x))
\end{aligned}$$

We have $\llbracket \mathtt{exp}_\gamma \rrbracket (n) = m$ where $n = (\mathtt{s}')^n(\diamond)$ and $m = (\mathtt{s})^{2^{\gamma n}}(\diamond)$.

Constructors have the following quasi-interpretation

$$\begin{aligned}
(\!|\diamond|\!) &= 0 \\
(\!|\mathtt{s}|\!)(X) &= X + 1 \\
(\!|\mathtt{s}'|\!)(X) &= 2^\gamma X + 2^\gamma - 1 \\
(\!|\tilde{\mathtt{s}}|\!)(X) &= 2X + 1
\end{aligned}$$

And this program admits the following quasi-interpretations

$$\begin{aligned}
(\!|\mathtt{mk}_k|\!)(X) &= X \\
(\!|\mathtt{d}|\!)(X) &= 2X \\
(\!|\mathtt{exp}|\!)(X) &= X + 1 \\
(\!|\mathtt{exp}_\gamma|\!)(X) &= X + 1
\end{aligned}$$

This program terminates by $\prec_{rpo}$ with a product status. So, $\lambda n.2^{\gamma n}$ is a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program and also a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program. □ □

Theorem 57. *A exponential time function is computed by an affine RPO^{QI}_{Lin}-program (resp. RPO^{QI}_{Pro}-program or $RPO^{QI}_{Pro+Lin}$-program).*

Theorem 58. *An exponential space function is computed by an affine RPO^{QI}-program.*

8.7 Simulation of doubly exponential computations

Proposition 59. *Let $\gamma > 0$ be a constant. The function $\lambda n.2^{2^{\gamma n}}$ is computed by an multiplicative RPO^{QI}_{Lin}-program (resp. RPO^{QI}_{Pro}-program).*

Proof. The function $\lambda n.2^{2^{\gamma n}}$ is computed by

$$\begin{aligned}
\mathtt{dmk}_\gamma(\diamond) &\to \diamond \\
\mathtt{dmk}_\gamma(\mathtt{s}''(x)) &\to \tilde{\mathtt{s}}''(\ldots(\tilde{\mathtt{s}}''(\mathtt{dmk}_\gamma(x)))\ldots) && \gamma \text{ times} \\
\mathtt{square}(x) &\to \mathtt{mult}(x,x) && \mathtt{mult} \text{ is defined in Prop 53} \\
\mathtt{dexp}(\diamond) &\to \mathtt{s}(\mathtt{s}(\diamond)) \\
\mathtt{dexp}(\tilde{\mathtt{s}}''(x)) &\to \mathtt{square}(\mathtt{dexp}(x)) \\
\mathtt{dexp}_\gamma(x) &\to \mathtt{dexp}(\mathtt{dmk}_\gamma(x))
\end{aligned}$$

We have $[\![\mathtt{dexp}_\gamma]\!]\,(n) = m$ where $n = (\mathtt{s}'')^n(\diamond)$ and $m = (\mathtt{s})^{2^{2^{\gamma n}}}(\diamond)$.

Constructors have the following quasi-interpretation

$$\begin{aligned}
(\!|\diamond|\!) &= 0 \\
(\!|\mathtt{s}|\!)(X) &= X+1 \\
(\!|\mathtt{s}''|\!)(X) &= \theta_\gamma(X) \\
(\!|\tilde{\mathtt{s}}''|\!)(X) &= (X+2)^2
\end{aligned}$$

where

$$\begin{aligned}
\theta_0(X) &= X \\
\theta_{k+1}(X) &= (\theta_k(X)+2)^2 && 0 \le k < \gamma
\end{aligned}$$

And the program admits the following quasi-interpretations

$$\begin{aligned}
(\!|\mathtt{dmk}_\gamma|\!)(X) &= X \\
(\!|\mathtt{square}|\!)(X) &= X^2 \\
(\!|\mathtt{dexp}|\!)(X) &= X+2 \\
(\!|\mathtt{dexp}_\gamma|\!)(X) &= X+2
\end{aligned}$$

This program terminates by $\prec_{rpo}$ with a product status. So, $\lambda n.2^{2^{\gamma n}}$ is a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Pro}}$-program and also a $\mathrm{RPO}^{\mathrm{QI}}_{\mathrm{Lin}}$-program. □ □

Theorem 60. *A doubly exponential time function is computed by a multiplicative RPO^{QI}_{Lin}-program (resp. RPO^{QI}_{Pro}-program or $RPO^{QI}_{Pro+Lin}$-program).*

Theorem 61. *A doubly exponential space function is computed by a multiplicative RPO^{QI}-program.*

Resource Control Graphs

Jean-Yves Moyen

ACM Transactions on Computational Logic 10(4), 2009

Abstract:

Resource Control Graphs are an abstract representation 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.
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, i.e. of programs that can compute without allocating new memory.
This new tool is able to encompass several existing analyses and similarities with other studies suggest that even more analyses might be expressable in this framework, thus giving hopes for a generic tool for studying programs.

1 Introduction

1.1 Motivations

The goal of this study is to predict usage and control computational resources, like space or time, that are used during the execution of a program. For this, we introduce a new tool called *Resource Control Graphs* and focus here on explaining how it can be used for termination proofs and space complexity management.

We present a data flow analysis of a low-level language by means of Resource Control Graph, and we try to illustrate that this is a generic concept from which several program properties could be checked.

Usual data flow analyses (see [NNH99] for a detailed overview) use transfer functions to express how a given property is modified when following the program's execution. Then, a fixed point algorithm finds for each label a set of all possible values for the property. For example, one might be interested in which sign a given variable can take at each point. The instructions of the program give constraints on this (from one label to the next one). Iterating these constraints with a fixed point algorithm can find the set of all possible signs for the variable at each label.

Here, we want to consider each execution separately. So, when iterating the transfer function and coming back to an already treated label, instead of unifying the new constraint with the old one and iterating towards a fixed point, we consider this as a new configuration. In the end, instead of obtaining one set associated to each label, we get a set of so called "walks", each associating one value to each occurrence of each label. For example, a first walk can tell that if starting with a positive value at a given label, the variable remains positive, but another walk tells that if starting with a negative value, the variable may become positive. In this case, the fixed point algorithm builds the set $\{+, -\}$ for each label.

Of course, we then need a way to study this set of walks and find common properties on them that yield some information about the program.

The first problem we consider is the one of detecting programs able to compute within a constant amount of space, that is without performing dynamic memory allocation. These were dubbed *Non Size Increasing* by [Hof99].

There are several approaches that try to solve this problem. The first protection mechanism is by monitoring computations. However, if the monitor is compiled with the program, it could itself cause memory leak or other problems. The second is the testing-based approach, which is complementary to static analysis. Indeed, testing provides a lower bound on the memory usage while static analysis gives an upper bound. The gap between both bounds is of some value in practice. Lastly, the third approach is type checking done by a bytecode verifier. In an untrusted environment (like embedded systems), the type protection policy (Java or .Net) does not allow dynamic allocation. Actually, the former approach relies on a high-level language that captures and deals with memory allocation features [AC03]. Our approach guarantees, and even provides, a proof certificate of upper bound on space computation on a low-level language without disallowing dynamic memory allocations.

The second problem that we study is termination of programs. This is done by adapting ideas of [LJBA01], [BA08] and [AA02]. The intuition is that a program terminates whenever there is no more resource to consume.

There are also long term theoretical motivations for our work. Indeed a lot of work has been done in the last twenty years to provide syntactic characterisations of complexity classes, *e.g.* by [BC92] or [LM93a]. Those characterisations are the foundation of recent research on describing broad classes of programs that run within some specified amount of time or space. Examples include works by [Hof99] as well as [NW06], [KJ09], [ACGZJ04] and [BMM11].

We believe that our Resource Control Graphs is able to encompass several of these analyses and express them in a common framework. In this sense, Resource Control Graphs are an attempt to build a generic tool for program analysis.

1.2 Coping with undecidability

All these theoretical frameworks share the common particularity of dealing with behaviours of programs (like time and space complexity) and not only with the inputs/outputs relation, which only depends on the computed function.

Following [Jon97], we call a *program property* a subset of all programs (for a given, yet unspecified, language) and we say that a property A is *extensional* if for all programs p, q computing the same function, we have $p \in A \Leftrightarrow q \in A$. That is, an extensional property is shared by all programs computing the same function.

On the other hand, properties not shared by programs computing the same function are called *intensional.* A typical extensional property is termination. Indeed, all programs computing the same function must terminate on the same inputs (where the function is defined). A typical intensional property is time complexity. Indeed, two programs with different complexities can compute the same function; for example, insertion sort computes the sorting function in time $O(n^2)$ while merge sort computes it in time $O(n \log(n))$.

Classical complexity theory focuses on functions or problems and extensional properties. It defines complexity classes (such as LogSpace, Ptime) as classes of *problems* and not classes of algorithms and studies complexity of problems (*e.g.* SAT, QBF, ...) and relationship between classes of problems ("Is P different from NP ?", ...) Here, we want to consider *intensional* complexity, that is try to understand why a given algorithm is more efficient than another one to compute the same function, and we want to study classes of *algorithms* rather than classes of problems or functions.

When studying the complexity of functions, one deals with extensional properties; however, the complexity of algorithms is an intensional property. Consider for example the class $\mathcal{C}$ of *functions* computable in time $O(n\log(n))$. This is a class of functions, hence an extensional property. Consider now the corresponding program property: A is the set of all programs computing a function from $\mathcal{C}$. Obviously, the merge sort algorithm is in A. But the insertion sort algorithm, with complexity $O(n^2)$ is *also* in A. Indeed, the function computed by the insertion sort, the sorting function, is in $\mathcal{C}$ because there exists a $O(n\log(n))$ program computing it.

On the other hand, the complexity of algorithms is an intensional property. Consider the program property B, the set of all programs whose complexity is $O(n\log(n))$. Now, the merge sort belongs to B because its complexity is $O(n\log(n))$ but the insertion sort does not belong to B because its complexity is $O(n^2)$. We no longer consider only the computed function, but also the algorithm used to compute it. For this reason, "bad" programs are discarded.

A typical goal of intensional study would be to have a criterion to decide whether a given program computes in polynomial time or not.

The study of extensional complexity quickly reaches the boundary of Rice's theorem. Any extensional property of programs is either trivial or undecidable. Intensional properties are even harder to decide as stated by [Asp08].

However, several very successful works do exist for studying both extensional properties (like termination) or intensional ones (like time or space complexity of programs). As these works provide decidable criteria, they must be either incomplete (reject a valid program) or unsound (accept an invalid program). Of course, the choice is usually to ensure soundness: if the program is accepted by the criterion, then the property (termination, polynomial bound,...) is guaranteed. This allows, for instance, the criterion to be seen as a certificate in a *proof carrying code* paradigm [Nec97].

When studying intensional properties (usually complexity of algorithms), two different kinds of approaches exist. The first one consists in restricting the syntax of programs so that any program necessarily satisfies the property. The work on primitive recursive functions, where the recurrence schemata are restricted to only primitive recursion, falls into this category. This approach gives many satisfactory results, such as the characterisations of PTIME by [Cob62] or [BC92], the works of Leivant and Marion on tiering and predicative analysis [LM93a] or the works of Jones on CONS-free programs [Jon00]. On the logical side, this leads to explicit management of resources in Linear Logic [Gir87].

All these characterisations usually have the very nice property of *extensional completeness* in the sense that each *function* in the class (of functions) considered can be computed by an algorithm with the property considered (*e.g.*, a function is in PTIME if and only if it can be defined by bounded primitive recursion (Cobham)). Unfortunately, *intensionality* is not their main concern: these methods usually do not capture natural algorithms, and programmers have to rewrite their programs in a non-natural way. [Col98] show that there exists no primitive recursive algorithm which computes $\min(x, y)$ in time $O(\min(x, y))$ (*i.e.* one has to perform recursion on either x or y).

So, the motto of this first family of methods can be described as leaving the proof burden to the programmer rather than to the analyser. If one can write a program with the given syntax (which, in some cases, can be a real challenge), then certain properties are guaranteed. The other family of methods follow a different direction: let the programmer write whatever he wants, but the analysis is not guaranteed to work.

Since any program can *a priori* be given to the analysis, decidability is generally achieved by using some kind of weak semantics during analysis. That is, one considers *more* than all the executions a program can have. This approach is more recent but has already some very successful results such as the Size Change Termination [LJBA01] or the *mwp*-polynomials of [KJ09].

This second kind of methods can thus be described as not meddling with the programmer and let the whole proof burden lay on the analysis. Of course, as the analysis is incomplete, one usually finds out that certain kinds of programs are not analysed correctly and have to be rewritten. But this restriction is done *a prosteriori* and not *a priori* and it can be tricky to find out what exactly causes the analysis to fail.

Resource Control Graphs are intended to live within this second kind of analysis. Hence, the toy language used as an example is Turing-complete and is not restricted.

1.3 Outline

Section 2 introduces the stack machines, used everywhere in the remainder of this paper, as a simple yet powerful programming language. Section 3 describes the core idea of Resource Control Graphs that can be summed up as finding a decidable (recursive) superset of all the executions that still ensures a given property (such as termination or a complexity bound). Then, Section 4 immediately shows how this can be used in order to detect Non Size Increasing programs. Section 5 presents *Vector Addition Systems with States* that are generalised into *Resource Systems with States* in Section 6. They form the backbone of the *Resource Control Graphs.* Section 7 presents the tool itself and explains how to build a Resource Control Graph for a program and how it can be used to study the program. Section 8 shows application of RCGs in building termination proofs similar to the Size Change Termination principle.

Finally, Section 9 discusses how matrix algebra could be used in program analyses, leading to several possible further developments.

1.4 Notations

In a directed graph[1] $G = (S, A)$, we write $s \xrightarrow{a} s'$ to say that a is an edge between s and s'. Similarly, we write $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ to say that $a_1 \dots a_n$ is a path passing through vertices $s_0, \cdots, s_n$. Or simply $s_0 \xrightarrow{w} s_n$ if $w = a_1 \dots a_n$. The notation $s \to s'$ means that there exists an edge a such that $s \xrightarrow{a} s'$, and $\xrightarrow{+}$, $\xrightarrow{*}$ are the transitive and reflexive-transitive closures of $\to$.

A partial order $\prec$ is a *well partial* order if there is no infinite decreasing sequence and no infinite anti-chain. That is, for every infinite sequence $x_1, \cdots, x_n, \dots$ there are indexes $i < j$ such that $x_i \preceq x_j$. This means that the order is well-founded (no infinite decreasing sequence) but also that there is no infinite sequence of pairwise incomparable elements. The order induced by the divisibility relation on $\mathbb{N}$, for example, is well-founded but is not a well partial order since the sequence of all prime numbers is an infinite sequence of pairwise incomparable elements.

We use $\mathbb{Z}$ (resp. $\mathbb{N}$) to denote the set of integers (resp. the set of integers ≥ 0). We also denote, $\overline{\mathbb{Z}} = \mathbb{Z} \bigcup \{+\infty\}$. When working with vectors of $\mathbb{Z}^k$, $\leq$ denotes the component-wise partial order: that is $a \leq b$ if and only if $a_i \leq b_i$ for all $1 \leq i \leq k$. This is a well partial order on $\mathbb{N}^k$.

If w is a word and a a letter, we note $a.w$ the word that begins with the letter a and ends with w. Similarly, if w is a stack and a a value, $a.w$ denotes the stack whose top is a and whose tail is w.

2 Stack machines

2.1 Syntax

A stack machine consists of a finite number of *registers*, each able to store a letter of an alphabet, and a finite number of *stack*s, that can be seen as lists of letters. Stacks can only be modified by usual `push` and `pop` operations, while registers can be modified by a given set of operators each of them assumed to be computed in a single unit of time.

Definition 62 ((Stack machine))**.** Stack machines are defined by the following grammar[2]:

(Alphabet)	Σ			finite set of symbols
(Programs)			$p ::=$	$\texttt{lbl}_1 : \texttt{i}_1; \dots; \texttt{lbl}_n : \texttt{i}_n;$
(Instructions)	$\mathcal{I}$	$\ni$	$\texttt{i} ::=$	`if` (test) `then goto` $\texttt{lbl}_0$ `else goto` $\texttt{lbl}_1$ |
				$\mathbf{r} := \texttt{pop}(\mathbf{stk}) \mid \texttt{push}(\mathbf{r}, \mathbf{stk}) \mid \mathbf{r} := \texttt{op}(\mathbf{r}_1, \cdots, \mathbf{r}_k) \mid \texttt{end}$
(Labels)	$\mathcal{L}$	$\ni$	`lbl`	finite set of labels
(Registers)	$\mathcal{R}$	$\ni$	$\mathbf{r}$	finite set of registers
(Stacks)	$\mathcal{S}$	$\ni$	$\mathbf{stk}$	finite set of stacks

Each operator has a fixed arity k and n is an integer constant. The syntax of a program induces a function $\texttt{next} : \mathcal{L} \to \mathcal{L}$ such that $\texttt{next}(\texttt{lbl}_i) = \texttt{lbl}_{i+1}$ and a mapping $\iota : \mathcal{L} \to \mathcal{I}$ such that $\iota(\texttt{lbl}_k) = \texttt{i}_k$. The `pop` operation removes the top symbol of a stack and puts it in a register. The `push` operation copies the symbol in the register onto the top of the stack. The `if`-instruction gives control to either $\texttt{lbl}_0$ or $\texttt{lbl}_1$ depending on the outcome of the test. Each operator is interpreted with respect to a given semantic function $[\![\texttt{op}]\!]$.

The precise sets of labels, registers and stacks can be inferred from the program. Hence if the alphabet is fixed, the machine can be identified with the program itself.

The syntax `lbl : if` (test) `then goto` $\texttt{lbl}_0$ can be used as a shorthand for `lbl : if` (*test*) `then goto` $\texttt{lbl}_0$ `else goto next(lbl)`. Similarly, we can abbreviate `if` true `then goto lbl` as `goto lbl`, that is an unconditional jump to a given label. The kinds of tests or operators allowed are not specified here. Of course, tests must be computable (for obvious reasons) in constant time and space, so that they do not play an important part when dealing with complexity properties. Comparisons between letters of the alphabet (*e.g.* $\leq$ if they are integers) or checking whether a stack is empty are typical tests that can be used.

If the alphabet contains a single letter, then the registers are useless and the stacks can be seen as unary numbers. The machine then becomes a standard counter machine [SS63].

[1]We use $s \in S$ to designate vertices and $a \in A$ to designate edges. The choice of using French initials ("Sommet" and "Arête") rather than the usual (V, E) is done to avoid confusion between vertices and the valuations introduced later.

[2]We use a bold face font for registers and stacks and a typewriter font for instructions

$$\frac{\mathtt{i} = \iota(\mathtt{IP}) = \mathbf{r} := \mathtt{op}(\mathbf{r}_1, \cdots, \mathbf{r}_k) \quad \sigma' = \sigma\{\mathbf{r} \leftarrow [\![\mathtt{op}]\!](\sigma(\mathbf{r}_1), \ldots, \sigma(\mathbf{r}_k))\}}{p \vdash \langle \mathtt{IP}, \sigma\rangle \xrightarrow{\mathtt{i}} \langle \mathtt{next(IP)}, \sigma'\rangle}$$

$$\frac{\iota(\mathtt{IP}) = \mathtt{if}\ \text{(test)}\ \mathtt{then\ goto\ lbl}_1\ \mathtt{else\ goto\ lbl}_2 \quad \text{(test) is true}}{p \vdash \langle \mathtt{IP}, \sigma\rangle \xrightarrow{\text{(test)}_{\text{true}}} \langle \mathtt{lbl}_1, \sigma\rangle}$$

$$\frac{\iota(\mathtt{IP}) = \mathtt{if}\ \text{(test)}\ \mathtt{then\ goto\ lbl}_1\ \mathtt{else\ goto\ lbl}_2 \quad \text{(test) is false}}{p \vdash \langle \mathtt{IP}, \sigma\rangle \xrightarrow{\text{(test)}_{\text{false}}} \langle \mathtt{lbl}_2, \sigma\rangle}$$

$$\frac{\mathtt{i} = \iota(\mathtt{IP}) = \mathbf{r} := \mathtt{pop}(\mathbf{stk}) \quad \sigma(\mathbf{stk}) = a.w \quad \sigma' = \sigma\{\mathbf{r} \leftarrow a, \mathbf{stk} \leftarrow w\}}{p \vdash \langle \mathtt{IP}, \sigma\rangle \xrightarrow{\mathtt{i}} \langle \mathtt{next(IP)}, \sigma'\rangle}$$

$$\frac{\mathtt{i} = \iota(\mathtt{IP}) = \mathbf{r} := \mathtt{pop}(\mathbf{stk}) \quad \sigma(\mathbf{stk}) = \epsilon}{p \vdash \langle \mathtt{IP}, \sigma\rangle \xrightarrow{\text{ß}} \perp}$$

$$\frac{\mathtt{i} = \iota(\mathtt{IP}) = \mathtt{push}(\mathbf{r}, \mathbf{stk}) \quad \sigma' = \sigma\{\mathbf{stk} \leftarrow \sigma(\mathbf{r}).\sigma(\mathbf{stk})\}}{p \vdash \langle \mathtt{IP}, \sigma\rangle \xrightarrow{\mathtt{i}} \langle \mathtt{next(IP)}, \sigma'\rangle}$$

Figure 1: Small steps semantics

Example 63. The following program reverses a list in stack $\mathbf{l}$ and puts the result in stack $\mathbf{l}'$ (assuming $\mathbf{l}'$ is empty at the beginning of the execution). It uses register $\mathbf{a}$ to store intermediate letters. The empty stack is denoted [].

```
0 : if l = [] then goto end;   |    3 : goto 0;
1 : a := pop(l);               |  end : end;
2 : push(a, l');               |
```

2.2 Semantics

Definition 64 ((Stores))**.** A *store* is a function σ assigning a symbol (letter of the alphabet) to each register and a finite string in Σ^* to each stack. Store update is denoted $\sigma\{x \leftarrow v\}$.

Definition 65 ((States))**.** Let p be a stack program. A *state* of p is a pair $\theta = \langle \mathtt{IP}, \sigma\rangle$ where the *Instruction Pointer* IP is a label and σ is a store. Let Θ be set of all states, Θ^* (Θ^ω) be the set of finite (infinite) sequences of states and $\Theta^{*\omega}$ be the union of the two.

Definition 66 ((Executions))**.** The operational semantics of Figure 1 defines a relation[3] $p \vdash \theta \xrightarrow{\mathtt{i}} \theta'$.

An *execution* of a program p is a sequence (finite or not) $p \vdash \theta_0 \xrightarrow{\mathtt{i}_1} \theta_1 \xrightarrow{\mathtt{i}_2} \ldots \xrightarrow{\mathtt{i}_n} \theta_n \ldots$

An infinite execution is said to be *non-terminating.* A finite execution is *terminating.* If the program admits no infinite execution, then it is *uniformly terminating.*

We use $\perp$ to denote runtime error. We may also allow operators to return $\perp$ if we want to allow operators that generate errors. It is important to notice that $\perp$ is not a state, and hence, is not considered when quantifying over all states.

If the instruction is not specified, we write simply $p \vdash \theta \to \theta'$ and use $\xrightarrow{+}$, $\xrightarrow{*}$ for the transitive and reflexive-transitive closures.

Definition 67 ((Traces))**.** The *trace* of an execution $p \vdash \theta_0 \xrightarrow{\mathtt{i}_1} \theta_1 \xrightarrow{\mathtt{i}_2} \ldots \xrightarrow{\mathtt{i}_n} \theta_n \ldots$ is the instructions sequence $\mathtt{i}_1 \ldots \mathtt{i}_n \ldots$

[3]Notice that the label i on the edge is technically not an instruction since for tests we also keep the information of which branch is taken.

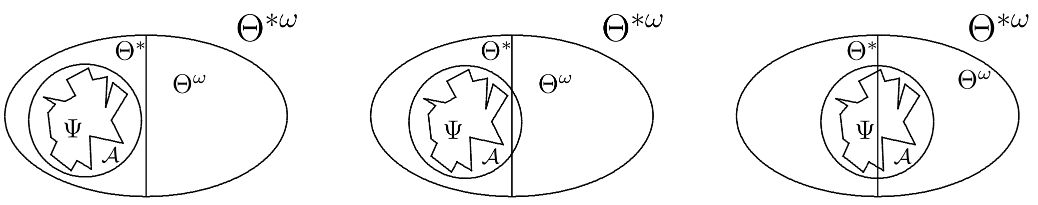


Figure 2: Sequences of states, executions and admissible sequences

Definition 68 ((Length))**.** Let $\theta = \langle \texttt{IP}, \sigma \rangle$ be a state. Its *length* $|\theta|$ is the sum of the number of elements in each stack[4]. That is:

$$|\theta| = \sum_{\mathbf{stk} \in \mathcal{S}} |\mathbf{stk}|$$

The length of a state corresponds to the usual notion of space consumption. Since there is a fixed number of registers and each can only store a finite number of different values, the space needed to store all registers is always bounded. So, we do not take registers into account while computing space usage.

The notion of length allows to define usual time and space complexity classes.

Definition 69 ((Running time, running space))**.** The *time usage* of an execution is the length of the corresponding sequence (possibly infinite for non-terminating programs). Let f be an increasing function from $\mathbb{N}$ to $\mathbb{N}$. We say that the *running- time* of a program is bounded by f if the time usage of each execution is bounded by $f(|\theta|)$ where θ is the first state of the execution.

The *space usage* of an execution is the least upper bound of the length of a state in it (if it exists, $+\infty$ otherwise). Let f be an increasing function from the non-negative integers to the non-negative integers. We say that the *running-space* of a program is bounded by f if the space usage of each execution is bounded by $f(|\theta|)$ where θ is the first state of the execution.

Definition 70 ((Complexity))**.** Let $f : \mathbb{N} \to \mathbb{N}$ be an increasing function. The class $T(f)$ is the set of functions that can be computed by a program whose running time is bounded by f. The class $S(f)$ is the set of functions that can be computed by a program whose running space is bounded by f.

FPTIME denotes the set of functions computable in polynomial time by a stack machine, that is $f \in$ FPTIME if and only if $f \in T(P)$ for some polynomial P.

If we want to define classes such as LOGSPACE, then we must, as usual, use some read-only input stacks that can only be `pop`ed but not `push`ed and some write-only output stacks. These play no role when computing the length of a state.

Stack machines are obviously Turing-complete and each model can simulate the other in a straightforward way.

3 A taste of RCG

This section describes the idea behind Resource Control Graphs in order to get a better grip on the formal definitions later on.

3.1 Admissible sequences

Consider an execution of a program. It can be described as a sequence of states. Clearly, not all sequences of states describe an execution. So we have a set of executions, Ψ, which is a subset of the set of all sequences of states (finite or infinite), $\Theta^{*\omega}$.

The undecidability results entail that, given a program, it is impossible to say whether the set Ψ of executions, and the set Θ^ω of infinite sequences of states, are disjoint. So, the idea here is to find a set $\mathcal{A}$ of *admissible* sequences that is a superset of the set of all executions, and whose intersection with Θ^ω can be computed. If this intersection is empty, then *a fortiori*, there are no infinite executions of the program; but if the intersection is not empty, then we cannot decide whether this is due to some non-terminating execution of the program or to some of the sequences added for the sake of the analysis. This means that depending on the machine considered and the way $\mathcal{A}$ is built, we can be in three different situations as depicted in Figure 2. We build $\mathcal{A} \supset \Psi$ such that $\mathcal{A} \bigcap \Theta^\omega$ is decidable. If it is empty, then the program uniformly terminates; otherwise, we cannot say anything. Of course, the undecidability theorem implies that if we require $\mathcal{A}$ to be recursive (or at least recursively separable from Θ^ω), then there is necessarily some

[4]Hence, it should more formally be $|\langle \texttt{IP}, \sigma \rangle| = \sum_{\mathbf{stk}_i \in \mathcal{S}} |\sigma(\mathbf{stk}_i)|$. Since explicitly mentioning the store everywhere would be quite unreadable, we use $\mathbf{stk}_i$ instead of $\sigma(\mathbf{stk}_i)$ and, similarly, $\mathbf{r}$ instead of $\sigma(\mathbf{r})$, when the context is clear.

programs for which the situation is the one in the middle (in Figure 2), i.e. the program uniformly terminates, but our analysis cannot determine that it does.

One conceptually simple way to represent all the possible executions (and only these), is to build a *state-transition graph*. This is a directed graph where each vertex is a state of the program and there is an edge from vertex x to y if and only if it is possible to go from state x to y in a single step of the operational semantics. Of course, since there are infinitely many different stores, there are infinitely many possible states and the graph is infinite.

3.2 The folding trick

Using the state-transition graph to represent executions is not convenient since handling an infinite graph can be tedious. To circumvent this, we must look into states and decompose them.

A state is actually a pair of one label and one store. The label corresponds to the *control* of the program while the store represents *memory*. A first try to get rid of the infinite state-transition graph is then to only consider the control part of each state.

Thus, there is only finitely many different nodes in the graph (since there are only finitely many different labels). By identifying all states bearing the same label, it becomes possible to "fold" the infinite state-transition graph into a finite graph, called the *Control Flow Graph* (CFG) of the program. The CFG is a usual tool for program analyses and transformations and can directly be built from the program.

Definition 71 ((Control Flow Graph))**.** Let p be a program. Its *Control Flow Graph* (CFG) is a directed graph $G = (S, A)$ where:

- $S = \mathcal{L}$. There is one vertex for each label.
- If $\iota(\texttt{lbl}) = \texttt{if}$ (test) $\texttt{then goto lbl}_1$ $\texttt{else goto lbl}_2$ then there is one edge from $\texttt{lbl}$ to $\texttt{lbl}_1$ labelled $(\text{test})_{\text{true}}$ and one from $\texttt{lbl}$ to $\texttt{lbl}_2$ labelled $(\text{test})_{\text{false}}$.
- If $\iota(\texttt{lbl}) = \texttt{end}$ then there is no edge going out of $\texttt{lbl}$.
- Otherwise, there is one edge from $\texttt{lbl}$ to $\texttt{next}(\texttt{lbl})$ labelled $\iota(\texttt{lbl})$.

Vertices and edges are named after, respectively, the label or instruction[5] they represent. No distinction is made between the vertex and the label or the edge and the instruction as long as the context is clear.

Example 72. The CFG of the reverse program is displayed on Figure 3.

With state-transition graphs, there was a one-to-one correspondence between executions of the program and (maximal) paths in the graph. This is no longer true with Control Flow Graphs. Now, to each execution corresponds a path (finite or infinite) in the CFG. The converse, however, is not true. There are paths in the CFG that correspond to no execution.

Let $\mathcal{P}$ be the set of paths in the CFG. Since we can associate a path to each execution, we can say that $\mathcal{P}$ is a superset of Ψ.

This leads to a first try at building a set of admissible[6] sequences by choosing $\mathcal{A} = \mathcal{P}$.

However, as soon as the graph contains loops, $\mathcal{P}$ contains infinite sequences. So this is quite a poor try at building an admissible set of sequences, corresponding exactly to the trivial analysis "*A program without loops uniformly terminates*".

In order to do better, we need to plug back the memory into the CFG.

3.3 Walks

So, in order to take memory into account but still keep the CFG, we do not consider vertices any more but states again. Clearly, each state is associated to a vertex of the CFG. Moreover, to each instruction $\texttt{i}$, we can associate a function $[\![\texttt{i}]\!]$ such that for all states θ, θ' for which $p \vdash \theta = \langle \texttt{IP}, \sigma\rangle \xrightarrow{\texttt{i}} \langle \texttt{IP}', \sigma'\rangle = \theta'$, we have $\sigma' = [\![\texttt{i}]\!](\sigma)$.

So, instead of considering paths in the graph, we can now consider walks. Walks are sequences of states following a path where each new store is computed according to the semantics function $[\![\texttt{i}]\!]$ of the edge just followed.

The only case where the CFG has out-degree greater than 1 is for tests. In order to prevent the wrong branch to be taken, the semantics function $[\![(\text{test})_{\text{true}}]\!]$ can be a partial function only defined for stores where the test is true (and conversely for the false branch of tests).

[5] Again, since the two branches of tests are separated, some edges do not correspond exactly to an instruction of the program. We nonetheless continue to call these "instructions".

[6] $\mathcal{P}$ is indeed recursive. By adapting Lemma 110, it is possible to show that $\mathcal{P}$ is an omega-regular language and hence can be recognised by a Büchi's automaton.

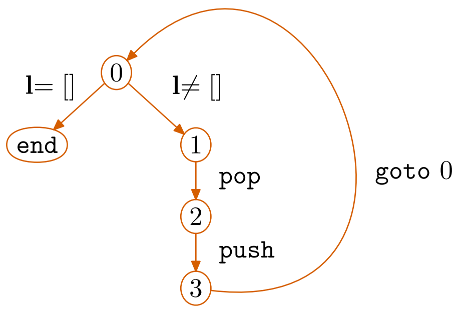


Figure 3: CFG of the reverse program.

But if we do this exactly that way, then there is a bijection between the executions and the walks and everything remains undecidable.

So the idea at this point is to keep both branches of the test possible, that is more or less replacing a deterministic test by a non-deterministic choice between the two outcomes. This leads to a set of walks bigger than the set of executions but, hopefully, recursively separable from the set of infinite sequences of states.

4 Monitoring space usage

In order to illustrate the ideas of the previous Section, we introduce here the notion of Resource Control Graph for the specific case of monitoring space usage. In Section 7, this notion is fully generalised to define Resource Control Graphs.

4.1 Space Resource Control Graphs

Definition 73 ((Weight))**.** For each instruction `i`, we define a *weight* $k_{\mathtt{i}}$ as follows:

- The weight of any instruction that is neither `push` nor `pop` is 0.
- The weight of a `push` instruction is +1.
- The weight of a `pop` instruction is −1.

Proposition 74. *For all states θ such that $p \vdash \theta \xrightarrow{i} \theta'$, we have $|\theta'| = |\theta| + k_{\mathtt{i}}$.*

It is important here that both θ and θ' are states. Indeed, this means that when an error occurs ($\perp$), nothing is said about the weight of the instruction causing the error.

Definition 75 ((Space Resource Control Graph))**.** Let p be a program. Its Space Resource Control Graph (Space-RCG) is a weighted directed graph G such that:

- G is the Control Flow Graph of p.
- For each edge `i`, the weight $\omega(\mathtt{i})$ is $k_{\mathtt{i}}$.

Definition 76 ((Configurations, walks))**.** Let $G = (S, A)$ be a graph. A *configuration* is a pair $\eta = (s, v)$ where $s \in S$ is a vertex and $v \in \mathbb{Z}$ is the *valuation*. A configuration is *admissible* if and only if $v \in \mathbb{N}$.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0) \xrightarrow{a_1} \ldots \xrightarrow{a_n} (s_n, v_n) \xrightarrow{a_{n+1}} \ldots$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \xrightarrow{a_{n+1}} \ldots$ and for all $i > 0$, $v_i = v_{i-1} + \omega(a_i)$. A walk is *admissible* if all its configurations are admissible.

Definition 77 ((Traces))**.** The *trace* of a walk is the sequence of all edges followed by the walk, in order.

Proposition 78. *Let p be a program, G be its Space-RCG and $p \vdash \theta_1 = \langle IP_1, \sigma_1 \rangle \to \ldots \to \theta_n = \langle IP_n, \sigma_n \rangle$ be an execution with trace t, then there is an admissible walk in G, $(IP_1, |\theta_1|) \to \ldots \to (IP_n, |\theta_n|)$, with the same trace t.*

Proof. By construction of the Space-RCG and induction on the length of the execution. □

4.2 Characterisation of Space usage

Theorem 79. *Let f be a total function $\mathbb{N} \to \mathbb{N}$. Let p be a program and G be its Space-RCG.*
$p \in S(f)$ if and only if for each initial state $\theta_0 = \langle IP_0, \sigma_0 \rangle$ with execution $p \vdash \theta_0 \overset{*}{\to} \theta_n$, the trace of the execution is also the trace of an admissible walk $(IP_0, |\theta_0|) \to (IP_1, v_1) \to \ldots \to (IP_n, v_n)$ where, for each k, $v_k \leq f(|\theta_0|)$.

Proof. Proposition 78 tells us that $v_k = |\theta_k|$. Then, both implications hold by definition of space usage. □

Definition 80 ((Resource awareness))**.** A Space-RCG is *f-resource aware* if for any admissible walk $(s_0, v_0) \overset{*}{\to} (s_n, v_n)$, $v_n \leq f(v_0)$.

Non-admissible walks are not taken into account because they never correspond to real executions of the program.

Corollary 81. *Let $f : \mathbb{N} \to \mathbb{N}$ be a total function, p be a program and G be its Space-RCG.*
If G is f-resource aware, then $p \in S(f)$.

Here, the converse is not true because the Space-RCG can have admissible walks with uncontrolled valuations that do not correspond to any real execution.

4.3 Non Size Increasingness

The study of Non Size Increasing (NSI) functions was introduce by [Hof99]. Former syntactical restrictions for Ptime, such as the safe recurrence of [BC92], forbid certain iterations of functions which can yield super-polynomial growth. However, Bellantoni-Cook approach excludes perfectly regular algorithms such as the insertion sort where the insertion function is iterated. The idea behind NSI is then that iterating functions *that do not increase the size of data* is harmless.

Hofmann detects Non Size Increasing programs in a typed functional language by adding a special type, $\diamond$, which can be seen as the type of pointers to free memory. Here, the valuations of Space RCG play exactly the same role as Hofmann's $\diamond$, that is managing the memory freed by previous de-allocation and reuse it rather than re-allocating new memory.

Even if this work was inspired by Hofmann's, there is currently no explicit link or equivalence Theorem between the programs detected by one or the other.

Definition 82 ((Non Size Increasing))**.** A program is *Non Size Increasing* (NSI) if its space usage is bounded by $\lambda x.x + \alpha$ for some constant α.

NSI is the class of functions that can be computed by Non Size Increasing programs. That is, $NSI = \bigcup_\alpha S(\lambda x.x + \alpha)$.

Proposition 83. *Let p be a program and G be its Space-RCG. If G is $\lambda x.x + \alpha$-resource aware for some constant α, then p is NSI.*

Proof. This is a direct consequence of Theorem 79. □

Theorem 84. *Let p be a program and G be its Space-RCG. G is $\lambda x.x + \alpha$-resource aware (for some α) if and only if it contains no cycle of strictly positive weight.*

Proof. If there is no cycle of strictly positive weight, then let α be the maximum weight of any path in G. Since there is no cycle of strictly positive weight, it is well-defined. Consider a walk $(s_0, v_0) \overset{*}{\to} (s_n, v_n)$ in G. Since α is the maximum weight of a path, we have $v_n \leq v_0 + \alpha$. Hence, G is $\lambda x.x + \alpha$-resource aware.

Conversely, if there is a cycle of strictly positive weight, then it can be followed infinitely many times and provides an admissible walk with unbounded valuations. □

Building the Space-RCG can be done in linear time in the size of the program. Finding the maximum weight of a path can be done in polynomial time in the size of the graph (and so in the size of the program) with Bellman-Ford's algorithm ([CLR90] Chapter 25.5). So we can detect NSI programs and find the constant α in polynomial time in the size of the program.

Example 85. The Space-RCG of the reverse program (from Example 63) is displayed on Figure 4. Since it contains no cycle of strictly positive weight, the program is Non Size Increasing. Moreover, since the maximum weight of any path is 1, it can be computed in space $\lambda x.x + 1$, that is the constant α is 1 for this program.

Actually, the reverse program can be computed in space $\lambda x.x$. This is not detected because we consider here all paths and not only paths starting from 0 (the initial node, corresponding to the first label of the program). This could be improved, and should be for any practical use, but the sharp bound is not needed in our theoretical framework.

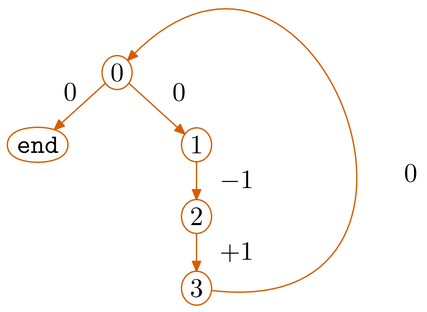


Figure 4: Space-RCG of the reverse program.

The set of NSI *programs* is undecidable. Hence, there exists some NSI programs that are not caught by the characterisation (*i.e.* have a Space-RCG with a cycle of positive weight). However, we now prove *extensional completeness*, that is for each NSI *function* f, there exists a program computing f whose Space-RCG has no cycle of positive weight.

Definition 86 ((Normalising programs))**.** Let p be a program and α be a constant. We define the program $\widetilde{p}_\alpha$ as follows:

- There is an extra stack **mem** (empty at start) and an extra symbol $\diamond$.
- The α first instructions of $\widetilde{p}_\alpha$ are `push`($\diamond$, **mem**). That is, $\widetilde{p}_\alpha$ starts by pushing α copies of $\diamond$ on **mem**.
- The following instructions of $\widetilde{p}_\alpha$ are the instructions of p, except that each `push` is followed by a `pop`(**mem**) and each `pop` is preceded by a `push`($\diamond$, **mem**).

Proposition 87. *If p runs in space $\lambda x.x + \alpha$, then $\widetilde{p}_\alpha$ computes the same function as p.*

Proof. As long as they do not cause runtime error (*i.e.* `pop`ing an empty stack), the added instructions do not interfere with the computed function because they only act on the new stack **mem**.

The only runtime error that **mem** can cause would be if one tries to `pop` it when its empty. However, if this happens, then a `push` on a non-**mem** stack must have happened just before while **mem** was empty. In the state just reached by this `push`, the sum of the lengths of the non-**mem** stacks would be $x + \alpha + 1$ where x is the length of the initial state. This contradicts the fact that p runs in space $\lambda x.x + \alpha$. □

Notice also that such a normalisation could be made for a program running in space $f(x)$ for any computable function f. However, in that case the simulation would require to compute $f(x)$ from the input and then push sufficiently many $\diamond$s. This would be quite tricky to do and require control over the space used to compute $f(x)$. Hence, adapting these ideas to classes of programs defined by their space complexity other than NSI cannot be done in a straightforward way and caution must be exerted.

Lemma 88. *Let p be a program. The Space-RCG of $\widetilde{p}_\alpha$ has no cycle of strictly positive weight.*

Proof. By construction of $\widetilde{p}_\alpha$. The first part (pushing α copies of $\diamond$ on **mem**) is done without any cycles in the control flow, by actually using α copies of the `push` instruction. The second part (simulating p) has each `push` paired with a `pop` (on another stack) and hence cannot generate paths of weight other than 0 or 1, and, especially, no cycle of weight different from 0. □

Theorem 89 ((Extensional completeness))**.** *Let f be a function in NSI. There exists a program p computing f whose Space-RCG has no cycle of strictly positive weight.*

Proof. Since f is in NSI, it is computed by a program q running in space $\lambda x.x + \alpha$ for some α. By Proposition 87 and Lemma 88, $p = \widetilde{q}_\alpha$ also computes f and has a Space-RCG without strictly positive cycle. □

This result means that this characterisation of NSI by programs whose Space-RCG have no cycle of strictly positive weight is extensionally complete: each function in NSI can be computed by a program that fits into the characterisation (that is, whose Space-RCG is $\lambda x.x+\alpha$-resource aware). Of course, intensional completeness (capturing all Non Size Increasing programs) is far from reached (but is unreachable with a decidable algorithm): there exist Non Size Increasing programs whose Space-RCG has cycle of positive weight (but, due to extensional completeness, the functions computed by these programs can also be computed by *another* program whose Space-RCG has no cycle of strictly positive weight).

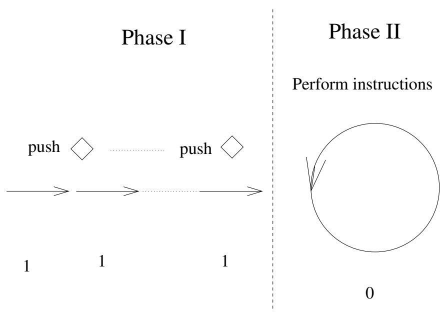


Figure 5: Space-RCG of a normalised program.

Example 90. The following two programs compute the same function, namely pushing five 0s onto stack **l**. The leftmost one uses a loop whose number of iterations is fixed inside the program by the assignment at label 0, while the rightmost one uses five copies of `push`.

Since the values of variables are not taken into account by the analysis, the Space-RCG of the first program has a loop of strictly positive weight corresponding to the loop in the program. Hence, this first program is Non Size Increasing, but not detected by the characterisation. This means that the characterisation is not *intensionally complete*: some programs have the wanted property but are left out. This is, obviously, an unwanted fact but nonetheless unavoidable because the set of Non Size Increasing programs is undecidable.

However, because of the extensional completeness, there must exist programs computing the same function that fit into the characterisation. The second program is an example of such a program. Indeed, its Control Flow Graph does not contain any cycles, hence, the Space-RCG does not have cycles either (and so, no cycle of positive weight).

```
  0 : a := 5;                        |   0 : push(0, l);
  1 : if a = 0 then goto end;        |   1 : push(0, l);
  2 : push(0, l);                    |   2 : push(0, l);
  3 : a := a − 1;                    |   3 : push(0, l);
  4 : goto 1;                        |   4 : push(0, l);
end : end                            | end : end
```

If one applies the normalisation of Definition 86 to the first program, one obtains:

```
0 : push(◇, mem);     |   6 : if a = 0 then goto end;
1 : push(◇, mem);     |   7 : pop(mem);
2 : push(◇, mem);     |   8 : push(0, l);
3 : push(◇, mem);     |   9 : a := a − 1;
4 : push(◇, mem);     |  10 : goto 6;
5 : a = 5;            | end : end;
```

This program computes the same function. The first part (in the leftmost column) allocates a constant amount of memory on a "free memory" stack. Since the amount of memory needed (the constant α in the Lemmas and Theorems above) is known, this can be done with no loops by using several copies of `push`. Then, the program mimics the loop of the first program. However, before allocating any new memory (that is, before performing any `push`), it starts by removing some free memory from **mem** (with a `pop`). This ensures that no cycle in the Space-RCG has a strictly positive weight.

4.4 Linear Space

Linear space seems to be closely related to NSI. Indeed, linear space functions can be computed in space $\lambda x.\beta x + \alpha$ and so NSI is a special case with $\beta = 1$. Hence we want to try and adapt our result to detect linear space usage.

Definition 91. FLinSpace denotes the set of functions computable in linear space by a stack machine, that if $f \in$ FLinSpace if and only if $f \in S(l)$ for some linear function $l : x \mapsto \beta x + \alpha$.

The idea is quite easy: since we are allowed to use a factor of β more space than what is initially allocated, it is sufficient to consider that every time some of the initial data is freed, β "tokens" ($\diamond$) are released and can later be used to control β different allocations.

In order to do so, the most convenient way is to design certain stacks of the machine as *input stacks* and the others must be initially empty. Then, a `pop` operation on an input stack would have weight $-\beta$ instead of simply -1 to account for this linear factor. However, doing so we must be careful that newly allocated memory (that is, further `push`) is only counted as 1 when freed again (to avoid a cycle of freeing one slot, allocating β, freeing these β slots and reallocating β^2 and so on). In order to do so, we simply require that the input stacks are read-only in the sense that it is not possible to perform a `push` operation on them.

Notice that any program can be turned into such a program by having twice as many stacks (one input and one work for each) and starting by copying all the input stacks into the corresponding working stacks and then only dealing with the working stacks.

With these programs, the invariant is not the length of states, but something slightly more complicated, namely β times the length of input stacks plus the length of work stacks. We call this measure *β-size.* Globally, we use size to denote some kind of measure on states that is used by the RCG for analysis. The terminology is close to the one used for Size Change Termination [LJBA01], where values are assumed to have some (well-founded) "size ordering" which is not specified and not necessarily related to the actual space usage of the data. Typically, termination of a program working over positive integers can be proved using the usual ordering on $\mathbb{N}$ as size ordering, even if the integers are all 32 bits integers, thus taking exactly the same space in memory.

Definition 92 ((Extended stack machines)). An *extended stack machine* is a stack machine with the following modification:

There are two disjoint sets of stacks, $\mathcal{S}_i$ is the set of *input stacks* and $\mathcal{S}_w$ is the set of *working stacks.* There are two instructions $\mathtt{pop}_i$ and $\mathtt{pop}_w$ depending on whether an input or working stack is considered but only one $\mathtt{push} = \mathtt{push}_w$ instruction, that is it is impossible to `push` anything on an input stack.

The *β-size* of a state is β times the length of input stacks plus the length of working stacks, that is:

$$||\theta||_\beta = \beta \sum_{\mathbf{stk}_i \in \mathcal{S}_i} |\mathbf{stk}_i| + \sum_{\mathbf{stk}_w \in \mathcal{S}_w} |\mathbf{stk}_w|$$

The *weight* of $\mathtt{pop}_i$ is $-\beta$, the weight of $\mathtt{pop}_w$ is -1, the weight of `push` is $+1$. The weight of any other instruction is 0.

The β-Space RCG is built as the Space-RCG: the underlying graph is the control flow graph and the weight of each edge is the weight of the corresponding instruction.

Proposition 78 becomes:

Proposition 93. *Let p be a program, G_β be its β-Space RCG and $p \vdash \theta_1 = \langle IP_1, \sigma_1\rangle \to \ldots \to \theta_n = \langle IP_n, \sigma_n\rangle$ be an execution with trace t, then there is an admissible walk $(IP_1, ||\theta_1||_\beta) \to \ldots \to (IP_n, ||\theta_n||_\beta)$ with the same trace t.*

Then, adapting Theorem 79 and Theorem 84, we have:

Proposition 94. *Let p be a program and G_β be its β-Space RCG. If G_β is $\lambda x.x + \alpha$-resource aware for some constant α, then $p \in S(\lambda x.\beta x + \alpha)$.*

Theorem 95. *Let p be a program and G_β be its β-Space RCG. G_β is $\lambda x.x + \alpha$-resource aware (for some α) if and only if it contains no cycle of strictly positive weight.*

Corollary 96. *Let p be a program. If there exists β such that its β-Space RCG contains no cycle of strictly positive weight, then p computes a function in FLINSPACE.*

This can be checked in NPTIME since β is polynomially bounded in the size of the program.

Also for FLINSPACE, the normalisation process of programs can be performed. The first phase of the normalised program consists in first pushing onto **mem** α copies of $\diamond$, then repeatedly copying each input stack onto the corresponding working stack and each time a symbol is copied, $\beta - 1$ new $\diamond$s are pushed onto **mem** (so that the global space usage from now on is always $\beta x + \alpha$). This means that here also the characterisation is extensionally complete: for each FLINSPACE function, there exists one program computing it that fits into the characterisation.

Example 97. The following program "double-reverses" a list. It is similar to the reverse program but each element is present twice in the result. The list **l** is an input stack (and hence cannot be `push`ed) while **l**′ is a working stack.

```
  0 : if l = [] then goto end;  |   3 : push_w(a, l');
  1 : a := pop_i(l);            |   4 : goto 0;
  2 : push_w(a, l');            | end : end;
```

Its β-Space RCG is displayed on Figure 6. Since it contains no cycle of strictly positive weight if $\beta \geq 2$, the program is in LINSPACE. More precisely, it can be computed in space $\lambda x.2x$

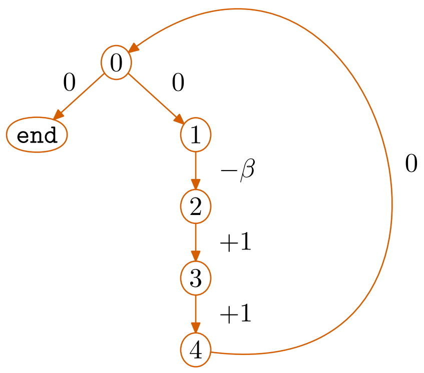


Figure 6: β-Space RCG of the double-reverse program.

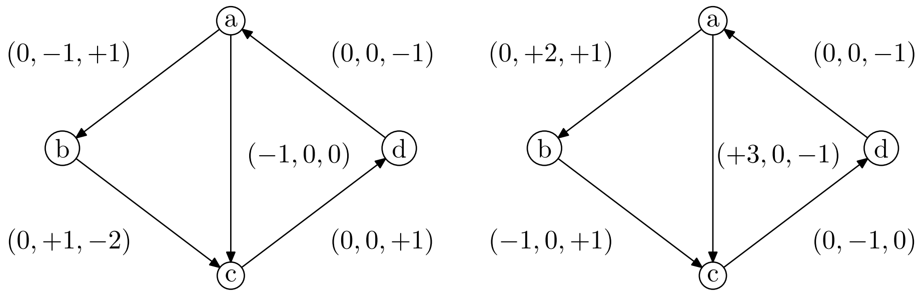


Figure 7: Two VASS

5 Vector Addition System with States

This section describes Vector Addition Systems with States (VASS) which are known to be equivalent to Petri Nets [Reu89]. Resources Control Graphs are a generalisation of VASS.

5.1 Definitions

Definition 98 ((VASS, configurations, walks))**.** A *Vector Addition System with States* is a directed graph $G = (S, A)$ together with a *weight function* $\omega : A \to \mathbb{Z}^k$ where k is a fixed integer.

A *configuration* is a pair $\eta = (s, v)$ where $s \in S$ is a vertex and $v \in \mathbb{Z}^k$ is the *valuation*. A configuration is *admissible* if and only if $v \in \mathbb{N}^k$.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0)\xrightarrow{a_1}\ldots\xrightarrow{a_n}(s_n, v_n)$ such that $s_0\xrightarrow{a_1}s_1\xrightarrow{a_2}\ldots\xrightarrow{a_n}s_n$ and for all $i > 0$, $v_i = v_{i-1} + \omega(a_i)$. A walk is *admissible* if all its configurations are admissible.

We say that the path $a_1 \ldots a_n$ is the *underlying path* of the walk and the walk *follows* this path. Similarly, G is the *underlying graph* for the VASS.

As for graphs and paths, we write $\eta \to \eta'$ if there exists an edge a such that $\eta\xrightarrow{a}\eta'$ and $\xrightarrow{+}$, $\xrightarrow{*}$ for the closures.

Definition 99 ((Weight of a path))**.** Let V be a VASS and $a_1 \ldots a_n$ be a path in it. The weight of edges is extended to paths canonically: $\omega(a_1 \ldots a_n) = \sum \omega(a_i)$, *i.e.* ω is a morphism between $(A, \cdot)$ (the free monoid generated by the edges) and $(\mathbb{Z}^k, +)$.

Example 100. Figure 7 displays two VASS. More formally, the first one should be described as a graph $G = (S, A)$ with:

- $S = \{a, b, c, d\}$
- $A = \{a\xrightarrow{a_1}b, a\xrightarrow{a_2}c, b\xrightarrow{a_3}c, c\xrightarrow{a_4}d, d\xrightarrow{a_5}a\}$
- $\omega(a_1) = (0, -1, +1)$, $\omega(a_2) = (-1, 0, 0)$, $\omega(a_3) = (0, +1, -2)$, $\omega(a_4) = (0, 0, +1)$, $\omega(a_5) = (0, 0, -1)$.

Every finite path is the underlying path of an admissible walk:

Lemma 101. *Let V be a VASS and $a_1 \ldots a_n$ be a finite path in it. There exists a valuation v_0 such that for $0 \leq i \leq n$, $v_0 + \omega(a_1 \ldots a_i) \in \mathbb{N}^k$.*

Proof. Because the path is finite, the jth component of $\omega(a_1 \dots a_i)$ is greater than some α_j (of course, this bound is not necessarily reached with the same i for all components, but nonetheless such a bound exists for each component separately). By putting $\beta_j = \max(0, -\alpha_j)$ (that is 0 if α_j is positive), then $v_0 = (\beta_1, \cdots, \beta_k)$ verifies the property. □

Example 102. Let us consider the first VASS of Figure 7 and the path $a \xrightarrow{a_1} b \xrightarrow{a_3} c$. This path has weight $(0, 0, -1)$. If we consider the initial valuation $(0, 0, 0)$ and the corresponding walk $(a, (0, 0, 0)) \xrightarrow{a_1} (b, (0, -1, 1)) \xrightarrow{a_3} (c, (0, 0, -1))$, this walk is not admissible because the second and third valuations have a strictly negative coefficient. However, following the same path, it is always possible to take a initial valuation "big enough" in order to get an admissible walk. Here, the walk $(a, (0, 1, 1)) \xrightarrow{a_1} (b, (0, 0, 2)) \xrightarrow{a_3} (c, (0, 1, 0))$ follows the same path and is admissible.

Lemma 103. *Let $(s_0, v_0) \to \dots \to (s_n, v_n)$ be an admissible walk in a VASS. Then, for all $v'_0 \geq v_0$ (component-wise comparison), $(s_0, v'_0) \to \dots \to (s_n, v'_n)$ is an admissible walk (following the same path).*

Proof. By monotonicity of the addition. □

Example 104. Continuing the previous example, any valuation larger (component-wisely) than $(0, 1, 1)$ leads to an admissible walk when following the same path. So, among other, the walk starting at $(a, (17, 14, 42))$ and following edges a_1 and a_3 is admissible.

Definition 105 ((Uniform termination))**.** A VASS is said to be *uniformly terminating* if it admits no infinite admissible walk. That is, every walk is either finite or reaches a non-admissible configuration.

Theorem 106. *A VASS is* not *uniformly terminating if and only if there exists a cycle whose weight is in $\mathbb{N}^k$ (that is, is non-negative with respect to each component).*

Proof. If such a cycle exists, starting and ending at vertex s, then by Lemma 101 there exists v_0 such that the walk starting at (s, v_0) and following the cycle is admissible. After following the cycle once, the configuration (s, v_1) is reached. Since the weight of the cycle is non-negative, $v_1 \geq v_0$. Then, by Lemma 103 the walk can follow the cycle one more time, reaching (s, v_2), and still be admissible. By iterating this process, it is possible to build an infinite admissible walk.

Conversely, let $(s_0, v_0) \to \dots \to (s_n, v_n) \to \dots$ be an infinite admissible walk. Since there are only finitely many vertices, there exists at least one vertex s' appearing infinitely many times in it. Let (s'_l, v'_l) be the occurrences of the corresponding configurations in the walk. Since the component-wise order over vectors of $\mathbb{N}^k$ is a well partial order, there exists i, j such that $v'_i \leq v'_j$. The cycle followed between s'_i and s'_j has a non-negative weight. □

Example 107. The second VASS of Figure 7 is not uniformly terminating. Indeed, if we consider the cycle $C = a_1a_3a_4a_5a_1a_3a_4a_5a_2a_4a_5$, it has weight $(1, 1, 0)$. By Lemma 101, there exists a valuation $v = (m, n, p)$ such that the walk starting at $(a, (m, n, p))$ and following this cycle is admissible (it is sufficient to choose $(m, n, p) \geq (2, 0, 0)$). After following the cycle once, the configuration is $(a, (m + 1, n + 1, p))$ from which the cycle can be followed once again, thus reaching $(a, (m + 2, n + 2, p))$. Repeating this leads to an infinite admissible walk.

5.2 Decidability of the uniform termination

Definition 108 ((Linear parts, semi-linear parts))**.** Let $(M, +)$ be a commutative monoid[7]. A *linear part* of M is a subset of the form $v + V^*$ where $v \in M$ and V is a finite subset of M. That is, if $V = \{v_1, \cdots, v_p\}$, a linear part can be expressed as:

$$\left\{ v + \sum_{i=1}^{i=p} n_i v_i | n_i \in \mathbb{N} \right\}$$

A *semi-linear part* of M is a finite union of linear parts.

Recall that rational parts are built from +, union and Kleene's star *. When dealing with words (that is the free monoid generated by a finite alphabet), + is word concatenation (not commutative) and so rational parts are exactly the regular languages.

Lemma 109. *In a commutative monoid, semi-linear parts are exactly the rational parts.*

Proof. Semi-linear parts are expressed as rational parts.

Conversely, it is sufficient to show that the set of semi-linear parts contains all finite parts and is closed by union, sum and *.

Semi-linear parts contains finite part and are closed under union by definition. Closure under sum is obtained because $(a + A^*) + (b + B^*) = (a + b) + (A \bigcup B)^*$ and sum is distributive over union $((A \bigcup B) + C = (A + C) \bigcup (B + C))$.

[7]Recall that a commutative monoid is a monoid (a set with an associative internal operation admitting a neutral element) whose operation is commutative. A typical example of commutative monoid is $(\mathbb{N}, \times)$ (inverse for each element is not needed).

The hard point is the closure under * which is a consequence of commutativity. It holds because $(v+V^*)^* = (v+(\{v\}\bigcup V)^*)\bigcup\{0\}$ (the key idea being that $(a(b^*))^* = a^*b^*$ in a commutative monoid). See [Reu89] (Proposition 3.5) for details. □

Lemma 110. *The set of non-empty cycles in a graph is a rational part (of the free monoid generated by the edges).*

Proof. Consider the graph as an automaton with each edge labelled by a unique label. The set of paths between two given vertices is a regular language; specifically, the set of cycles that begin and end at vertex s is regular. The full set of cycles is the union of those sets over all the (finitely many) vertices, and is consequently also regular. □

Example 111. Consider again the VASS of Figure 7 or, rather, their underlying graph. The set of (possibly empty) cycles from a to itself is described by the regular expression $A = ((a_1a_3a_4a_5)|(a_2a_4a_5))^*$ and corresponds exactly to the rational language recognised by this expression. Then the set of all (non-empty) cycles in these VASS is the language recognised by the regular expression:

$$(((a_1a_3a_4a_5)|(a_2a_4a_5))A)|(a_3a_4a_5Aa_1)|(a_4a_5A(a_1a_3|a_2))|(a_5A(a_1a_3|a_2)a_4)$$

where each of the four alternatives corresponds to the set of (non-empty) cycles from one vertex to itself.

Corollary 112. *The set of weights of (non-empty) cycles in a VASS is a semi-linear part of $\mathbb{Z}^k$.*

Proof. Since the weight function ω is a morphism between $(A,\cdot)$ and $(\mathbb{Z}^k,+)$, it preserves rational parts. Hence, the set of weights of cycles is a rational part of $\mathbb{Z}^k$. Since $+$ is commutative, it is also a semi-linear part. □

Notice that the proofs are constructive. Hence the semi-linear part can be built effectively.

Example 113. For concision, we write here ω_i instead of $\omega(a_i)$.

First, let us look at the weights of cycles from a to itself. By applying the weight morphism (ω) to A, we obtain the regular expression:

$$((\omega_1+\omega_3+\omega_4+\omega_5)|(\omega_2+\omega_4+\omega_5))^*$$

To express this as a semi-linear part, we must change the alternatives (|) into union of sets. This leads to:

$$\{\omega_1+\omega_2+\omega_4+\omega_5, \omega_2+\omega_4+\omega_5\}^*$$

which is a semi-linear part of $\mathbb{Z}^3$. For the first VASS, this is:

$$\{(0,0,-1),(-1,0,0)\}^* = \{(-k,0,-l)|k,l\in\mathbb{N}\}$$

Next, consider the expression describing cycles from b to itself: $a_3a_4a_5Aa_1$. When applying the weight to it, we obtain:

$$\{\omega_3+\omega_4+\omega_5\}+\{\omega_1+\omega_2+\omega_4+\omega_5,\omega_2+\omega_4+\omega_5\}^*+\{\omega_1\}$$

By commutativity of addition, this can be expressed as the semi-linear part:

$$(\omega_1+\omega_3+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5,\omega_2+\omega_4+\omega_5\}^*$$

Again, if we consider the first VASS, this is:

$$(0,0,-1)+\{(0,0,-1),(-1,0,0)\}^* = \{(-k,0,-l)|k,l\in\mathbb{N},l>0\}$$

Then, we must do the same work for the three other alternatives (corresponding to cycles from a, c and d). This leads, to the following semi-linear parts:

- for a, c and d: $(\omega_1+\omega_3+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5,\omega_2+\omega_4+\omega_5\}^*\bigcup(\omega_2+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5,\omega_2+\omega_4+\omega_5\}^*$
- for b: $(\omega_1+\omega_3+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5,\omega_2+\omega_4+\omega_5\}^*$

The expression for b is different from the others because non-empty cycles at b must go at least once through the large cycle while other non-empty cycle can go through the small cycle only.

The resulting semi-linear part of $\mathbb{Z}^k$ describing weights of cycles corresponds to the union of these semi-linear parts, namely:

$$\{\ (\omega_1+\omega_3+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5,\omega_2+\omega_4+\omega_5\}^*\ \}$$
$$\bigcup\ \{\ (\omega_2+\omega_4+\omega_5)+\{\omega_1+\omega_2+\omega_4+\omega_5,\omega_2+\omega_4+\omega_5\}^*\ \}$$

For the first VASS, this is:

$$((0,0,-1)+\{(0,0,-1),(-1,0,0)\}^*)\bigcup((-1,0,0)+\{(0,0,-1),(-1,0,0)\}^*)$$
$$=\{(-k,0,-l)|k,l\in\mathbb{N},k+l>0\}$$

For the second VASS, this is:

$$((-1,1,1)+\{(-1,1,1),(3,-1,-2)\}^*)\bigcup((3,-1,-2)+\{(-1,1,1),(3,-1,-2)\}^*)$$
$$=\{(3l-k,k-l,k-2l)|k,l\in\mathbb{N},k+l>0\}$$

Theorem 114. *Uniform termination of VASS is in NPtime.*

Proof. By Theorem 106, a VASS is *not* uniformly terminating if and only if there is a cycle whose weight is in $\mathbb{N}^k$. Since the set of weights for all cycles is a semi-linear part of $\mathbb{Z}^k$, we can consider in turn each linear part in it. The full set intersects $\mathbb{N}^k$ if and only if at least one of its linear parts does. Thus, all we need is a method for deciding whether a linear part of $\mathbb{Z}^k$ intersects with $\mathbb{N}^k$:

Let $U=\{u_1,\cdots,u_p\}$ and $u+U^*$ be a linear part of $\mathbb{Z}^k$. It intersects $\mathbb{N}^k$ if and only if there exist $n_1,\cdots,n_p\in\mathbb{N}$ such that $u+\sum n_iu_i\geq 0$.

This can be solved in NPtime using usual integer linear programming techniques. □

Since VASS and Petri nets are equivalent, this also shows that uniform termination of Petri nets is decidable. Without going through the equivalence, a direct and simpler proof can be made for Petri nets. Such a proof can be found in [Moy03], (Theorem 60, page 83).

Example 115. Consider again the two VASS of Figure 7. The set of weights of non-empty cycles of the first VASS corresponds to the semi-linear part:

$$((0,0,-1)+\{(0,0,-1),(-1,0,0)\}^*)\bigcup((-1,0,0)+\{(0,0,-1),(-1,0,0)\}^*)$$

The first linear part of the union, $(0,0,-1)+\{(0,0,-1),(-1,0,0)\}^*$, intersects $\mathbb{N}^3$ if and only if there exist $n_1,n_2\in\mathbb{N}$ such that:

$$(0,0,-1)+n_1\times(0,0,-1)+n_2\times(-1,0,0)\geq(0,0,0)$$

This is clearly impossible.

Similarly, the second linear part cannot intersect $\mathbb{N}^3$. Hence, the set of weights of non-empty cycles does not intersect $\mathbb{N}^3$ and the VASS is uniformly terminating.

For the second VASS, the weights correspond to the semi-linear part:

$$((-1,1,1)+\{(-1,1,1),(3,-1,-2)\}^*)\bigcup((3,-1,-2)+\{(-1,1,1),(3,-1,-2)\}^*)$$

There, for the first linear-part, the system becomes:

$$(-1,1,1)+n_1\times(-1,1,1)+n_2\times(3,-1,-2)\geq(0,0,0)$$

Usual Integer Linear Programming techniques show that the system has a solution, for example with $n_1=1,n_2=1$, corresponding to the cycle $(a_1a_3a_4a_5)^2(a_2a_4a_5)$ whose weight is $(1,1,0)$. Hence, the VASS is not uniformly terminating.

However, any infinite walk starting from, for example, the configuration $(a,(0,14,0))$ is not admissible. Deciding whether a given configuration leads to an infinite admissible walk or not is a different problem from uniform termination.

It is worth noticing that in the second case, the cycle detected is *not* a simple cycle. So the problem is different from the one of detecting simple cycles in graphs and requires a specific solution.

5.3 VASS as Resource Control Graphs

Before the formal definition of Resource Control Graphs, we show here how VASS can be used to build proofs of uniform termination of programs.

In the rest of this section, we consider the following size function:

$$||\langle\texttt{IP},\sigma\rangle||=(|\mathbf{stk}_1|,\cdots,|\mathbf{stk}_s|)_{\mathbf{stk}_i\in\mathcal{S}}$$

that is, the vector whose components are the lengths of the different stacks of a given program. Moreover, we use (e_i) to denote the canonical basis of $\mathbb{Z}^k$, that is e_i is the vector whose jth component is $\delta_{i,j}$.

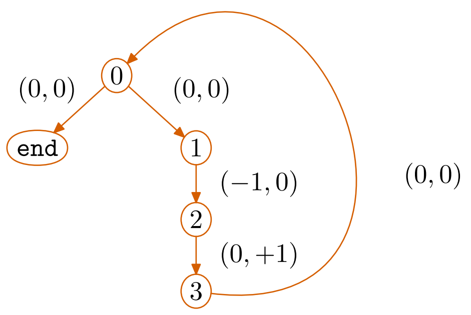


Figure 8: The Resource Control VASS for the reverse program

Definition 116 ((Weights))**.** To each instruction, we assign the following *weight*:

- $\omega(\mathbf{r} := \mathtt{pop}(\mathbf{stk}_i)) = -e_i$
- $\omega(\mathtt{push}(\mathbf{r}, \mathbf{stk}_i)) = e_i$
- $\omega(\mathtt{i}) = 0$ for all other instructions.

Definition 117 ((Resource Control VASS))**.** Let p be a program. Its *Resource Control VASS* is a VASS whose underlying graph is the Control Flow Graph of p and edge $\mathtt{i}$ has weight $\omega(\mathtt{i})$ as defined above.

Proposition 118. *Let p be a program and G be its Resource Control VASS. If $\theta_0 = \langle IP_0, \sigma_0\rangle \xrightarrow{*} \langle IP_n, \sigma_n\rangle = \theta_n$ is an execution of p, then $(IP_0, ||\theta_0||) \xrightarrow{*} (IP_n, ||\theta_n||)$ is an admissible walk of G with the same trace.*

Proof. By induction on the length of the execution. Notice that executions leading to errors ($\perp$) are not taken into account here. □

Theorem 119. *Let p be a program and G be its Resource Control VASS. If G is uniformly terminating, then p is uniformly terminating.*

Proof. If p is not uniformly terminating, then by the previous proposition there exists an infinitely long execution that can be mapped onto an infinite admissible walk. □

Since uniform termination of VASS is decidable, this allows to detect uniform termination of a broad class of programs. Of course, the converse is not true since uniform termination of programs is not decidable.

Example 120. The Resource Control VASS of the reverse program is displayed on Figure 8. Since it is uniformly terminating, so is the reverse program.

Weighted graphs, as used in Section 4 to prove Non-Size Increasingness of programs are the special case of VASS when the dimension is one.

6 Resource Systems with States

Resource Systems with States (RSS) are a generalisation of the VASS seen in the previous section. For VASS, the only information kept is a vector of integers, and only addition of vectors can be performed. When modelling programs, this is not sufficient. Indeed, if one wants to closely represent the memory of a stack machine, a vector is not sufficient. Moreover, vector addition is not powerful enough to represent common operations such as copy of a variable ($x := y$).

Hence, we now relax the constraints on valuations and weights. We allow valuations to be drawn from any set and allow as weight any function mapping valuations to valuations. Notice that in the case of VASS, each weight is addition of a vector v, which could be represented as the function $\lambda x.x + v$.

For the sake of generality, we even allow the sets of valuations to be different for each vertex. This may seem strange, but a typical use of that is to have vectors with different numbers of components as valuations (that is the set of valuations for vertex s_i would be $\mathbb{Z}^{k_i}$) and matrix multiplications as weights (where the matrices have the correct number of rows and columns). Of course, one can always take the (disjoint) union of these sets, but this usually makes the notations harder to read and understand. See Example 157 for more details.

6.1 Graphs and States

Definition 121 ((RSS, configurations, walks))**.** A *Resource System with States* (RSS) is a tuple (G, V, V^+, W, ω) where

- $G = (S, A)$ is a directed graph, $S = \{s_1, \cdots, s_n\}$ is the set of vertices and $A = \{a_1, \cdots, a_m\}$ is the set of edges.
- $V_1, \cdots, V_n$ are the sets of *valuations*. V is their union.
- $V_i^+ \subset V_i$ are the sets of *admissible valuations*. V^+ is the union of them.
- $W_{i,j} : V_i \to V_j$ are the sets of *weights*. W is the union of them.
- $\omega : A \to W$ is the *weight function* such that $\omega(a) \in W_{i,j}$ if $s_i \xrightarrow{a} s_j$.

When it is clear what both the valuations and weight sets are, we name the RSS after the underlying graph G.

A *configuration* is a pair $\eta = (s, v)$ where $s = s_i \in S$ is a vertex of the graph and $v \in V_i$ is a valuation. A configuration is *admissible* if $v \in V_i^+$ is admissible.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0) \xrightarrow{a_1} \ldots \xrightarrow{a_n} (s_n, v_n) \xrightarrow{a_{n+1}} \ldots$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \xrightarrow{a_{n+1}} \ldots$ and for all $i > 0$, $v_i = \omega(a_i)(v_{i-1})$. A walk is *admissible* if all configurations in it are admissible.

The walk *follows* path p which is called either *underlying path* or *trace* of the walk.

As earlier, we write $\eta \to \eta'$ if the relation holds for an unspecified edge and $\xrightarrow{+}$, $\xrightarrow{*}$ for the transitive and reflexive-transitive closures.

The idea behind having both valuations and admissible valuations is that it allows V to have nice algebraic properties not shared by V^+. Moreover, it also allows the set of valuations to be the closure of the admissible valuations under the weight functions, thus removing the deadlock problem of reaching something that would not be a valuation (and replacing it by the more semantical problem of detecting non admissible valuations). Typically with VASS, V is the ring $\mathbb{Z}^k$, and V^+ is $\mathbb{N}^k$. Since weights can add any vector, with positive or negative components, to a valuation, V is the closure of V^+ under this operation. Moreover, VASS do not suffer from the deadlock problems that appear in Petri nets (but this is done by introducing the problem of deciding whether a walk is admissible).

Notice that either unions (for V, V^+ or W) can be considered to be a disjoint union without loss of generality.

Definition 122 ((Weight of a path))**.** Let G be an RSS. The weight function can be canonically extended over all paths in G by choosing $\omega(ab) = \omega(b) \circ \omega(a)$.

$(W, \circ)$ is a magma. It is not a monoid because the identity is not unique. There is a finite set of neutral elements, the identities over each V_i.

Notice that we do not actually need the whole W. Only the part generated by the individual weights of edges is necessary to handle an RSS. We overload the notation and call it W as well.

In the following, to improve readability, we write $v \circledast \omega(a)$ instead of $\omega(a)(v)$ and $\omega(a) \fatsemi \omega(b)$ instead of $\omega(b) \circ \omega(a)$. When following paths, we now have: $\omega(ab) = \omega(b) \circ \omega(a) = \omega(a) \fatsemi \omega(b)$. So, this allows for a more natural expression of weights of paths[8].

Example 123. For the VASS of the previous section, we have $V_i = \mathbb{Z}^k$ and $V_i^+ = \mathbb{N}^k$ for all i, and $\omega(a_i) = \lambda x.x + u_i$ for some vector $u_i \in \mathbb{Z}^k$. Or, we could describe VASS by saying that $V = W = \mathbb{Z}^k$, $V^+ = \mathbb{N}^k$ and $\circledast = \fatsemi = +$.

The notation with $\circledast$ and $\fatsemi$ is much more convenient, especially to easily handle weights of paths, as is done in the lemmas and theorems of the previous section.

If we consider V_i as objects and $\omega \in W$ as arrows, we have a category. Indeed, identity exists for each V_i and composition of two arrows is properly defined.

6.2 Properties of RSS

Order

Definition 124 ((Ordered RSS))**.** An *ordered RSS* is an RSS $G = (G, V, V^+, W, \omega)$ together with a partial ordering $\prec$ over valuations such that the restriction of $\prec$ to V^+ is a well partial order.

For VASS, the component-wise order on vectors of the same length is the well partial order (over $V^+ = \mathbb{N}^k$) that was used in the previous section.

[8]From an algebraic point of view, ω is considered as a morphism between $(A, \cdot)$ and $(W, \fatsemi)$, and $\circledast$ is a right-action of W on V. Moreover, $(W, \fatsemi)$ often appears to be isomorphic to a well known structure (usually a group, such as $(\mathbb{Z}^k, +)$ for VASS).

Definition 125 ((Monotonicity, positivity))**.** Let (G, V, V^+, W, ω) be an ordered RSS. We say that it is *increasing* if all weight functions $\omega(a_i)$ are increasing with respect to $\prec$. Since the composition of increasing functions is still increasing, the weight function of any path is increasing.

We say that (G, V, V^+, W, ω) is *positive* if for each $v \in V^+$ and $v' \in V$, $v \prec v'$ implies $v' \in V^+$.

VASS are both increasing and positive. Monotonicity is the key of Lemma 103 while positivity is implicitly used in the proof of Theorem 106 to say that the valuation reached after one cycle is still admissible.

Definition 126 ((Resource awareness))**.** Let G be an ordered RSS and $f : V \to V$ be a function. G is *f-resource aware* if for any walk $(s_0, v_0)\xrightarrow{*}(s_n, v_n)$ we have $v_n \preceq f(v_0)$

Uniform termination

Definition 127 ((Uniform termination))**.** Let G be an RSS. G is *uniformly terminating* if there is no infinite admissible walk over G.

Notice that if an RSS is not uniformly terminating, then there exists an infinite admissible walk that stays entirely within one strongly connected component of the underlying graph. In the following, when dealing with infinite walks we suppose without loss of generality that the RSS is strongly connected.

Theorem 106 can be generalised to RSS:

Theorem 128. *If G does* not *uniformly terminate, then there is an admissible cycle $(s, v)\xrightarrow{+}(s, u)$ with $v \preceq u$. If G is increasing and positive, then this is an equivalence.*

Proof. If an infinite admissible walk exists, then we can extract from it an infinite sequence of admissible configurations (s', v_k) with fixed s', since there is only a finite number of vertices. Since the order is a well partial order on V^+, there exists a $i < j$ with $v_i \preceq v_j$, thus leading to an admissible cycle.

If such a cycle exists, then it is sufficient to follow it infinitely many times to have an infinite admissible walk. Monotonicity is needed to ensure that every time one follows the cycle, the valuation does indeed not decrease. Positivity is needed to ensure that when going through never decreasing valuations one does not leave V^+. ☐

Proposition 129. *Let $G = (G, V, V^+, W, \omega)$ be an RSS.*

1. *If V is finite, then W is finite.*
2. *If V is finite, then uniform termination of G is decidable.*
3. *If both V and W are enumerable, then uniform termination for an ordered RSS G is semi-decidable.*

Proof.

1. The set of functions $\mathcal{F}(V, V)$ is finite and contains W.
2. If there are only finitely many valuations, any infinite walk eventually comes back to exactly the same configuration, hence the cycle of Theorem 128 becomes $(s, v)\xrightarrow{+}(s, v)$. Then it is possible to compute all the possible weights of cycles (there are only finitely many of them) and check with each valuation whether the condition is met. Notice that this does not require the RSS to be ordered.
3. By enumerating the cycles and the valuations simultaneously, computing the new valuation after going through the cycle and checking with the ordering whether this satisfies Theorem 128.

☐

Corollary 112 can be generalised:

Proposition 130. *If $(W, ⨾)$ is commutative, then the set of weights of cycles of an RSS is semi-linear.*

This allows us to easily find candidates for a generalisation of Theorem 114 if the set of "positive" weights is easily expressible (as in the case of VASS). Among other properties: if it is itself semi-linear, then uniform termination is decidable (because intersection between two semi-linear parts is decidable).

6.3 Equational versus constraint based approach

Up to now, the only weights we have considered are functions, meaning that if $s\xrightarrow{a}s'$, for each valuation v there is only one valuation v' such that $(s, v)\xrightarrow{a}(s', v')$. Sometimes, it is more convenient to have several possible results because approximations of the values leads to a loss of information. In this case, the weights considered are relations rather than functions and we require $v' \in \widehat{\omega}(a)(v)$ rather than $v' = \omega(a)(v)$.

Constraints RSS

Definition 131 ((Constraints RSS, configurations, walks))**.**
A *Constraints RSS* is a tuple (G, V, V^+, W, ω) where

- $G = (S, A)$ is a directed graph.
- $V_i^+ \subset V_i$ are, respectively, the sets of *admissible valuations* and *valuations*.
- $W_{i,j} : V_i \to \mathcal{P}(V_j)$ are the sets of *weights*.
- $\widehat{\omega} : A \to W$ is the *weight function* such that $\widehat{\omega}(a) \in W_{i,j}$ if $s_i \xrightarrow{a} s_j$.

Configurations and admissible configurations are defined as earlier.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0) \xrightarrow{a_1} \dots \xrightarrow{a_n} (s_n, v_n) \xrightarrow{a_{n+1}} \dots$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n \xrightarrow{a_{n+1}} \dots$ and for all $i > 0$, $v_i \in \widehat{\omega}(a_i)(v_{i-1})$. A walk is *admissible* if all configurations in it are admissible.

It is important to notice that even if weight functions return sets (that is, they are relations rather than functions), each step of a walk has to choose one element from this set as a new valuation. That is, we do not consider configurations with sets as valuations, but rather introduce some kind of non-determinism in the RSS. The main use for this is when some valuations are in no way related to the previous ones and can be anything (*e.g.* if a value is provided via some external mechanism such as a `scanf` instruction).

Definition 132 ((Weight of a path))**.** Let G be an RSS. The weight function can be canonically extended over all paths in G by choosing $\widehat{\omega}(ab)(x) = \bigcup_{y \in \widehat{\omega}(a)(x)} \widehat{\omega}(b)(y)$.

As earlier, uniform termination means that there exists no infinite admissible walk. However, monotonicity is now expressed: $x \preceq y \Rightarrow \forall x' \in \widehat{\omega}(x), \exists y' \in \widehat{\omega}(y)/x' \preceq y'$.

Then, Theorem 128 becomes:

Theorem 133. *Let G be a positive increasing Constraints RSS. G is* not *uniformly terminating if and only if there is an admissible cycle $(s, v_0) \xrightarrow{+} (s, v_1)$ such that $v_0 \preceq v_1$.*

Proof. If an admissible infinite walk exists, then we can extract from it an admissible cycle in exactly the same way as with Theorem 128.

Conversely, if a non-decreasing admissible cycle c exists, let $(s, v_0) \xrightarrow{a} (s', v_0') \xrightarrow{*} (s, v_1)$ be the first, second and last configurations when following the cycle. By hypothesis, $v_0 \preceq v_1$.

Then, there exists $v_1' \in \widehat{\omega}(a)(v_1)$ such that $(s, v_1) \xrightarrow{a} (s', v_1')$ and $v_0' \preceq v_1'$. By positivity of the VASS, v_1' is still admissible.

By iterating this process, we build an admissible cycle $(s, v_1) \xrightarrow{c} (s, v_2)$ with $v_1 \preceq v_2$. Then, this can be done *ad infinitum* thus leading to an admissible infinite walk. ☐

Constraints VASS

Let us show how this concept applies to VASS and why it can be useful when studying programs. Remember that $\overline{\mathbb{Z}} = \mathbb{Z} \bigcup \{+\infty\}$.

Definition 134 ((Constraints VASS))**.** A *Constraints VASS* is a directed graph $G = (S, A)$ together with a *weight function* $\omega : A \to \overline{\mathbb{Z}}^k$ where k is a fixed integer.

A *configuration* is a pair $\eta = (s, v)$ where $s \in S$ and $v \in \mathbb{Z}^k$. It is *admissible* if $v \in \mathbb{N}^k$.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0) \xrightarrow{a_1} \dots \xrightarrow{a_n} (s_n, v_n)$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ and for all $i > 0$, $v_i \leq v_{i-1} + \omega(a_i)$. A walk is *admissible* if all configurations in it are admissible.

To express a Constraints VASS as a Constraints RSS, we should consider the weight function $\widehat{\omega}(a) : \mathbb{Z}^k \to \mathcal{P}(\mathbb{Z}^k)$ such that $\widehat{\omega}(a)(v) = \{v' | v' \leq v + \omega(a)\}$. Then, the relation between valuations in a walk is the general $v_i \in \widehat{\omega}(a_i)(v_{i-1})$. Since all constraints have the same shape, we can express this in a more readable way. Constraints VASS are positive and increasing. When there is no $+\infty$ in the weights, it is always “best” to choose the greatest possible valuation, that is, use the (regular) VASS with the same underlying graph and weight function.

Example 135. Consider the following functional program computing Ackermann’s function:

$$\begin{aligned}
\texttt{Ack}(0, n) &\to n + 1 \\
\texttt{Ack}(m + 1, 0) &\to \texttt{Ack}(m, 1) \\
\texttt{Ack}(m + 1, n + 1) &\to \texttt{Ack}(m, \texttt{Ack}(m + 1, n))
\end{aligned}$$

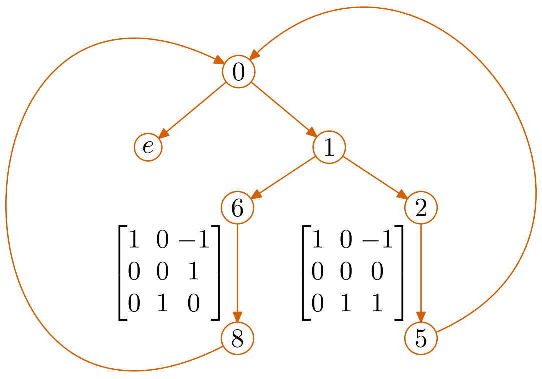

0
e
1
6
2
8
5
$\begin{bmatrix}1&0&-1\\0&0&1\\0&1&0\end{bmatrix}$
$\begin{bmatrix}1&0&-1\\0&0&0\\0&1&1\end{bmatrix}$

Figure 9: Constraints VASS for Ackermann's function

For functional programs, an equivalent of the CFG is the *call graph*. There is one vertex for each function symbol (here only one) and one edge for each call (here, 3). Since there are two positive integers in the program, it is natural to choose (m, n) as valuation.

The first line does not perform any call, hence there is no edge corresponding to it in the graph (since termination is studied here, the first line can never lead to non-termination, hence it is safe to have nothing corresponding to it in the graph).

The second line performs one call where the arguments of the function go from $(m+1, 0)$ to $(m, 1)$, this corresponds to adding $(-1, 1)$ to the valuation.

The third line performs two calls. The inner call is from $\texttt{Ack}(m + 1, n + 1)$ to $\texttt{Ack}(m + 1, n)$ (embedded in some context). That is, in this call, the arguments of the function go from $(m+1, n+1)$ to $(m+1, n)$, so the corresponding edge is labelled $(0, -1)$.

However, when considering the outer call in the last line the second argument becomes $\texttt{Ack}(m+1, n)$ which cannot be related to the parameter n in any easy way. So, using a regular VASS, this call would not be representable.

With a Constraints VASS, we can represent this last call. Indeed, not knowing anything on the result simply means that we can relax all constraints on it which is represented by the vector $(-1, +\infty)$. The constraints VASS for Ackermann's function is displayed on Figure 9.

Since this Constraints VASS is uniformly terminating, so is Ackermann's function.

This example shows that Constraints VASS are useful and that we can apply the ideas behind RCGs to functional programs.

7 Resource Control Graphs

Instead of the weighted graphs or VASS used before, we now use any RSS to model programs. A set of admissible valuations is given to each state and weight functions simulate the corresponding instruction.

Since we can now have any approximation of the memory (the stores) for valuations, we cannot simply use the length of a state to abstract it. Instead, we consider given a *size function* that associates to each state (or to each store) some size. The size function is unspecified in general. Of course, when using RCG to model programs, the first thing to do is usually to determine a suitable size function (according to the studied property). Notice that depending on the size function, weights of instructions can or cannot be defined properly (that is, some sizes are either too restrictive or too loose and no function can accurately reproduce on the size the effect of a given instruction on actual data). In this case, the RCG cannot be defined and another size function has to be considered.

7.1 Resource Control Graphs

Definition 136 ((RCG))**.** Let p be a program and G be its control flow graph. Let V^+ be a set of admissible valuations (and $\prec$ be a well partial order on it). Let $||\bullet|| : \Theta \to V^+$ be a size function from states to valuations and $V^+_{\texttt{lbl}}$ be the image by $||\bullet||$ of all states $\langle \texttt{lbl}, \sigma\rangle$ for all stores σ.

For each edge $\texttt{i}$ of G, let $\omega(\texttt{i})$ be a function such that for all states θ, $p \vdash \theta \xrightarrow{\texttt{i}} \theta'$ implies $\omega(\texttt{i})(||\theta||) = ||\theta'||$. Let V be the closure of V^+ over all the weight functions $\omega(\texttt{i})$.

The *Resource Control Graph* (RCG) of p is the RSS built on G with weights $\omega(i)$ for each edge i, valuations V and admissible valuations V^+ (ordered by $\prec$). $V^+_{\texttt{lbl}}$ being the admissible valuations for vertex $\texttt{lbl}$.

As stated before, we write $v \circledast \omega(\texttt{i})$ instead of $\omega(\texttt{i})(v)$ and $\omega(\texttt{i}) \,\fatsemi\, \omega(\texttt{j})$ instead of $\omega(\texttt{j}) \circ \omega(\texttt{i})$.

Lemma 137. *Let p be a program, G be its RCG and $p \vdash \theta_0 \to \ldots \to \theta_n$ be an execution with trace t. There exists an admissible walk $(s_0, ||\theta_0||) \to \ldots \to (s_n, ||\theta_n||)$ with the same trace t.*

Theorem 138. *Let p be a program and G be its RCG. If G is uniformly terminating, then p is also uniformly terminating.*

Example 139. A Space-RCG as defined in Section 4 is a special case of general RCG. In this case, $||\theta|| = |\theta|$, which leads to $V^+_{\mathtt{lbl}} = V^+ = \mathbb{N}$ for each label `lbl`. Similarly, $\omega(\mathtt{i}) = \lambda x.x + k_{\mathtt{i}}$ with $k_{\mathtt{i}}$ as in Definition 73. Since $k \in \mathbb{Z}$, the closure of V^+ over the weight functions is $V = \mathbb{Z}$.

In this case, resource awareness of the Space-RCG (or β-Space-RCG) guarantees a resource bound on the program execution.

Example 140. For a better representation of programs, the size can be the vector where each component is the length of a stack: $||\langle \mathtt{IP}, \sigma\rangle|| = (|\mathbf{stk}_1|, \cdots, |\mathbf{stk}_s|)_{\mathbf{stk}_i \in \mathcal{S}}$. This corresponds exactly to what is done with the Resource Control VASS of Section 5.3. As shown, this allows to decide uniform termination of several programs.

This termination analysis is close to the Size Change Termination [LJBA01] in the sense that the size of data is monitored and a well ordering on it ensures that it cannot decrease forever. It is sufficient to prove uniform termination of most common lists programs such as reversing a list or insertion sort. It also accepts some programs that are not handled by the original SCT, because it can take into account not only size decreases, but also increases. In this way, a program that would loop on something like `pop pop push` (2 `pop`s and 1 `push`) is not caught by SCT but is proved uniformly terminating with this analysis. In this sense, it is closer to the SCT with difference constraints (δSCT) [BA08].

This method is in NPTIME since, as we have shown, uniform termination of VASS is in NPTIME. The original SCT, as well as fan-in free δSCT, is PSPACE-complete. However, this simple method does not allow for data duplication or copy. Lee, Jones and Ben-Amram already claimed in the original paper on SCT that there exists a poly-time algorithm for SCT dealing with “programs whose size-change graphs have in- and out-degrees bounded by 1”. It is easy to check that VASS can only model such kind of programs accurately[9], hence the NP bound is not a big surprise.

Moreover, this method has a fixed definition of size and hence does not detect termination of programs whose termination argument does not depend on the decrease of the length of a list. Among others, any program working solely on integers (represented as letters of the alphabet) is not analysed correctly.

Example 141. This representation can be improved. Typically, using Resource Control VASS, it is impossible to detect anything happening to registers. If we have a suitable size function $||\bullet|| : \Sigma \to \mathbb{N}$ for registers[10], we can choose $||\langle \mathtt{IP}, \sigma\rangle|| = (||\mathbf{r}_1||, \cdots, ||\mathbf{r}_r||)_{\mathbf{r}_i \in \mathcal{R}}$. In this case, depending on the operators, weight could be either vector addition or matrix multiplication (to allow the copy of a register).

Remark 142. Taking exactly the image of $||\bullet||$ as the set of admissible valuations V^+ might be a bit too difficult. Indeed, this set might have any shape and is probably not really easy to handle. So, it is sometimes more convenient to consider a superset of it in order to easily decide whether a valuation is admissible or not. The convex hull (in V) of the image of $||\bullet||$ is a typical example of such a superset. Notice that it is very similar to the idea of trying to find an admissible set of sequences of states which is more manageable than the set of executions. Here, we try to find an admissible set of valuations which is more manageable than the actual set of sizes. For more details on how to build and manage such a superset, see the work of [Ave06].

Remark 143. The size function is not fixed and may depend on the property one wants to study, the program being analysed or even each particular program point, as can be seen in the coming example. We do not address here the problem of finding a suitable size function for a given program. As hinted, it might be a simple vector of functions over stacks and registers but it can also be a more complicated function such as a linear combination or so. Hence, with a proper size function, one is able not only to check that a given register (seen as an integer) is always positive but also that a given register is always bigger that another one. This is similar to Avery's functional inequalities [Ave06].

Obviously, the problem of finding a suitable size function is undecidable (for each terminating programming, there exists a suitable size function). However, many programs can be analysed with a uniform family of similar size functions. hence, it is interesting to have a tool to automatically infer good size function from programs, such as convex-hull analysis.

Example 144. Let us consider the following program, working on integers (that is, the alphabet is the set of 32 bits

[9] And cannot even model all those programs due to the restriction on copying variables.

[10] Note that the *size* function used here is in no way related to the *length* of a state. It plays no role when computing the space usage of a state and may simply be seen as an ordering over the alphabet.

positive integers[11] and overflow or underflow throws an exception):

```
  0 : $\mathbf{i} := 0$;                                          4 : if $\mathbf{i} < \mathbf{n}$ then goto 2;
  1 : if $regi \geq \mathbf{n}$ then goto 5;                      5 : $\mathbf{i} := \mathbf{i} + 1$;
  2 : $\mathbf{i} := \mathbf{i} + 1$;                           end : end;
  3 : some instructions modifying neither $\mathbf{i}$ nor $\mathbf{n}$
```

This is simply a loop `for(i=0;i<n;i++)` (in a `C`-like syntax). If we consider a size function that simply takes the vector of the registers, that is $||\langle \mathtt{IP}, \sigma\rangle|| = (\mathbf{i}, \mathbf{n})$, then the loop has weight $(+1, 0)$ and thus lead to a cycle of positive weight. However, a clever analysis of the program could detect that inside the loop we must necessarily have $n - i > 0$ and thus suggest the size $||\langle \mathtt{IP}, \sigma\rangle|| = \mathbf{n} - \mathbf{i}$. Using this, the loop has weight -1 and we can prove uniform termination of the program.

As stated, we do not address here the problem of finding a correct size function for a given program. This problem is undecidable in general. But invariants can often be automatically generated, usually by looking at the pre- and post-conditions of the loops.

Notice also that this inequality must hold only in the loop. Indeed, at label 5, we have $\mathbf{i} > \mathbf{n}$. Hence using this size function everywhere would cause troubles since then $||(5, \sigma)||$ is not admissible.

Having different sets of valuations for each labels, that is a size function operating differently on each label, can solve this problem. By choosing $||\langle \mathtt{IP}, \sigma\rangle|| = (\mathbf{i}, \mathbf{n})$ for $\mathtt{IP} = 0, 1, 5, \mathtt{end}$ and $||\langle \mathtt{IP}, \sigma\rangle|| = (\mathbf{i}, \mathbf{n}, \mathbf{n} - \mathbf{i})$ otherwise, we can ensure that the "natural" sets of admissible valuations ($\mathbb{N}^2$ and $\mathbb{N}^3$) indeed correspond to the image of the size function (or at least a manageable superset of it).

In this case, of course, we need the weight between labels 1 and 2 to take into account new components appearing in the valuation. Here, this can be done by using a matrix multiplication since the new component in the valuation is a linear combination of the existing ones. See Example 157 for the complete construction of the RCG.

7.2 Constraints RCG

Constraints RSS can also be used instead of RSS to model programs and build RCG as done with the Ackermann's function of Example 135. In that case, the relation required between weights and sizes is:

for all states θ verifying $p \vdash \theta \xrightarrow{\mathtt{i}} \theta'$, $||\theta'|| \in \widehat{\omega}(\mathtt{i})(||\theta||)$.

Then, the simulation Lemma and uniform termination Theorem are still true:

Lemma 145. *Let p be a program, G be its Constraints RCG and $p \vdash \theta_0 \to \ldots \to \theta_n$ be an execution with trace t. There exists an admissible walk $(s_0, ||\theta_0||) \to \ldots \to (s_n, ||\theta_n||)$ with the same trace t.*

Proof. Because $||\theta_i||$ belongs to $\widehat{\omega}(a)(\theta_{i-1})$ and can thus always be chosen as the new valuation. □

Theorem 146. *Let p be a program and G be its RCG. If G is uniformly terminating, then p is also uniformly terminating.* □

8 δ-Size Change Termination

In Section 4, we have used RCG in order to have an analysis of running space similar to the Non Size Increasing approach of Hofmann. In this section, we use RCG to analyse termination of programs in a way similar to the Size Change Termination of Lee, Jones and Ben-Amram and, more precisely, to the δ-Size Change Termination of Ben-Amram.

We consider here the $(\overline{\mathbb{Z}}, \min, +)$ semi-ring and denote min as $\oplus$ and $+$ as $\otimes$. These operations are canonically extended to define multiplication of matrices[12] from $\mathcal{M}(\overline{\mathbb{Z}})$.

8.1 Matrices and graphs

Definition 147 ((Constraint graph))**.** Let M be a square matrix of dimension n. Its *constraint graph* is a weighted directed graph G such that:

- There are n vertices $X_i, 1 \leq i \leq n$ plus an extra vertex Y.
- If $M_{i,j} \neq +\infty$, there is an edge of weight $M_{i,j}$ from X_i to X_j.
- There is an edge of weight 0 from Y to X_i, for all i.

[11] Corresponding to the `C` type `unsigned`.

[12] That is, given two matrices A and B, $(A \oplus B)_{i,j} = A_{i,j} \oplus B_{i,j} = \min(A_{i,j}, B_{i,j})$ and $(A \otimes B)_{i,j} = \bigoplus_k A_{i,k} \otimes B_{k,j} = \min_k(A_{i,k} + B_{k,j})$. Similarly, if X is a vector and M a matrix, then $(X \otimes M)_j = \bigoplus_k X_k \otimes M_{k,j} = \min_k(X_k + M_{k,j})$.

Definition 148 ((l-weight))**.** Let G be a directed weighted graph. The *l-weight from a to b* is the minimum weight of all paths of length l from a to b, and $+\infty$ if there is no such path.

The coefficient $M^k_{i,j}$ is the k-weight from X_i to X_j in the constraint graph of M.

Lemma 149. *The system $X \leq X \otimes M$ has a solution if and only if there is no strictly negative coefficient in the diagonal of M^k, for all k. In that case, it admits a non-negative solution.*

It is possible to decide in polynomial time whether such a system admits a solution.

Proof. The matrix inequality corresponds to the set of inequalities $\{X_j \leq \min_i(X_i + M_{i,j}) | j \leq n\}$ which can, without modifying the set of solutions, be expressed as $\{X_j \leq X_i + M_{i,j} | i, j \leq n\}$.

If for every $k > 0$ there is no strictly negative coefficient in the diagonal of M^k, the constraints graph G has no cycle of strictly negative weight. In this case, we can choose for X_i the weight of the shortest path from Y[13]. This is well defined because there is no cycle of strictly negative weight and provides a solution for the system because $X_j \leq X_i + M_{i,j}$ holds for all i, j by definition of shortest paths.

Conversely, if there is a path of strictly negative weight, then it is easy to see that by adding the inequalities corresponding to the edges in this path one eventually infers the contradiction $X_i < X_i$ and the system must have no solution.

If there is a solution X, then $X + (1, \dots, 1)$ is also a solution. Hence, there exists a solution where all values are non-negative.

The system admits a solution if and only if the constraint graph has no cycle of strictly negative weight. This can be decided in polynomial time by Bellman-Ford's algorithm. □

8.2 Size Change Termination

We explain here how to build RCG in order to perform the same kind of analysis as the Size-Change Termination with difference constraints (δSCT) of [BA08]. Here, we use matrices rather than Size Change Graphs following the work of [AA02] where similar SCT matrices are used (but over a 3-valued set, thus mimicking the initial SCT and not the work with difference constraints).

In this whole section, we consider a fixed program p, and for each label $\mathtt{lbl}_a$ in it a fixed integer k_a. Let $V_a = \mathbb{Z}^{k_a}$ and $V_a^+ = \mathbb{N}^{k_a}$ be sets of (admissible) valuations associated with each label. We consider a size function $|| \bullet ||$ such that for each label $\mathtt{lbl}_a$ and for each store σ, $||\langle \mathtt{lbl}_a, \sigma\rangle|| \in V_a^+$.

Definition 150 ((Size Change Matrix))**.** Let $\mathtt{i}$ be an instruction in p corresponding to an edge from $\mathtt{lbl}_a$ to $\mathtt{lbl}_b$ in G. The *Size Change Matrix* (SCT matrix) of $\mathtt{i}$ is a $k_a \times k_b$ matrix $M^{(\mathtt{i})}$ of $\mathcal{M}_{k_a,k_b}(\overline{\mathbb{Z}})$ such that for all states θ_a with $p \vdash \theta_a \xrightarrow{\mathtt{i}} \theta_b$, $||\theta_b|| \leq ||\theta_a|| \otimes M^{(\mathtt{i})}$.

If $||\theta_a|| = (x_1, \cdots, x_{k_a})$ and $||\theta_b|| = (y_1, \cdots, y_{k_b})$, we have for each j: $y_j \leq \min_k\{x_k + M^{(\mathtt{i})}_{k,j}\}$ where the coefficients of $M^{(\mathtt{i})}$ can be any integer or $+\infty$.

Definition 151 ((Size Change RCG))**.** The *Size Change RCG* (SCT-RCG) of p is the Constraints RCG for p built with admissible valuations $\mathbb{N}^{k_a}$, and valuations $\mathbb{Z}^{k_a}$ for vertex $\mathtt{lbl}_a$. The weight for any edge $\mathtt{i}$ is such that $\widehat{\omega}(\mathtt{i})(v) = \{v' | v' \leq v \otimes M^{(\mathtt{i})}\}$ where $M^{(\mathtt{i})}$ is the SCT matrix for $\mathtt{i}$.

As for Constraints VASS, the common shape of constraints allows to use a weight function $\omega(\mathtt{i}) = M^{(\mathtt{i})}$ instead of the weight relation $\widehat{\omega}$ and require along a walk that $v_i \leq \omega(a_i)(v_{i-1})$ rather than $v_i \in \widehat{\omega}(a_i)(v_{i-1})$.

The uniform termination Theorem for Constraints RCG (Theorem 146) tells us that if the SCT-RCG is uniformly terminating then so is p.

SCT-RCG are both increasing and positive, so it is possible to apply Theorem 133.

Theorem 152. *Let G be the SCT-RCG of p. It is uniformly terminating if and only if it does not contain a cycle c of weight $M^{(c)}$ such that $X \leq X \otimes M^{(c)}$ admits a solution.*

Proof. If the system $X \leq X \otimes M^{(c)}$ admits a solution, then it admits an arbitrarily large solution. Hence, there exists an admissible cycle $(s, X) \xrightarrow{c} (s, X \otimes M^{(c)})$ and by Theorem 133, the SCT-RCG is not uniformly terminating.

Conversely, if the SCT-RCG is not uniformly terminating then, by Theorem 133, there exists a cycle of weight M such that $X \leq X \otimes M$ has a solution. □

Remark 153. The reader familiar with the original works of [LJBA01] or [BA08] may wonder why there is no idempotence condition in Theorem 152. As a matter of fact, it happens that any square matrix M on the $(\overline{\mathbb{Z}}, \min, +)$

[13]We consider here 'shortest' path in term of 'lowest weight' path. This weight cannot be more than 0 –there is a 0-weighted edge from Y to all other vertices– but can be strictly negative.

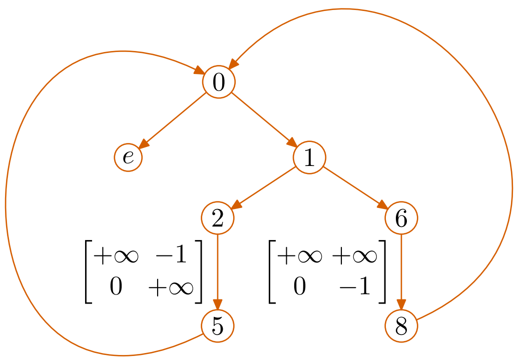

0
e
1
2
6
$\begin{bmatrix}+\infty & -1\\ 0 & +\infty\end{bmatrix}$
$\begin{bmatrix}+\infty & +\infty\\ 0 & -1\end{bmatrix}$
5
8

Figure 10: A Size Change Termination RCG.

semi-ring has a power $M' = M^k$ which is *strongly sign idempotent*, that is the coefficients $M'^n_{i,j}$, for all $n > 0$ all have the same sign [Moy08].

The matrices we use here, as well as the Size Change Graphs in the other works, represent the flow of data. The idea behind idempotence is that we want to detect a cycle in the program such that the corresponding flow of data is also circular, that is each variable flows to itself.

The dangerous cycles (with respect to termination) are those that (i) have an idempotent flow of data and (ii) do not have a decrease in one of the data. Indeed, these cycles could be repeated infinitely many time, leading to a potential infinite execution.

However, as stated for matrices at the beginning of the remark, each flow of data eventually becomes idempotent if repeated sufficiently many times. Hence, finding a cycle whose weight M is such that $X \leq X \otimes M$ admits a solution is sufficient to get a cycle with idempotent flow of data by repeating this cycle.

With RCG, the notion of valuation makes the inequality $X \leq X \otimes M$ pretty natural, since it exactly corresponds to what happens to valuations after going through the cycle. The original RCG works, however, does not have this notion of valuation but only the matrices (or graphs), seen as a description of the modification on the size of variables (independently to the actual values of the variables, that is the valuations). Hence, the idempotence condition was natural in this framework but the notion of RCG shows that one can actually get rid of it.

Notice that the flow of data is somewhat taken into account in Lemma 149 where we consider the sign of the coefficients of the diagonal of M^k. The coefficients on the diagonal of M^k describe how the data flows from x_i to itself after repeating the cycle k times.

By Lemma 149, the individual condition on cycles is decidable in polynomial time. The general condition, however, is undecidable. Nevertheless, if the matrices are *fan-in free*, that is in each column of each SCT matrix, there is at most one non-$+\infty$ coefficient, then the problem is PSPACE-complete. See [BA08] for details. Notice that in this paper, Ben-Amram uses mostly SCT graphs and not SCT matrices. The translation from one to the other is, however, quite obvious. Similarly we present here directly a condition on the cycles of the SCT-RCG without introducing the multipaths. This is close to the "graph algorithm" introduced in [LJBA01].

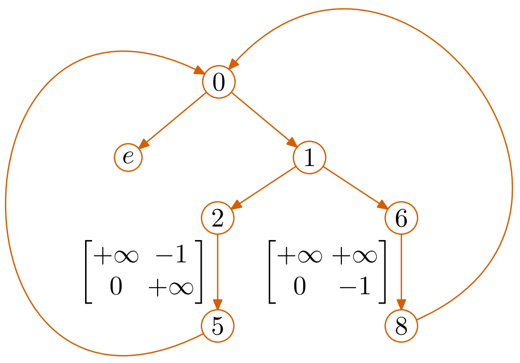
The simple Size Change Principle of [LJBA01] can be seen as an approximation of the δSCT principle where only labels in $\{-1, 0, +\infty\}$ are used. Since this only gives way to finitely many different SCT matrices, this is decidable in general (PSPACE-complete).

Example 154. Consider the following program (adapted from [LJBA01] fifth example):

```
  0  :  if y = 0 then goto end;   |    5  :  goto0;
  1  :  if x = 0 then goto 6;     |    6  :  x := y;
  2  :  a := x;                   |    7  :  y := y − 1;
  3  :  x := y;                   |    8  :  goto0;
  4  :  y := a − 1;               |  end  :  end;
```

It can be proved terminating by choosing the size function $||\theta|| = (\mathbf{x}, \mathbf{y}, \mathbf{a})$. With this size, its SCT-RCG is displayed on Figure 10. For reasons of convenience, instructions $2-4$, as well as $6-7$ are represented as a single edge (with a single matrix). This allows to completely forget the register $\mathbf{a}$ and so use $(\mathbf{x}, \mathbf{y})$ as size. Similarly, the other SCT matrices are not depicted since they are the identity matrix. Because the SCT-RCG is uniformly terminating, so is the program.

When working with this simple Size Change Principle (or any other restriction where there can be only finitely many different weights), Theorem 152 gives an algorithmic way of detecting uniform termination of the SCT-RCG. Indeed, when there are only finitely many different weights, there are only finitely many tuples (s, M, r) such that



0
e
1
2
6
5
8
$\begin{bmatrix} +\infty & -1 \\ 0 & +\infty \end{bmatrix}$
$\begin{bmatrix} +\infty & +\infty \\ 0 & -1 \end{bmatrix}$

Figure 11: MMSS as a RCG.

there exists a path from s to r whose weight is M. Then, it is possible to build all these tuples in an incremental way by starting with tuples (s, M, r) corresponding to each edge of the SCT-RCG and adding new tuples by composing existing ones with matching edges. This is the core idea of the "graph algorithm" of [LJBA01].

9 More on matrices

9.1 Matrix Multiplication System with States

If we use vectors as valuations and (usual) matrix multiplication as weights, we can define Matrix Multiplication Systems with States (MMSS) in a way similar to VASS. Admissible valuations are still the ones in $\mathbb{N}^k$ but k is not fixed for the RSS and may depend on the current vertex.

Definition 155 ((Matrix Multiplication System with States))**.** A *Matrix Multiplication System with States* (MMSS) is an RSS $G = (G, V, V^+, W, \omega)$ where:

- $V_i = \mathbb{Z}^{k_i}$, $V_i^+ = \mathbb{N}^{k_i}$ for some constant k_i (depending on the vertex s_i).
- Weights are matrices with integer coefficients.
- $\mathbin{;} = \circledast = \times$.

Using this, it is quite easy to model copy instructions of counters machines ($\mathbf{x} := \mathbf{y}$) simply by using the correct permutation matrix as a weight. To represent increment or decrement of a counter, an operation which was quite natural with VASS, we now need a small trick known as *homogeneous coordinates*[14]. One can simply represent the n counters as an $n+1$ component vector whose first component is always 1. Then, incrementing or decrementing a variable becomes a linear combination of components of the vector which can perfectly be done with matrix multiplication. For example, here is how one can model the copy ($\mathbf{x} := \mathbf{y}$) and the increment ($\mathbf{x} := \mathbf{x} + 1$).

$$(1, x, y) \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} = (1, y, y) \qquad (1, x, y) \times \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (1, x+1, y)$$

Example 156. Using homogeneous coordinates, the program of Example 154 has the MMSS depicted on Figure 11. Here, matrix multiplication is done on the usual $(\mathbb{Z}, +, \times)$ ring and not on the $(\overline{\mathbb{Z}}, \min, +)$ semi-ring as for SCT-RCG.

Example 157. Similarly, use of homogeneous coordinates allows to build an MMSS to prove uniform termination of the program of Example 144. It is depicted on the left part of Figure 12 (where label 3 has been omitted). The interesting thing here is the use of vectors of different lengths at different labels, thus allowing to add the constraint $\mathbf{n} - \mathbf{i} \geq 0$ only inside the loop. This example shows both the use of disjoint sets of valuations and how to work with the functional inequalities of [Ave06].

But there is even more. VASS are able to forbid a $x \neq 0$ branch of a test being taken in an admissible walk if x is 0 simply by decrementing x and then incrementing it immediately after. The net effect is null but if x is 0, the intermediate valuation is not admissible. This can still be done with MMSS. VASS, like Petri nets, are however not able to test if a component is empty, that is forbid the $x = 0$ branch of a test to be taken if x is not 0.

[14]Homogeneous coordinates were originally introduced by A. F. Möbius. They are used, for instance, in computer graphics for exactly the same purposes as we do here: representing a translation by a matrix multiplication.

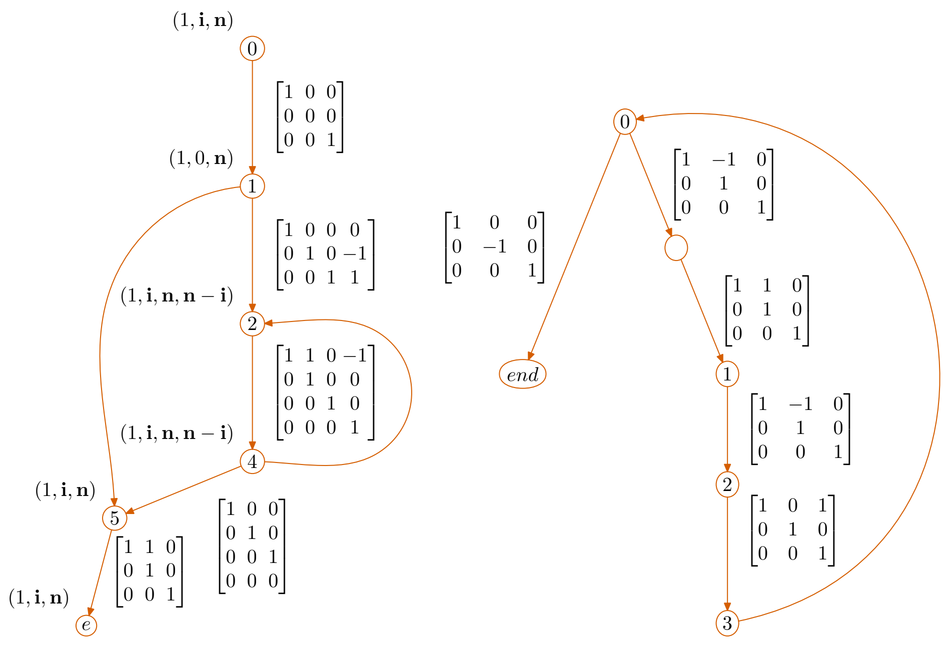


Figure 12: MMSS for loop and unary addition.

With MMSS, we can perform this test against 0. It is indeed sufficient to multiply the correct component of the valuation by -1. If it is different from 0, then the resulting valuation is not admissible.

So, using these tricks it is possible to perfectly model a counters machine by an MMSS: each execution of the machine corresponds to exactly one admissible walk in the MMSS and each admissible walk in the MMSS corresponds to exactly one execution of the machine.

This leads to the following theorem:

Theorem 158. *Uniform termination of MMSS is not decidable.* □

Example 159. Consider the following program, performing addition in unary (that is, repeatedly decrementing $\mathbf{x}$ while incrementing $\mathbf{y}$ until $\mathbf{x}$ is 0).

```
  0 : if x = 0 then goto end;  |    3 : goto 0;
  1 : x := x − 1;              |  end : end;
  2 : y := y + 1;              |
```

Right side of Figure 12 depicts an MMSS for this program such that there is a one-to-one correspondence between executions of the program and admissible walks of the MMSS. The size used is $(1, \mathbf{x}, \mathbf{y})$, the 1 appearing because of homogeneous coordinates. Notice that we need to add an intermediate label for the $\mathbf{x} \neq 0$ branch of the test in order to generate the temporary valuation containing $\mathbf{x} - 1$, only used to force admissible walks with $\mathbf{x} = 0$ to take the other branch.

On the other branch of the test, the -1 in the center of the matrix ensures that if $x > 0$ in the valuation at vertex 0, then following this edge leads to a non-admissible valuation. That is, this edge can only be followed if $x = 0$.

Since such a construction can be done for any counter machine (the unary addition program uses all possible instructions for counter machines) and since counter machines are Turing-complete, this shows why uniform termination of MMSS is not decidable in general.

This simulation of programs by matrix multiplications raises a surprising question. Indeed, matrix multiplications are only able to perform linear operations on data, while some programs can obviously perform non-linear operations.

This apparent contradiction is solved when we think more closely on how RSS work. Each walk in an MMSS corresponds to a matrix multiplication (because ω is a morphism), hence to a linear transformation on data. However, two different walks give rise to two different matrices, hence two different linear transformations.

When simulating a program, each different input results in following a different (admissible) walk in the MMSS. Hence, each different input value is subject to a different linear transformation. Of course, the other walks (that is, the other linear transformations) also exist and are considered for this datum when looking at the set of walks, but non-admissibility allows to dismiss them and only keep one.

So, from a transformation point of view, we can look at MMSS as a set of linear transformations for which the admissibility mechanism selects the proper transformation to apply on each piece of data.

For example, if we consider a program performing multiplication of two integers x and y, it is likely a loop on x, adding y to the result each time. The corresponding MMSS has multiple paths (infinitely many) that can each be a candidate for a walk once actual input data is provided. Different paths correspond to following the loop $1, 2, 3, \ldots, k, \ldots$ times. Then, the walk corresponding to each of these paths performs the linear transformation $(1, x, y) \mapsto (1, x - k, ky)$ representable by the matrix:

$$\begin{bmatrix} 1 & -k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & k \end{bmatrix}$$

However, when performing all these transformations on actual data, only those with $k \leq x$ have an admissible result and only the one with $k = x$ has all its intermediate valuations admissible. So, the admissibility mechanism selects the right linear transformation to apply.

When simulating a program computing a (non-linear) function by an MMSS, the simulation actually considers the function as being *piecewise linear*, computes the result of all the possible linear transformations implied and selects the one corresponding to current data. In general, it is possible that each linear transformation is only valid for a single value.

9.2 Tensors

Moreover, the study can go further. Indeed, using matrices of matrices (that is, tensors), we can represent the adjacency graph of an MMSS (a matrix where component (i, j) is the coefficient of the edge from vertices i to j). That is, a first order program can be represented as such a tensor. However, it may then be possible to uses these tensors (and tensor multiplication) in order to study second-order programs. In turn, the second order programs would probably be representable by a tensor (with more dimensions) and so on.

This could lead to a tensor algebra representing high order programs.

Example 160. Here is a tensor representing the MMSS of the unary addition (as depicted in Figure 12). This is simply the connectivity matrix of the graph where each edge is itself weighted by a matrix.

$$\begin{bmatrix}
0 & \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & 0 & 0 & 0 & \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
0 & 0 & \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & 0 & 0 & 0 \\
0 & 0 & 0 & \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & 0 & 0 \\
0 & 0 & 0 & 0 & \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & 0 \\
\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

9.3 Polynomial time

Another interesting approach of program analysis using matrices is the one done by [NW06] and [KJ09]. The programs they study are similar to our stack machines except that the (conditional) jump is replaced by a fixed iteration structure (`loop`) where the number of iterations is bounded by the length of a given stack.

Then, they assign to each basic instruction a matrix, called a *certificate* which contains information on how to polynomially bound the size of the registers (or stacks) after the instruction by their size before executing the instruction. When sequencing instructions, the certificate for the sequence turns out to be the product of the certificates for each instruction. A certificate for a loop is some kind of multiplicative closure of the certificate for the body, and a certificate for an `if`-statement is the least upper bound of the two branches.

If a certificate for the program exists, then the result of the program is polynomially bounded by the inputs. This bound on size can then be turned into a bound on the running time, given the shape of the loops.

So, these certificates can very well be expressed in an MMSS where the valuation would give information on the size of registers (depending on the size of the inputs of the program) and the weights of instructions are these certificates. This is exactly a Resources Control Graph for the program. If the program is certified, then this RCG is polynomially resource aware.

10 Conclusion

We have introduced a new generic framework for studying programs. This framework is highly adaptable via the size function and can thus study several properties of programs with the same global tool. Analyses apparently different such as the study of Non Size Increasing programs or the Size Change Termination can quite naturally be expressed in terms of Resource Control Graphs, thus showing the adaptability of the tool.

Moreover, other analyses look like they can also be expressed in this way, thus giving hopes for a very generic tool to express and study programs properties such as termination or complexity. It is even likely that higher order programs could be studied that way, thus giving insights for a better understanding of higher order complexity.

The theory of algorithms is still not well established. This work is really on the study of programs and not of functions. Further works in this direction will shed some light on the very nature of algorithms and hopefully give one day rise to a theoretical framework as solid as our knowledge of functions. Here, the study of MMSS and the tensors multiplication suggests that a tensors algebra might be used as a mathematical background for a theory of algorithms.

Acknowledgements

Many thanks to A. Ben-Amram for pointing out critical flaws in an earlier version of the proof of Theorem 114. Thanks also to M. Hofmann for pointing out the name “homogeneous coordinates” and its use in computer graphics.

Many thanks also to the anonymous referees whose detailed comments greatly helped improving the presentation of the results.

On quasi-interpretations, blind abstractions and implicit complexity

Patrick Baillot, Ugo dal Lago, Jean-Yves Moyen

Mathematical Structures in Computer Science 22(4), 2012

Abstract:

Quasi-interpretations are a technique for guaranteeing complexity bounds on first-order functional programs: in particular, with termination orderings, they give a sufficient condition for a program to be executable in polynomial time [MM00], which we call the P-criterion here. We study properties of the programs satisfying the P-criterion in order to improve the understanding of its intensional expressive power. Given a program, its blind abstraction is the non-deterministic program obtained by replacing all constructors with the same arity by a single one. A program is blindly polytime if its blind abstraction terminates in polynomial time. We show that all programs satisfying a variant of the P-criterion are in fact blindly polytime. Then we give two extensions of the P-criterion: one relaxing the termination ordering condition and the other (the bounded-value property) giving a necessary and sufficient condition for a program to be polynomial time executable, with memoisation

1 Introduction

Implicit computational complexity

Implicit computational complexity (ICC) explores machine-free characterisations of complexity classes, without referring to explicit resource bounds but instead viewing them as consequences of restrictions on program structures. For the most part, ICC has been developed in the functional programming paradigm by taking advantage of ideas from primitive recursion [BC92, Lei94], proof-theory and linear logic [Gir98], rewriting systems and functional programming [Jon99, BCMT01, BMM01, Mar03, BMM11], type systems [LM93b, Hof99], and so on.

ICC results usually include both a soundness and a completeness statement: the former saying that all programs of a given language (or those satisfying a criterion) satisfy a certain quantitative property; the latter saying that all *functions* (or *problems*) of the corresponding functional complexity class can be programmed in this language. For instance, in the case of polynomial time complexity, soundness refers to the possibility of evaluating programs in polynomial time, whereas completeness, which has an *extensional* nature, refers to the class FP of functions computable in polynomial time (or the class PTIME of problems solvable in polynomial time). Theorems of this kind have been given for many systems, including ramified recursion [BC92, Lei94], variants of linear logic [Gir98] and fragments of functional languages [Jon99]. Our main aim here is *not* just to define another characterisation of polynomial time computable functions, but rather to begin a study of *existing* characterisations in order to understand their properties better – intensional expressivity *in primis*.

Expressivity

One of the main motivations for ICC comes from programming language theory, because ICC suggests ways to control the complexity properties of programs, which is a difficult issue because of its infinite nature. However, extensional correspondence with complexity classes is usually not enough: a programming language (or a static analysis methodology for it) offering guarantees in terms of program safety is plausible only if it captures enough interesting and natural *algorithms*. In turn, this cannot be guaranteed by merely requiring that the system corresponds extensionally to a complexity class. This issue has been pointed out by several authors [MM00, Hof99, Mar03], and advances have been made in the direction of more liberal ICC systems: some examples are type systems for *non-size-increasing* computation [Hof99] and quasi-interpretations [BCMT01, BMM11]. However, it is not always easy to measure the improvements offered by a new ICC system, which are usually just illustrated by providing examples, though some experiments [AM08] have been done to compare some rewriting-based criteria on data bases of examples from term rewriting.

We think that in order to compare in a more appropriate way the algorithmic aspects of ICC systems and, in particular, to understand their limitations, specific analytic methods should be developed. Indeed, what we need are *sharp results* on the *intensional* expressive power of existing systems and on the intrinsic limits of implicit complexity as a way to isolate large (but decidable) classes of programs with bounded complexity. To do this, we aim to establish properties (like necessary conditions) of the *programs* captured by an ICC system. Such a characterisation is provided in [DL07], where Cobham's definitions by bounded recursion on notation are shown to correspond exactly to hereditarily polytime primitive recursive programs. In the current paper, we undertake an analogous study, but for an ICC system closer to programming practice. More specifically, we give a sufficient condition that all programs in some existing ICC systems must satisfy.

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 us to define various criteria to guarantee either space or time complexity bounds [BMM05, Ama05, BMP07, BMM11]. QIs offer several advantages: in particular, the language for which they are defined is simple to use, and, more importantly, the class of programs captured by this approach is large compared to that of other ICC systems. For instance, this class contains all primitive recursive programs from Bellantoni and Cook's function algebra [Moy03], but also general recursive programs (non primitive recursive). One of the proposed criteria, which we will call the *P-criterion* here, says that programs with certain QIs and recursive path orderings can be evaluated in polynomial time [BMM11]. A key point when proving that programs satisfying the P-criterion can be evaluated in polynomial time is the adoption of a caching mechanism, following [Mar03].

In this paper, we focus our attention on QIs, and prove a strong necessary condition for first-order functional programs to have a (uniform) QI. More precisely, we present a program transformation called *blind abstraction*, which collapses the constructors of a given arity to a single constructor, and modifies the rewriting rules accordingly. In general, this produces non-confluent programs, the efficiency of which can be measured by considering all possible evaluations of the program.

$$\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow v_i}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(v_1, \cdots, v_n)} \text{(Constructor)} \qquad \frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad t_i \downarrow v_i \quad \mathtt{f}(v_1, \cdots, v_n) \downarrow v}{\mathtt{f}(t_1, \cdots, t_n) \downarrow v} \text{(Split)}$$

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad r\sigma \downarrow v}{\mathtt{f}(v_1, \cdots, v_n) \downarrow v} \text{(Function)}$$

Figure 1: Call by value semantics with of a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$

The blind abstraction of a polytime first-order functional program is not itself polytime in general: blinding introduces many execution paths that are not available in the original program. However, we show that under certain assumptions, blinding a program satisfying the P-criterion with a (uniform) QI always produces a polytime program, independently of non-confluence.

We cannot claim this result to be a breakthrough, but it does represent one of the first attempts to delineate the class of programs captured by a mainstream ICC system. Indeed, it implies that any polynomial time program whose blind abstraction is *not* polytime *cannot* have a quasi-interpretation. As we will show in Section 4, there are many natural programs in this class.

Outline of the paper

We begin in Section 2 by describing the syntax and operational semantics of programs, followed by termination orderings and QIs in Section 3. We introduce blind abstractions in Section 4 and give the main property of the P-criterion (with respect to blinding) in Section 5, with applications to safe recursion. In Section 3.4, we consider a particular class of QIs, the compatible QIs, and study their properties. Finally, in Section 7, we define a generalisation of the previous termination ordering and of QIs (the bounded-values property), which also guarantees polytime soundness.

A preliminary version of this work was presented at the Eighth International Workshop on Logic and Computational Complexity in 2006 [BdLM06].

2 Programs as term rewriting systems

In Sections 2 and 3 we will recall some definitions and results from [BMM11], and adapt them to non-deterministic programs.

We should stress that we are actually interested in the study of deterministic programs, but as soon as we consider abstractions of programs, we will obtain non-deterministic rewriting systems, and, in order to analyse them, we will need our definitions and results to handle non-deterministic systems.

2.1 The syntax and semantics of programs

We consider first-order term rewriting systems (TRS) with disjoint sets $\mathcal{X}$, $\mathcal{F}$ and $\mathcal{C}$ of variables, function symbols and constructor symbols, respectively.

Definition 161 (Syntax)**.** The sets of terms and equations are defined by:

$$\begin{array}{lrcl}
\text{(constructor terms)} & \mathcal{T}(\mathcal{C}) \ni v & ::= & \mathbf{c}(v_1, \cdots, v_n) \\
\text{(terms)} & \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t & ::= & x \mid \mathbf{c}(t_1, \cdots, t_n) \mid \mathtt{f}(t_1, \cdots, t_n) \\
\text{(patterns)} & \mathcal{P} \ni p & ::= & x \mid \mathbf{c}(p_1, \cdots, p_n) \\
\text{(equations)} & \mathcal{D} \ni d & ::= & \mathtt{f}(p_1, \cdots, p_n) \to t
\end{array}$$

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$ and $\mathbf{c} \in \mathcal{C}$. We shall use a type writer font for function symbols and a bold face font for constructors.

Definition 162 (Programs)**.** A program is a tuple $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ where $\mathcal{E}$ is a set of equations in $\mathcal{D}$. Each variable in the right-hand side of an equation also appears in the left-hand side of the same equation. The program has a main function symbol in $\mathcal{F}$, which we shall also call $\mathtt{f}$.

In other words, our programs are constructor (term-rewriting) systems – see the *Examples of TRSs and special rewriting formats* chapter in [KdV03].

The domain of computation is the constructor algebra $\mathcal{T}(\mathcal{C})$. A substitution σ is a mapping from variables to terms. We use $\mathfrak{S}$ to denote the set of *constructor substitutions*, that is, substitutions σ with range $\mathcal{T}(\mathcal{C})$. Our programs are not necessarily deterministic, in other words, the TRS is not necessarily confluent. Non-confluent programs will be interpreted here by relations.

$$\dfrac{\dfrac{\dfrac{\mathbf{nil}\downarrow\mathbf{nil}}{\mathtt{f}(\mathbf{s}_1\mathbf{nil})\downarrow\mathbf{nil}} \quad \dfrac{\mathbf{nil}\downarrow\mathbf{nil}}{\mathtt{f}(\mathbf{s}_1\mathbf{nil})\downarrow\mathbf{nil}} \quad \dfrac{\mathbf{nil}\downarrow\mathbf{nil}}{\mathtt{append}(\mathbf{nil},\mathbf{nil})\downarrow\mathbf{nil}}}{\mathtt{append}(\mathtt{f}(\mathbf{s}_1\mathbf{nil}),\mathtt{f}(\mathbf{s}_1\mathbf{nil}))\downarrow\mathbf{nil}}}{\mathtt{f}(\mathbf{s}_0\mathbf{s}_1\mathbf{nil})\downarrow\mathbf{nil}}$$

Figure 2: Example of reduction proof.

We shall begin by considering a call-by-value semantics, which is defined in Figure 1. The meaning of $t\downarrow v$ is that t evaluates to the constructor term v. A derivation π of the judgement $t\downarrow v$ will be called a *reduction proof.* The program $\mathtt{f}$ computes a relation $[\![\mathtt{f}]\!]\subseteq\mathcal{T}(\mathcal{C})^n\times\mathcal{T}(\mathcal{C})$ defined by:

— for all $u_i\in\mathcal{T}(\mathcal{C})$, we have $v\in[\![\mathtt{f}]\!](u_1,\cdots,u_n)$ if and only if there is a derivation for $\mathtt{f}(u_1,\cdots,u_n)\downarrow v$.

The size $|J|$ of a judgement $J=t\downarrow v$ is the sum $|t|+|v|$ of the size of the two terms composing the judgement. The size $|\pi|$ of any reduction proof π is the sum of $|J|$ over all the occurrences of judgements J in π. In deterministic programs, the size $|\pi|$ of $\pi : t\downarrow v$ can be safely considered as the cost of computing v from t.

The following is an example of program that we will use throughout the paper ($i\in\{0,1\}$):

$$\begin{array}{ll}
\mathtt{f}(\mathbf{s}_0\mathbf{s}_i x) & \to \mathtt{append}(\mathtt{f}(\mathbf{s}_1 x),\mathtt{f}(\mathbf{s}_1 x))\\
\mathtt{f}(\mathbf{s}_1 x) & \to x\\
\mathtt{f}(\mathbf{nil}) & \to \mathbf{nil}\\
\mathtt{append}(\mathbf{s}_i x, y) & \to \mathbf{s}_i\mathtt{append}(x,y)\\
\mathtt{append}(\mathbf{nil}, y) & \to y
\end{array}$$

Figure 2 shows a derivation π.

The semantic rules in Figure 1 can be either active or passive.

Definition 163 (Classifying terms, rules and judgements)**.** The *passive* semantic rules are Constructor and Split; the only *active* rule is Function. Let $\pi : t\downarrow v$ be a reduction proof. If we have

$$\frac{e=\mathtt{g}(q_1,\cdots,q_n)\to q \quad \sigma\in\mathfrak{S} \quad q_i\sigma=u_i \quad q\sigma\downarrow u}{s=\mathtt{g}(u_1,\cdots,u_n)\downarrow u},$$

then we say that term s (respectively, judgement $J=s\downarrow u$) is *active* in π. We say that e is the equation *activated* by s (respectively, J) in π and $q\sigma$ (respectively, $q\sigma\downarrow u$) is the *activation* of s (respectively, J) in π. Other judgements (conclusions of Split or Constructor rules) are said to be *passive.*

Notice that the set of active terms in a derivation π is exactly the set of terms of the form $\mathtt{f}(v_1,\cdots,v_n)$ appearing in π, where v_i are constructor terms. Since the program may be non-deterministic, the equation activated by a term s depends on the reduction and on the occurrence of s in π, and not just on s.

Our aim now is to prove that the size $|\pi|$ of any reduction proof $\mathtt{f}(v_1,\cdots,v_n)\downarrow u$ does not really depend on passive judgements in π. This will motivate our study of the combinatorics of reduction by way of *call-trees*, which we shall define later.

Proposition 164. *For all programs, there exists a polynomial $p:\mathbb{N}^2\to\mathbb{N}$ such that for every derivation proof $\pi : f(v_1,\cdots,v_n)\downarrow u$, if A is the number of active judgements in π and S is the maximum size of any active judgement in π, then $|\pi|\le p(A,S)$.*

Proof. First, observe that for each program, there exists a polynomial $r:\mathbb{N}\to\mathbb{N}$ such that for any reduction proof π, for any active term t in it and for any activation s of t, we have $|s|\le r(|t|)$. That is, the size of s is polynomially bounded by the size of t. This is because any program consists of a *finite* set of equations, each of them leading to at most a polynomial increase in size. Now consider any occurrence of a passive judgement H inside a reduction proof π. Clearly, in the path from H to the root of π, there is at least an active judgement (by hypothesis, $\pi : \mathtt{f}(v_1,\cdots,v_n)\downarrow u$, so the conclusion of π is an active judgement itself): take the one that is immediately below H. In this way we can associate with any active judgement J in π a set of passive judgements D_J such that any passive judgement is in some D_J . We will now show that in D_J there are at most $r(S)$ judgement occurrences. Consider the (Function) rule with J as conclusion. The premise H of this (Function) rule is simply an activation of its conclusion, so its size is bounded by $r(S)$. Moreover, the left-hand side of any passive judgements in D_J is a subterm of the left-hand side of H, so there are at most $r(S)$ such passive judgements. This implies that the number of rule instances in the reduction proof π is at most

$$A+A\cdot r(S).$$

But what about the size of passive judgements in π? The left-hand side of any passive judgement has size at most $r(S)$. Indeed, any passive judgement in D_J can be written as $t \downarrow v$, where t is a subterm of an activation of the left-hand side of J. We now note that if v is the right-hand side of any passive judgement in π, then $|v| \leq \max\{S, |u|\}$, where u is the right-hand side of the conclusion of π (this follows by induction on π). As a consequence, $|v| \leq S$, because the right-hand side of the conclusion of π is part of an active judgement itself. Putting everything together, we get

$$|\pi| \leq (A + A \cdot r(S))(r(S) + S),$$

so $p(x, y)$ is simply $(x + xr(y))(r(y) + 2y)$, which completes the proof. □

$$\frac{\mathbf{c} \in \mathcal{C} \quad \langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle}{\langle C_0, \mathbf{c}(t_1, \cdots, t_n)\rangle \Downarrow \langle C_n, \mathbf{c}(v_1, \cdots, v_n)\rangle}\ \text{(Constructor)}$$

$$\frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad \langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle \quad \langle C_n, \mathtt{f}(v_1, \cdots, v_n)\rangle \Downarrow \langle C, v\rangle}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \Downarrow \langle C, v\rangle}\ \text{(Split)}$$

$$\frac{(\mathtt{f}, v_1, \cdots, v_n, v) \in C}{\langle C, \mathtt{f}(v_1, \cdots, v_n)\rangle \Downarrow \langle C, v\rangle}\ \text{(Read)}$$

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \langle C, r\sigma\rangle \Downarrow \langle C', v\rangle}{\langle C, \mathtt{f}(v_1, \cdots, v_n)\rangle \Downarrow \langle C' \cup \{(\mathtt{f}, v_1, \cdots, v_n, v)\}, v\rangle}\ \text{(Update)}$$

Figure 3: Memoisation semantics of a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}\rangle$.

We will now consider a call-by-value semantics with memoisation for confluent programs, following [MM00]. The idea is to maintain a cache C to avoid recomputing the same results several times. Each time a function call is performed, the semantics looks in the cache C. If the same call has already been computed, the result can be given immediately, otherwise we need to compute the corresponding value and store the result in the cache (for later reuse). The memoisation semantics is defined in Figure 3. The Update rule can only be triggered if the Read rule cannot, that is, if there is no v such that $(\mathtt{f}, v_1, \cdots, v_n, v) \in C$. In other words, memoisation corresponds to an automation of the algorithmic technique of dynamic programming. The expression $\langle C, t\rangle \Downarrow \langle D, v\rangle$ means that the result of reducing t is v, given a program $\mathtt{f}$ and an initial cache C. The final cache D contains C together with all the calls required to complete the computation. The size $|J|$ of a judgement $\langle C, t\rangle \Downarrow \langle D, v\rangle$ is now the sum $|t| + |v|$. The size $|\pi|$ of any reduction proof $\pi : \langle C, t\rangle \Downarrow \langle D, v\rangle$ is the sum of $|J|$ over all the occurrences of the judgements J in π.

$$\dfrac{\dfrac{\langle \emptyset, \mathbf{nil}\rangle \Downarrow \langle \emptyset, \mathbf{nil}\rangle}{\langle \emptyset, \mathtt{f}(\mathrm{s}_1\mathbf{nil}) \Downarrow \langle C, \mathbf{nil}\rangle} \quad \dfrac{}{\langle C, \mathtt{f}(\mathrm{s}_1\mathbf{nil})\rangle \Downarrow \langle C, \mathbf{nil}\rangle} \quad \dfrac{\langle C, \mathbf{nil}\rangle \Downarrow \langle C, \mathbf{nil}\rangle}{\langle C, \mathtt{append}(\mathbf{nil}, \mathbf{nil})\rangle \Downarrow \langle D, \mathbf{nil}\rangle}}{\dfrac{\langle \emptyset, \mathtt{append}(\mathtt{f}(\mathrm{s}_1\mathbf{nil}), \mathtt{f}(\mathrm{s}_1\mathbf{nil}))\rangle \Downarrow \langle D, \mathbf{nil}\rangle}{\langle \emptyset, \mathtt{f}(\mathrm{s}_0\mathrm{s}_1\mathbf{nil})\rangle \Downarrow \langle E, \mathbf{nil}\rangle}}$$

Figure 4: Example of reduction proof.

Figure 4 shows a reduction proof π of our example program in the memoisation semantics, where we have used the following abbreviations:

$$\begin{aligned}
C &= \{(\mathtt{f}(\mathrm{s}_1\mathbf{nil}), \mathbf{nil})\} \\
D &= \{(\mathtt{f}(\mathrm{s}_1\mathbf{nil}), \mathbf{nil}), (\mathtt{append}(\mathbf{nil}, \mathbf{nil}), \mathbf{nil})\} \\
E &= \{(\mathtt{f}(\mathrm{s}_1\mathbf{nil}), \mathbf{nil}), (\mathtt{append}(\mathbf{nil}, \mathbf{nil}), \mathbf{nil}), (\mathtt{f}(\mathrm{s}_0\mathrm{s}_1\mathbf{nil}), \mathbf{nil})\}.
\end{aligned}$$

We also need to classify the rules, terms and judgements in the memoisation semantics.

Definition 165 (Classifying terms, rules and judgements). Constructor and Split are *passive* semantic rules; Update is an *active* rule; and Read is a *semi-active* rule. Active terms and judgements, activated equations, activations and dependencies are defined in a similar way to the call-by-value case. Semi-active terms and judgements are similarly defined from semi-active rules.

As in the call-by-value semantics, we can concentrate our attention on active judgements when reasoning about the size of π of a reduction.

Proposition 166. *Consider the memoisation semantics of Figure 3. For all programs, there exists a polynomial p such that for all derivation proofs $\pi : \langle \emptyset, f(v_1, \cdots, v_n) \Downarrow \langle C, u \rangle$, if A is the number of distinct active judgements in π and S is the maximum size of an active judgement in π, then $|\pi| \leq p(A, S)$.*

Proof. We first observe that the conclusions of active rule instances in π are distinct from each other. So A is indeed the number of active rule instances in π. The total number of active and passive rule instances can be bound as in Proposition 164 by

$$A + A \cdot r(S).$$

Since semi-active judgements form a subset of the set of leaves in π and since the number of premises of a rule is statically bounded (by $(k+1)$, where k is the maximum arity of a symbol), the number of semi-active judgements is polynomially bounded by the number of non-leaf judgements in π.

Bounding the size of any judgement in π requires a little more care than in Proposition 164. We can proceed as follows by considering a sub-derivation $\rho : \langle D, t \rangle \Downarrow \langle E, v \rangle$ in π:

- The size $|t|$ of t is at most $r(S)$.
- The cardinals of D and E are bounded by A.
- The size of elements of D and E is bounded by S.
- The size $|v|$ of v is at most S.

This concludes the proof. □

Lemma 167. *Consider the memoisation semantics of Figure 3. Let $J = \langle D, t \rangle \Downarrow \langle E, v \rangle$ be a semi-active judgement in a proof $\pi : \langle \emptyset, t \rangle \Downarrow \langle C, v \rangle$. Then there exists an active judgement $H = \langle F, t \rangle \Downarrow \langle G, v \rangle$ in π. We say that H is the active judgement corresponding to J.*

Proof. The proof follows from the fact that the pair can only be in the cache if an active judgement put it there. □

There is no obvious way to use memoisation with non-confluent programs. Indeed, the same function call can lead to several different results. Several ideas could be used to define a memoisation semantics for non-confluent programs, but they all have problems, so we will not use any of them here and only use memoisation when the program is confluent. For sufficient conditions to check that a program is confluent, see, for instance, Huet's work [Hue80].

2.2 Call trees and call dags

Following [BMM11], we will now present call-trees, which we shall use as a tool throughout the paper. Note that this is not a new kind of semantics for programs but an analysis tool, which we will use to study both the call-by-value and memoisation semantics of execution. A call-tree indeed provides a static view of an execution and describes all function calls. Hence, we can study dependencies between function calls without taking care of the extra details provided by the underlying rewriting relation.

Definition 168 (States)**.** A state is a tuple $(\mathtt{f}, v_1, \cdots, v_n, v)$ where $\mathtt{f}$ is a function symbol of arity n, $v_1, \cdots, v_n$ are constructor terms and $v \in [\![\mathtt{f}]\!](v_1, \cdots, v_n)$. Assume that $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ and $\mu = (\mathtt{g}, u_1, \cdots, u_n, u)$ are two states. A transition is a triplet $\eta \overset{e}{\rightsquigarrow} \mu$ such that:

1. e is an equation $\mathtt{f}(p_1, \cdots, p_n) \to t$ of $\mathcal{E}$,
2. there is a substitution σ such that $p_i\sigma = v_i$ for all $1 \leq i \leq n$,
3. there is a subterm $\mathtt{g}(s_1, \cdots, s_m)$ of t such that $s_i\sigma \downarrow u_i$ for all $1 \leq i \leq m$.

We write $\overset{*}{\rightsquigarrow}$ for the reflexive transitive closure of $\bigcup_{e \in \mathcal{E}} e$.

The above definition of a state is slightly different from, though essentially equivalent to, the one given in [BMM11].

Definition 169 (Call trees)**.** Consider the call-by-value semantics. Let $\pi : t \downarrow v$ be a reduction proof. Its set of call trees is the set of trees Θ_π obtained by only keeping active terms in π.

In other words, if t is passive, that is,

$$\frac{b \in \mathcal{F} \bigcup C \quad \pi_i : t_i \downarrow v_i}{b(t_1, \cdots, t_n) \downarrow b(v_1, \cdots, v_n)} \text{(Constructor) or (Split)},$$

then $\Theta_\pi = \bigcup \Theta_{\pi_i}$. If t is active, that is,

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to s \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \rho : s\sigma \downarrow v}{\mathtt{f}(v_1, \cdots, v_n) \downarrow v} \text{(Function)},$$

then Θ_π only contains the tree whose root is $(\mathtt{f}, v_1, \cdots, v_n, v)$ and children are Θ_ρ.

The following is the only element of Θ_π, where π is the reduction proof in Figure 2:

$$\begin{tikzcd}
 & (\mathtt{f}, \mathrm{s}_0\mathrm{s}_1\mathbf{nil}, \mathbf{nil}) \arrow[dl] \arrow[d] \arrow[dr] & \\
(\mathtt{f}, \mathrm{s}_1\mathbf{nil}, \mathbf{nil}) & (\mathtt{f}, \mathrm{s}_1\mathbf{nil}, \mathbf{nil}) & (\mathtt{append}, \mathbf{nil}, \mathbf{nil}, \mathbf{nil})
\end{tikzcd}$$

When using the memoisation semantics, some function calls are not recomputed but instead fetched from the cache, so it is convenient to represent the dependencies of function calls by a directed acyclic graph (dag) rather than by a tree.

Definition 170 (Call dags)**.** Consider the memoisation semantics. Let $\pi : \langle C, t\rangle \Downarrow \langle D, v\rangle$ be a reduction proof. Its set of call dags Θ_π is defined as in Definition 169 by keeping active terms, and by replacing each semi-active term by a link to the corresponding active term, according to Lemma 167.

For instance, the following dag is the only element of Θ_π, where π is the reduction proof in Figure 4:

$$\begin{tikzcd}
(\mathtt{f}, \mathrm{s}_0\mathrm{s}_1\mathbf{nil}, \mathbf{nil}) \arrow[d, bend right] \arrow[d, bend left] \arrow[dr] & \\
(\mathtt{f}, \mathrm{s}_1\mathbf{nil}, \mathbf{nil}) & (\mathtt{append}, \mathbf{nil}, \mathbf{nil}, \mathbf{nil})
\end{tikzcd}$$

Fact 1 (Call tree arity)*.* Let $\mathtt{f}$ be a program and consider the call-by-value semantics. There exists a fixed integer k such that given a derivation π of a term of the program and a tree $\mathcal{T}$ of Θ_π, all nodes in $\mathcal{T}$ have at most k children.

Proof. For each right-hand side term r of an equation of f, consider the number of maximal subterms of r with a function as head symbol. We can then take k to be the maximum of these integers over the (finite) set of equations of $\mathtt{f}$. □

Lemma 171. *Consider the call-by-value semantics. Let π be a reduction proof, $\mathcal{T}$ be a call-tree in Θ_π and consider two states $\eta = (f, v_1, \cdots, v_n, v)$ and $\mu = (g, u_1, \cdots, u_n, u)$ such that μ is a child of η. Then there is an equation e such that $t = f(v_1, \cdots, v_n)$ activates e in π and $\eta \overset{e}{\rightsquigarrow} \mu$. Conversely, if $\eta \overset{e}{\rightsquigarrow} \mu$, then μ is a child of η in $\mathcal{T}$.*

Proof. The proof is by a trivial induction on the structure of π. □

This implies that in the deterministic case our definition of call trees is equivalent to the one in [BMM11]. However, we did need a new definition in order to deal with non-determinism.

Recall that the size $|\pi|$ of a reduction proof π (in either the call-by-value or memoisation semantics) takes into account all the symbol occurrences in π: in this way, it can be considered to be a reasonable approximation to the execution time. As a consequence, bounding execution time in a computation boils down to:

1. bound the size (number of nodes) in the call tree or call dag; and
2. bound the size of the states appearing in the call tree (dag).

Proposition 172. *For any program f:*
Call-by-value semantics*: There is a polynomial $p_f : \mathbb{N}^2 \to \mathbb{N}$ such that whenever $\pi : f(u_1, \cdots, u_n) \downarrow v$ and the set Θ_π of call trees contains $a_\pi \in \mathbb{N}$ states whose size is at most $s_\pi \in \mathbb{N}$, then $|\pi| \leq p_f(a_\pi, s_\pi)$.*
Memoisation semantics*: Assume f is deterministic. There is a polynomial $q_f : \mathbb{N}^2 \to \mathbb{N}$ such that whenever $\pi : \langle \emptyset, f(u_1, \cdots, u_n)\rangle \Downarrow \langle C, v\rangle$ and the set Θ_π of call dags contains $a_\pi \in \mathbb{N}$ states whose size is at most $s_\pi \in \mathbb{N}$, then $|\pi| \leq q_f(a_\pi, s_\pi)$.*

Proof. This is an easy corollary of Propositions 164 and 166. □

The naive model where each rule takes unary time to be executed is not very realistic with the memoisation semantics. Indeed, each Read and Update rule needs to perform a lookup in the cache, and this would in practice take time proportional to the size of the cache (and the size of elements in it). However, the size of the final cache is exactly the number of Update rules in the proof (because only Update modifies the cache), and the size of terms in the cache is bounded by the size of active terms (only active terms are stored in the cache). So Proposition 172 gives a polynomial bound on the execution time.

3 Orderings and quasi-interpretations

This section presents a new way of looking at concepts and results that have already appeared in the literature [BMM11].

3.1 Termination orderings

Definition 173 (Precedence)**.** Let $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ be a program. A *precedence* $\preceq_{\mathcal{F}}$ is a preorder over $\mathcal{F} \bigcup \mathcal{C}$. We use $\approx_{\mathcal{F}}$ to denote the associated equivalence relation. A precedence is *compatible* with $\mathtt{f}$ if for each equation $\mathtt{g}(p_1, \cdots, p_n) \to r$ and each symbol b appearing in r, we have $b \preceq_{\mathcal{F}} \mathtt{g}$, and it is *separating* if for each $\mathbf{c} \in \mathcal{C}$, $\mathtt{f} \in \mathcal{F}$, $\mathbf{c} \prec_{\mathcal{F}} \mathtt{f}$ (that is, constructors are the smallest elements of $\prec_{\mathcal{F}}$, and functions are the biggest). It is *fair* if for all constructors $\mathbf{c}$ and $\mathbf{d}$ with the same arity, we have $\mathbf{c} \approx_{\mathcal{F}} \mathbf{d}$, and it is *strict* if for all distinct constructors $\mathbf{c}$ and $\mathbf{d}$, we have $\mathbf{c}$ and $\mathbf{d}$ are incomparable.

Any strict precedence can be canonically extended into a fair precedence.

$$\frac{s = t_i \text{ or } s \prec_{ppo} t_i}{s \prec_{ppo} \mathtt{f}(\ldots, t_i, \ldots)}\ \mathtt{f} \in \mathcal{F} \bigcup \mathcal{C} \qquad \frac{\forall i\ s_i \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n) \qquad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}}{g(s_1, \cdots, s_m) \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n)}\ \mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{(s_1, \cdots, s_n) \prec^p_{rpo} (t_1, \cdots, t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \qquad \forall i\ s_i \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(s_1, \cdots, s_n) \prec_{ppo} \mathtt{f}(t_1, \cdots, t_n)}\ \mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C}$$

Figure 5: Definition of $\prec_{ppo}$

Definition 174 (Product extension)**.** Let $\prec$ be an ordering over a set S. Its *product extension* is an ordering $\prec^p$ over tuples of elements of S such that $(m_1, \cdots, m_k) \prec^p (n_1, \cdots, n_k)$ if and only if:

1. $\forall i, m_i \preceq n_i$; and
2. $\exists j$ such that $m_j \prec n_j$.

Definition 175 (PPO)**.** Given a separating precedence $\preceq_{\mathcal{F}}$, the recursive path ordering $\prec_{ppo}$ is defined in Figure 5. If $\prec_{\mathcal{F}}$ is strict (respectively, fair) and separating, then the ordering is the *Product Path Ordering* PPO (respectively, the *Extended Product Path Ordering* EPPO).

Of course, we could consider other ordering extensions. The usual choices are the lexicographic extension, leading to Lexicographic Path Ordering or the Multiset extension, leading to Multiset Path Ordering. It is also possible to add a notion of *status* to function symbols [KL80] indicating with which extension the parameters must be compared. This leads to the more general Recursive Path Ordering (RPO). However, here we will only use the (extended) Product Path Ordering, and will not describe any of the others.

Given a precedence $\preceq_{\mathcal{F}}$, an equation $l \to r$ is decreasing with respect to $\preceq_{\mathcal{F}}$ if we have $r \prec_{ppo} l$. A program is ordered by PPO if there is a separating precedence $\prec_{\mathcal{F}}$ on $\mathcal{F}$ such that each equation is decreasing with respect to $\prec_{\mathcal{F}}$. Recall that $\prec_{ppo}$ guarantees termination [Der82]. Notice that in our case, since patterns cannot contain function symbols, if there is a precedence such that the program is ordered by the corresponding $\prec_{ppo}$, then there is also a compatible one with the same condition.

Lemma 176. *Let f be a program and $\prec_{\mathcal{F}}$ be a separating precedence compatible with it. Let $\eta = (f, v_1, \cdots, v_n, v)$ be a state in a call tree (respectively, dag) $\mathcal{T}$ and $\mu = (g, u_1, \cdots, u_m, u)$ be a descendant of η in $\mathcal{T}$. Then $g \preceq_{ppo} f$.*

Proof. The claim follows from the fact that the precedence is compatible with $\mathtt{f}$. ☐

Proposition 177 (Computing by rank)**.** *Let f be a program and $\prec_{\mathcal{F}}$ be a separating precedence compatible with it. Let $\mathcal{T}$ be a call dag. Let $A_{\mathcal{T}}$ be the maximum number of descendants of a node with the same precedence:*

$$A_{\mathcal{T}} = \max_{\eta=(f,v_1,\cdots,v_n,v)\in\mathcal{T}} |\{\mu = (g, u_1, \cdots, u_m, u),\ \eta \textit{ is an ancestor of } \mu \textit{ and } g \approx_{\mathcal{F}} f\}|$$

The size of $\mathcal{T}$ (the number of nodes) is polynomially bounded by $A_{\mathcal{T}}$.

Proof. Let $\mathtt{f}$ be a function symbol. Its rank is $rk(\mathtt{f}) = \max_{\mathtt{g}\prec_{\mathcal{F}}\mathtt{f}} rk(\mathtt{g}) + 1$. Let d be the maximum number of function symbols in the right-hand side of $\mathtt{f}$ and k be the maximum rank. We will prove by induction on $k-i$ that there are at most $B_i = k^{k-i} \times d^{k-i} \times A_{\mathcal{T}}^{k-i+1}$ nodes in $\mathcal{T}$ at rank i:

- If $k - i = 0$, then $i = k$, and there are at most $A_{\mathcal{T}} = k^{k-k} d^{k-k} A_{\mathcal{T}}^{k-k+1} = B_k$ nodes at rank k.
- Suppose the hypothesis is true for all ranks $j > i$. Each node has at most d children. Hence, there are at most $d\sum_{k\geq h>i} B_h$ nodes at rank i whose parent has rank different from i. Each of these nodes has at most $A_{\mathcal{T}}$ descendants at rank i, so there are at most $d \times A_{\mathcal{T}} \times k \sum_{k\geq h>i} B_h$ nodes at rank i. But

$$\begin{aligned} d \cdot A_{\mathcal{T}} \cdot \sum_{k\geq h>i} B_h &\leq d \cdot A_{\mathcal{T}} \cdot \sum_{k\geq h>i} k^{k-h} \cdot d^{k-h} \cdot A_{\mathcal{T}}^{k-h+1} \\ &\leq d \cdot A_{\mathcal{T}} \cdot \sum_{k\geq h>i} k^{k-(i+1)} \cdot d^{k-(i+1)} \cdot A_{\mathcal{T}}^{k-(i+1)+1} \\ &\leq d \cdot A_{\mathcal{T}} \cdot k \cdot k^{k-(i+1)} \cdot d^{k-(i+1)} \cdot A_{\mathcal{T}}^{k-(i+1)+1} \\ &= k^{k-i} \cdot d^{k-i} \cdot A_{\mathcal{T}}^{k-i+1}. \end{aligned}$$

So B_i is polynomially bounded in $A_{\mathcal{T}}$, as is the size of the call tree (dag).

This concludes the proof. □

Thus, to bound the number of nodes in a call-dag (and thus, by Proposition 172 bound the derivation's size), it suffices to establish the bound rank by rank.

Proposition 178. *Let f be a program terminating by PPO, $\mathcal{T}$ be a call dag and $\eta = (f, v_1, \cdots, v_n, v)$ be a node in $\mathcal{T}$. The number of descendants of η in $\mathcal{T}$ with the same rank as η is polynomially bounded by $|\eta|$.*

Proof. Because of the PPO termination ordering, if $\mu = (\mathtt{g}, u_1, \cdots, u_m, u)$ is a descendant of η with $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, then u_i is a subterm of some v_j. There are at most $|v_j|$ subterms of v_j, so there are at most $c \cdot d \cdot \max |v_i|$ possible nodes (where c is the number of functions with the same precedence as $\mathtt{f}$ and d is the maximum arity of symbols in $\mathtt{f}$). □

This is point (2) in the proof of [BMM11], Lemma 51. Notice that this only works on a call dag (because identical nodes collapse) and not on a call tree.

3.2 Quasi-interpretations

We restrict ourselves to additive QIs as defined in [BMM11].

Definition 179 (Assignment)**.** An assignment of a symbol $b \in \mathcal{F}\bigcup\mathcal{C}$ whose arity is n is a function $(\!|b|\!) : (\mathbb{R})^n \to \mathbb{R}$ such that:
Subterm: $(\!|b|\!)(X_1, \cdots, X_n) \geq X_i$ for all $1 \leq i \leq n$.
Weak monotonicity: $(\!|b|\!)$ is increasing with respect to each variable:

$$\text{for all } 1 \leq i \leq n,\ X_i \leq Y_i \Rightarrow (\!|b|\!)(X_1, \cdots, X_n) \leq (\!|b|\!)(Y_1, \cdots, Y_n)$$

Additivity: $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^n X_i + a$ if $\mathbf{c} \in \mathcal{C}$ (where $a \geq 1$).
Polynomial: $(\!|b|\!)$ is bounded by a polynomial.

We extend assignments $(\!|\cdot|\!)$ to terms in the canonical way. Given a term t with n variables, the assignment $(\!|t|\!)$ is a function $(\mathbb{R})^n \to \mathbb{R}$ defined by the rules

$$\begin{aligned} (\!|b(t_1, \cdots, t_n)|\!) &= (\!|b|\!)((\!|t_1|\!), \cdots, (\!|t_n|\!)) \\ (\!|x|\!) &= X. \end{aligned}$$

Given two functions $f : (\mathbb{R})^n \to \mathbb{R}$ and $g : (\mathbb{R})^m \to \mathbb{R}$ such that $n \geq m$, we say that $f \geq g$ if and only if $\forall X_1, \cdots, X_n \in \mathbb{R} : f(X_1, \cdots, X_n) \geq g(X_1, \cdots, X_m)$. There are some wellknown and useful consequences of such definitions. We have $(\!|s|\!) \geq (\!|t|\!)$ if t is a subterm of s. Then, for every substitution σ, we have $(\!|s|\!) \geq (\!|t|\!)$ implies that $(\!|s\sigma|\!) \geq (\!|t\sigma|\!)$.

Definition 180 (Quasi-interpretation)**.** A program assignment $(\!|\cdot|\!)$ is an assignment of each program symbol. An assignment $(\!|\cdot|\!)$ of a program is a quasi-interpretation (QI) if for each equation $l \to r$,

$$(\!|l|\!) \geq (\!|r|\!).$$

In the following, unless explicitly specified, $(\!|\cdot|\!)$ will always denote a QI and not an assignment.

Lemma 181. *There exists a constant a such that for any constructor term v, we have $|v| \leq (\!|v|\!) \leq a|v|$.*

Proof. By induction, we can show that the constant a depends on the constants in the QI of constructors. □

Lemma 182. *Assume f has a QI. Let $\eta = (f, v_1, \cdots, v_n, v)$ and $\mu = (g, u_1, \cdots, u_m, u)$ be two states such that $\eta \overset{*}{\rightsquigarrow} \mu$. Then, $(\!|g(u_1, \cdots, u_m)|\!) \leq (\!|f(v_1, \cdots, v_n)|\!)$.*

Proof. The claim follows from the fact that $\mathtt{g}(u_1, \cdots, u_m)$ is a subterm of a term obtained by reduction from $\mathtt{f}(v_1, \cdots, v_n)$. □

Proposition 183. *Let f be a program (of arity n) admitting a QI. Then there is a polynomial $p_f : \mathbb{N}^n \to \mathbb{N}$ such that for every $\pi : \langle \emptyset, f(v_1, \cdots, v_n)\rangle \Downarrow \langle D, v\rangle$, the size of any active judgement in π is bounded by $p_f(|v_1|, \cdots, |v_n|)$.*

Proof. Let $\mathtt{g}(u_1, \cdots, u_m) \downarrow r$ be an active judgement in π. The size of

$$s = \mathtt{g}(u_1, \cdots, u_m)$$

is bounded by

$$m \cdot \max |u_i| + 1 \leq m \cdot \max(\!|u_i|\!) + 1 \leq m \cdot (\!|s|\!) + 1.$$

By Lemma 182, $(\!|s|\!) \leq (\!|t|\!)$, where $t = \mathtt{f}(v_1, \cdots, v_n)$. But by the polynomiality of QIs, there is a polynomial p such that $(\!|t|\!) \leq p((\!|v_1|\!), \cdots, (\!|v_n|\!))$. Since v_i are constructor terms, $(\!|v_i|\!) \leq a|v_i|$. Moreover, $|r| \leq (\!|r|\!) \leq (\!|s|\!)$. The claim then follows easily. □

If we now combine this bound on the size of active terms with the bound on the number of active terms of Proposition 178, we can apply Proposition 172 and conclude that programs terminating by PPO and admitting a QI are polynomial time computable. In fact, all polytime functions can be obtained in this way.

Theorem 184 (P-criterion, [BMM11])**.** *Any deterministic program f that*

1. *terminates by PPO and*
2. *admits a QI*

is executable in polynomial time with the memoisation semantics. Conversely, any function in FP can be represented by a program of this kind.

4 Blind abstractions of programs

4.1 Definitions

Our idea is to associate with a given program $\mathtt{f}$ an abstract program $\overline{\mathtt{f}}$ obtained by forgetting each piece of data and replacing it by its *shape*: for instance, strings over a finite alphabet are replaced by tally integers (their length), binary trees over a finite alphabet by binary trees without labels, and so on. Note that in this way, even if $\mathtt{f}$ is deterministic, the associated $\overline{\mathtt{f}}$ will not be deterministic in general.

To do this we will first define a target language:
Variables: $\overline{\mathcal{X}} = \mathcal{X}$,
Function symbols: We define the map $\overline{(\cdot)}$ on function symbols by $\overline{\mathtt{f}} = \overline{\mathtt{g}}$ if and only if $\mathtt{f} = \mathtt{g}$. Then $\overline{\mathcal{F}} = \{\overline{\mathtt{f}}/\mathtt{f} \in \mathcal{F}\}$.
Constructor symbols: We define the map $\overline{(\cdot)}$ on constructor symbols by $\overline{\mathbf{c}} = \overline{\mathbf{d}}$ if and only if $\mathbf{c}$ and $\mathbf{d}$ have the same arity. Then $\overline{\mathcal{C}} = \{\overline{\mathbf{c}}/\mathbf{c} \in \mathcal{C}\}$.

This language defines a set of values $\mathcal{T}(\overline{\mathcal{C}})$, a set of terms $\mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$ and a set of patterns $\overline{\mathcal{P}}$.

The *blinding map* is the natural map $\mathcal{B} : \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \to \mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$ induced by the maps above: basically, it just identifies constructors of the same arity. It induces similar maps on values and patterns. We will write $\overline{t}$ (respectively, $\overline{p}$) for $\mathcal{B}(t)$ (respectively, $\mathcal{B}(p)$).

For instance, binary strings built from constructors $\{\mathbf{s}_0, \mathbf{s}_1, \mathbf{nil}\}$ will be mapped to unary strings (or tally integers) corresponding to their length, built from $\{\mathbf{s}, \mathbf{0}\}$, where $\overline{\mathbf{s}_0} = \overline{\mathbf{s}_1} = \mathbf{s}$ and $\overline{\mathbf{nil}} = \mathbf{0}$.

The blinding map extends to equations in the expected way: given an equation $d = l \to r$ of the language $(\mathcal{X}, \mathcal{F}, \mathcal{C})$, we set $\overline{d} = \mathcal{B}(d) = \overline{l} \to \overline{r}$. Finally, given a program $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}\rangle$, its blind image is $\overline{\mathtt{f}} = \langle \overline{\mathcal{X}}, \overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{E}}\rangle$, where the

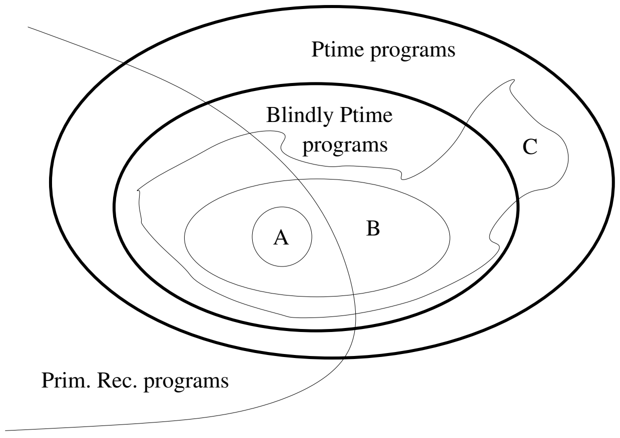


Figure 6: Subclasses of programs

equations are obtained by $\overline{\mathcal{E}} = \{\overline{d}/d \in \mathcal{E}\}$. Observe that even if $\mathtt{f}$ is an orthogonal program, this will not necessarily be the case for $\overline{\mathtt{f}}$ because some patterns are identified by $\mathcal{B}$. The denotational semantics of $\overline{\mathtt{f}}$ can be viewed as a relation over the domain $\mathcal{T}(\overline{\mathcal{C}})$.

We have mentioned strings over a binary alphabet, but note that on some other datatypes, such as lists over tally integers, the blinding map might just act as the identity and thus not have much interest. Other notions of abstractions are probably worth considering, but we will stick here to the blinding map for simplicity.

4.2 Complexity definitions

Definition 185 (Strongly polytime)**.** We say a (possibly) non-deterministic program $\mathtt{f}$ (of arity n) is *strongly polytime* if there exists a polynomial $p_{\mathtt{f}} : \mathbb{N}^n \to \mathbb{N}$ such that for any sequence $v_1, \cdots, v_n$ and any $\pi : \mathtt{f}(v_1, \cdots, v_n) \downarrow u$, we have $|\pi| \leq p_{\mathtt{f}}(|v_1|, \cdots, |v_n|)$.

Of course, similar definitions would also make sense for complexity bounds other than polynomial time. In the case of a deterministic program, this definition coincides with that of a polynomial time program.

Definition 186 (Blindly polytime)**.** A program $\mathtt{f}$ is *blindly polytime* if its blind abstraction $\overline{\mathtt{f}}$ is strongly polytime.

Fact 2. If a program $\mathtt{f}$ is blindly polytime, it is polynomial time in the call-by-value semantics.

Indeed, it is sufficient to observe that any execution (derivation proof) of $\mathtt{f}$ can be mapped by $\mathcal{B}$ to an execution of $\overline{\mathtt{f}}$. But, of course, not all $\overline{\mathtt{f}}$ executions are obtained in this way. Note that for our running example in Figure 7, $\mathtt{f}$ terminates in polynomial time, but this is not the case for $\overline{\mathtt{f}}$. Indeed, if we use the denotation

$$\underline{n} = \underbrace{\mathtt{s} \ldots \mathtt{s}}_{n}\ \mathbf{0},$$

we have that $\overline{\mathtt{f}}(\underline{n})$ can be reduced in an exponential number of steps, with a $\pi : \overline{\mathtt{f}}(\underline{n}) \downarrow \mathbf{0}$.

	$\mathtt{f}$			$\overline{\mathtt{f}}$	
$\mathtt{f}(\mathtt{s}_0\mathtt{s}_i x)$	$\to$	$\mathtt{append}(\mathtt{f}(\mathtt{s}_1 x), \mathtt{f}(\mathtt{s}_1 x))$	$\overline{\mathtt{f}}(\mathtt{ss}x)$	$\to$	$\overline{\mathtt{append}}(\overline{\mathtt{f}}(\mathtt{s}x), \overline{\mathtt{f}}(\mathtt{s}x))$
$\mathtt{f}(\mathtt{s}_1 x)$	$\to$	x	$\overline{\mathtt{f}}(\mathtt{s}x)$	$\to$	x
$\mathtt{f}(\mathbf{nil})$	$\to$	$\mathbf{nil}$	$\overline{\mathtt{f}}(0)$	$\to$	0
$\mathtt{append}(\mathtt{s}_i x, y)$	$\to$	$\mathtt{s}_i\mathtt{append}(x, y)$	$\overline{\mathtt{append}}(\mathtt{s}x, y)$	$\to$	$\mathtt{s}\ \overline{\mathtt{append}}(x, y)$
$\mathtt{append}(\mathbf{nil}, y)$	$\to$	y	$\overline{\mathtt{append}}(0, y)$	$\to$	y

Figure 7: Blind abstraction of our running example

Note that the property of being blindly polytime is indeed a strong condition since it means, in some sense, that the program will terminate with a polynomial bound for reasons that are indifferent to the actual content of the input but only depend on its size. For instance, one can check that the insertion sort algorithm, written in a natural way, gives a blindly polytime program (see Example 187).

Another way to think of blinding is to imagine that some *errors* might occur during the execution of the program, where by an error we mean that a constructor in the current term is replaced by another constructor of same arity (for instance, a 0-bit in a string is replaced by a 1-bit): blindly polytime programs are then programs that remain polynomial time, no matter how many errors occur during the execution.

Our motivation for introducing blind abstraction and the notion of a blindly polytime program is to provide a more abstract method for comparing different ICC systems than just simply providing examples handled by one system but not by the other. Indeed, this kind of abstraction gives a way to subdivide the class of polytime programs into smaller classes.

To illustrate our idea, consider the picture given in Figure 6. The fat lines represent classes of programs defined by an intensional property (being polytime or blindly polytime). Imagine we have three ICC criteria A, B and C implying that any program satisfying one of these criteria is polytime. If one criterion, say B, is only able to validate blindly polytime programs while another one, C, can handle non-blindly polytime programs, then one can argue that C is more accurate than B. Moreover, if A only validates primitive recursive programs while B validates some blindly polytime programs that are not primitive recursive, then B is more accurate than A.

	`isort`			$\overline{\mathtt{isort}}$	
$\mathtt{insert}(\mathbf{nil})$	$\to$	$\mathrm{s}_1\mathbf{nil}$	$\overline{\mathtt{insert}}(\mathbf{0})$	$\to$	$\mathrm{s}\mathbf{0}$
$\mathtt{insert}(\mathrm{s}_0x)$	$\to$	$\mathrm{s}_0\mathtt{insert}(x)$	$\overline{\mathtt{insert}}(\mathrm{s}x)$	$\to$	$\mathrm{s}\overline{\mathtt{insert}}(x)$
$\mathtt{insert}(\mathrm{s}_1x)$	$\to$	$\mathrm{s}_1\mathrm{s}_1x$	$\overline{\mathtt{insert}}(\mathrm{s}x)$	$\to$	$\mathrm{ss}x$
$\mathtt{isort}(\mathbf{nil})$	$\to$	$\mathbf{nil}$	$\overline{\mathtt{isort}}(\mathbf{0})$	$\to$	$\mathbf{0}$
$\mathtt{isort}(\mathrm{s}_0x)$	$\to$	$\mathrm{s}_0\mathtt{isort}(x)$	$\overline{\mathtt{isort}}(\mathrm{s}x)$	$\to$	$\mathrm{s}\overline{\mathtt{isort}}(x)$
$\mathtt{isort}(\mathrm{s}_1x)$	$\to$	$\mathtt{insert}(\mathtt{isort}(x))$	$\overline{\mathtt{isort}}(\mathrm{s}x)$	$\to$	$\overline{\mathtt{insert}}(\overline{\mathtt{isort}}(x))$

Figure 8: Insertion sort program.

	`bsort`	
$\mathtt{bubble}(\mathbf{nil}, b)$	$\to$	$< \mathbf{nil}, b >$
$\mathtt{bubble}(\mathrm{s}_i\mathbf{nil}, b)$	$\to$	$< \mathrm{s}_i\mathbf{nil}, b >$
$\mathtt{bubble}(\mathrm{s}_0\mathrm{s}_ix, b)$	$\to$	$\mathtt{let}\ < y, b' >= \mathtt{bubble}(\mathrm{s}_ix, b)\ \mathtt{in}\ < \mathrm{s}_0y, b' >$
$\mathtt{bubble}(\mathrm{s}_1\mathrm{s}_1x, b)$	$\to$	$\mathtt{let}\ < y, b' >= \mathtt{bubble}(\mathrm{s}_1x, b)\ \mathtt{in}\ < \mathrm{s}_1y, b' >$
$\mathtt{bubble}(\mathrm{s}_1\mathrm{s}_0x, b)$	$\to$	$\mathtt{let}\ < y, b' >= \mathtt{bubble}(\mathrm{s}_1x, b)\ \mathtt{in}\ < \mathrm{s}_0y, \mathtt{false} >$
$\mathtt{bsort}(x)$	$\to$	$\mathtt{let}\ < y, b >= \mathtt{bubble}(x, \mathtt{true})\ \mathtt{in}$ $\mathtt{if}\ b\ \mathtt{then}\ y\ \mathtt{else}\ \mathtt{bsort}(y)$

	$\overline{\mathtt{bsort}}$	
$\overline{\mathtt{bubble}}(\mathbf{0}, b)$	$\to$	$< \mathbf{0}, b >$
$\overline{\mathtt{bubble}}(\mathrm{s}\mathbf{0}, b)$	$\to$	$< \mathrm{s}\mathbf{0}, b >$
$\overline{\mathtt{bubble}}(\mathrm{ss}x, b)$	$\to$	$\mathtt{let}\ < y, b' >= \overline{\mathtt{bubble}}(\mathrm{s}x, b)\ \mathtt{in}\ < \mathrm{s}y, b' >$
$\overline{\mathtt{bubble}}(\mathrm{ss}x, b)$	$\to$	$\mathtt{let}\ < y, b' >= \overline{\mathtt{bubble}}(\mathrm{s}x, b)\ \mathtt{in}\ < \mathrm{s}y, b' >$
$\overline{\mathtt{bubble}}(\mathrm{ss}x, b)$	$\to$	$\mathtt{let}\ < y, b' >= \overline{\mathtt{bubble}}(\mathrm{s}x, b)\ \mathtt{in}\ < \mathrm{s}y, \mathtt{false} >$
$\overline{\mathtt{bsort}}(x)$	$\to$	$\mathtt{let}\ < y, b >= \overline{\mathtt{bubble}}(x, \mathtt{true})\ \mathtt{in}$ $\mathtt{if}\ b\ \mathtt{then}\ y\ \mathtt{else}\ \overline{\mathtt{bsort}}(y)$

Figure 9: Bubble sort program.

Example 187. Figures 8 and 9 list two programs for sorting binary strings. The first uses insertion sort while the second uses bubble sort. In the first case, since we are working on binary strings, it is sufficient to have an auxiliary function to insert 1 into a sorted string and pre-pend 0 at the beginning of the result. In the second case, we need to introduce pairing (and `let in`) as well as a boolean value as a flag to check whether a permutation occurred or not. This is a quite straightforward extension of the language. In `bsort`, the flag is initially set to `true`, and is turned to `false` when an exchange is performed. Booleans and boolean tests remain unchanged by the blinding map.

Both programs are polytime. The first terminates by PPO and admits a QI, and hence satisfies the P-criterion. However, it does not follow the safe recursion schemata [BC92] because the result of `isort` is used to control the recursion of `insert`. The insertion sort is also blindly polytime. Indeed, every recursive call decreases the length of the string whatever the precise value of the elements is.

On the other hand, the bubble sort program is not blindly polytime or even blindly terminating. Indeed, the length of the string does not decrease and termination is ensured only because the elements inside the string are reordered. In fact, the same comparison can be done several times, and the termination of the algorithm is due to the fact that the same comparison always yields the same result. This is no longer true after blinding the program because the comparison between elements of the string then becomes non-deterministic.

We will now discuss the behaviour of the blinding map with respect to criteria on TRS based on recursive path orderings (RPO) and QIs [BMM11].

4.3 Blinding and recursive path orderings

We want to examine what happens with the ordering through the blinding operation. We have the following lemma.

Lemma 188. *Let f be a program. If f terminates by PPO, then $\overline{f}$ terminates by PPO.*

Indeed, $\mathcal{F}$ and $\overline{\mathcal{F}}$ are in one–one correspondence, and it is easy to see that if $\prec_{\mathcal{F}}$ is a precedence that gives a PPO ordering for `f`, then the corresponding $\prec_{\overline{\mathcal{F}}}$ does the same for $\overline{\texttt{f}}$.

The converse is not true – see, for example, Figure 7 where the first equation terminates by PPO on the blind side but not on the non-blind side. However, we do have the following proposition.

Proposition 189. *Let f be a program. The following statements are equivalent:*

1. *f terminates by EPPO.*
2. *$\overline{f}$ terminates by EPPO.*
3. *$\overline{f}$ terminates by PPO.*

Proof. Just observe that PPO and EPPO coincide on $\mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$, and that $\overline{t} \prec_{EPPO} \overline{s}$ implies $t \prec_{EPPO} s$. □

4.4 Blinding and quasi-interpretations

Assume the program `f` admits a quasi-interpretation $(\!|\cdot|\!)$. Then, in general, this does not imply that $\overline{\texttt{f}}$ admits a quasi-interpretation. Indeed one reason why $(\!|\cdot|\!)$ cannot be simply converted into a quasi-interpretation for $\overline{\texttt{f}}$ is because, in general, a quasi-interpretation might give different assignments to several constructors of the same arity – for instance, when $(\!|\mathbf{s}_0|\!)(X) = X + 1$ and $(\!|\mathbf{s}_1|\!)(X) = X + 2$. Then, when considering $\overline{\texttt{f}}$, there is no natural choice for $(\!|\mathbf{s}|\!)$.

However, in most examples encountered in practice, a restricted class of quasi-interpretations is used.

Definition 190 (Uniform assignments)**.** An assignment for $\texttt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ is *uniform* if all constructors of the same arity have the same assignment: for each $\mathbf{c}, \mathbf{d} \in \mathcal{C}$, we have $arity(\mathbf{c}) = arity(\mathbf{d})$ implies $(\!|\mathbf{c}|\!) = (\!|\mathbf{d}|\!)$. A quasi-interpretation of `f` is uniform if it is defined by a uniform assignment.

We now have the following proposition.

Proposition 191. *The program f admits a uniform quasi-interpretation* if and only if *$\overline{f}$ admits a quasi-interpretation.*

5 Linear programs and call-by-value semantics

5.1 Definitions and main property

We want to use the blinding transformation to examine the properties of programs satisfying the P-criterion (Theorem 184). In fact, when we do this, we will restrict our attention to particular programs, specifically, *linear* programs. Essentially, this means that only one recursive call is allowed in recursive definitions (equations). This is a fairly natural restriction when we want to control the complexity since it prevents an exponential blowup on the number of recursive calls at execution time. Indeed, many common programs are linear. This notion was used in [BMM11], in combination with lexicographic path ordering (LPO). We will now give a formal definition.

Definition 192 (Linearity)**.** Let `f` be a program terminating by a recursive path ordering given by a precedence $\preceq_{\mathcal{F}}$, and `g` be a function symbol in `f`. We say that `g` is *linear* in `f` with respect to $\preceq_{\mathcal{F}}$ if, in the right-hand side term of any equation for `g`, there is at most one occurrence of a function symbol `h` with same precedence as `g`. The program `f` is linear with respect to $\preceq_{\mathcal{F}}$ if all its function symbols are linear with respect to $\preceq_{\mathcal{F}}$.

We will not mention the precedence $\preceq_{\mathcal{F}}$ explicitly when there is no ambiguity.

Proposition 193. *Let* `f` *be a (possibly non-deterministic) program that:*

1. *terminates by PPO;*
2. *admits a quasi-interpretation; and*
3. *is linear.*

Then `f` *is strongly polytime.*

Before giving the proof, we note that compared with Theorem 184 (the P-criterion Theorem from [BMM11]), the program here does not need to be deterministic, though linearity is assumed for all function symbols.

We have considered here the linearity assumption because we are interested in nondeterministic programs and memoisation is problematic when there is non-determinism (see Section 2.1); linearity is a sufficient condition to avoid the use of memoisation and maintain the complexity properties.

Proof. We consider a derivation proof $\pi : \mathtt{f}(v_1, \cdots, v_n) \downarrow u$. As the program has a quasi-interpretation, Proposition 183 gives a bound S on the size of active judgements which depends polynomially on $|v_1|, \cdots, |v_n|$. Let $\mathcal{T}$ be a call tree associated to π and $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ be a node in it. Now consider the subset of nodes of $\mathcal{T}$ obtained by keeping η, the sons of η with same precedence as `f` and proceeding hereditarily in this way. We call this set the set of descendants of $\eta = (\mathtt{f}, v_1, \cdots, v_n, v)$ with the same precedence as `f`, and the linearity of the program actually implies that this set is a branch; so the number of its elements corresponds to its depth. Termination by PPO ensures that if $\mu = (\mathtt{g}, u_1, \cdots, u_m, u)$ is the child of η with $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, then $|u_i| \leq |v_i|$, and there is at least one j such that $|u_j| < |v_j|$. So, the number of descendants of η with the same precedence is bounded by $\sum_{i=1}^{n} |v_i|$. This thus bounds the number of active judgements by rank. So we can now use Proposition 164 and conclude that $|\pi|$ is polynomially bounded in A, S, and thus also in $|v_1|, \cdots, |v_n|$. Therefore `f` is strongly polytime. □

Theorem 194. *Let* `f` *be a (possibly non-deterministic) program that:*

1. *terminates by PPO;*
2. *admits a uniform quasi-interpretation; and*
3. *is linear.*

Then `f` *is blindly polytime.*

Proof. Note that:

- `f` terminates by PPO, so, by Lemma 188, $\overline{\mathtt{f}}$ also terminates;
- the quasi-interpretation for `f` is uniform, so, by Proposition 191, $\overline{\mathtt{f}}$ admits a quasi-interpretation;
- `f` is linear, so $\overline{\mathtt{f}}$ is linear too.

So, by Proposition 193, we can deduce that $\overline{\mathtt{f}}$ is strongly polytime. Therefore `f` is blindly polytime. □

5.2 Bellantoni–Cook programs

To show an example application of Theorem 194 to a simple class of programs, we will consider the class of Bellantoni–Cook safe recursive programs [BC92]. Of course, this is a very particular case, but we think it is interesting since this class is an important reference in the implicit computational complexity literature.

Let BC be the class of Bellantoni–Cook programs written in a Term Rewriting System framework as in [Moy03]. Function arguments are classified as either *safe* or *normal* arguments – recurrences can only occur over normal arguments and their result can only be used in a safe position. We use here a semi-colon to distinguish between normal (on the left) and safe (on the right) parameters.

Definition 195 (Bellantoni–Cook programs)**.** The class BC is the smallest class of programs containing the initial functions:
Constant: $\mathbf{0}$
Successors: $\mathbf{s}_i(x), i \in \{0, 1\}$
Projection: $\pi_j^{n,m}(x_1, \cdots, x_n; x_{n+1}, \cdots, x_m) \to x_j$

Predecessor:
$$\mathtt{p}(;\mathbf{0}) \to \mathbf{0}$$
$$\mathtt{p}(;\mathbf{s}_i(x)) \to x$$

Conditional:

$$\begin{aligned}\mathtt{C}(;\mathbf{0},x,y) &\to x\\ \mathtt{C}(;\mathbf{s}_0(z),x,y) &\to x\\ \mathtt{C}(;\mathbf{s}_1(z),x,y) &\to y\end{aligned}$$

and closed by:

Safe recursion:

$$\begin{aligned}\mathtt{f}(\mathbf{0},x_1,\cdots,x_n;y_1,\cdots,y_m) \to&\mathtt{g}(x_1,\cdots,x_n;y_1,\cdots,y_m)\\ \mathtt{f}(\mathbf{s}_i(z),x_1,\cdots,x_n;y_1,\cdots,y_m) \to&\mathtt{h}_i(z,x_1,\cdots,x_n;y_1,\cdots,y_m,\\ &\mathtt{f}(z,x_1,\cdots,x_n;y_1,\cdots,y_m)), i\in\{0,1\}\end{aligned}$$

with $\mathtt{g},\mathtt{h}_i \in$ BC (previously defined)
Safe composition:

$$\begin{aligned}\mathtt{f}(x_1,\cdots,x_n;y_1,\cdots,y_m) \to \mathtt{g}(&\mathtt{h}_1(x_1,\cdots,x_n;),...,\mathtt{h}_p(x_1,\cdots,x_n;);\\ &\mathtt{l}_1(x_1,\cdots,x_n;y_1,\cdots,y_m),...,\mathtt{l}_q(x_1,\cdots,x_n;y_1,\cdots,y_m))\end{aligned}$$

with $\mathtt{g},\mathtt{h}_i,\mathtt{l}_j \in$ BC.

It is easy to see that any BC program terminates by PPO and is linear.

Proposition 196 (Quasi-interpretations for BC-programs). *A BC-program admits the following quasi-interpretation:*

$$\begin{aligned}(\!|\mathbf{0}|\!) &= 1\\ (\!|\mathbf{s}_i|\!)(X) &= X+1\\ (\!|\pi|\!)(X_1,\cdots,X_{n+m}) &= \max(X_1,\cdots,X_{n+m})\\ (\!|p|\!)(X) &= X\\ (\!|C|\!)(X,Y,Z) &= \max(X,Y,Z).\end{aligned}$$

For functions defined by safe recursion or composition,

$$(\!|f|\!)(X_1,\cdots,X_n;Y_1,\cdots,Y_m) = q_f(X_1,\cdots,X_n) + max(Y_1,\cdots,Y_m)$$

with q_f defined as follows:

- *if f is defined by safe recursion, then*

$$q_f(A,X_1,\cdots,X_n) = A(q_{h_0}(A,X_1,\cdots,X_n) + q_{h_1}(A,X_1,\cdots,X_n)) + q_g(X_1,\cdots,X_n);$$

- *if f is defined by safe composition, then*

$$qf(X_1,\cdots,X_n) = q_g(q_{h_1}(X_1,\cdots,X_n),...,q_{h_p}(X_1,\cdots,X_n)) + \sum_i q_{l_i}(X_1,\cdots,X_n).$$

Proof. It is easy to check that all the conditions of Definitions 179 and 180 are satisfied. □

6 Semi-lattice of quasi-interpretations

The study of necessary conditions on programs satisfying the P-criterion has drawn our attention to uniform quasi-interpretations. This suggests we should consider quasi-interpretations with fixed assignments for constructors and examine their properties as a class.

Definition 197 (Compatible assignments). Let `f` be a program and $(\!|\cdot|\!)_1$, $(\!|\cdot|\!)_2$ be two assignments for `f`. We say that they are *compatible* if for any constructor symbol $\mathbf{c}$ we have $(\!|\mathbf{c}|\!)_1 = (\!|\mathbf{c}|\!)_2$. A family of assignments for `f` is compatible if its elements are pairwise compatible. We will use similar definitions for quasi-interpretations.

Each choice of assignments for constructors thus defines a *maximal compatible family* of quasi-interpretations for a program `f`: all quasi-interpretations for `f` that take these values on $\mathcal{C}$. We consider the extensional order $\leq$ on assignments defined by $(\!|\cdot|\!)_1 \leq (\!|\cdot|\!)_2$ if and only if

$$\forall b\in\mathcal{C}\bigcup\mathcal{F}, \forall\vec{x}\in(\mathbb{R}^+)^k, (\!|b|\!)_1(\vec{x}) \leq (\!|b|\!)_2(\vec{x}).$$

Given two compatible assignments $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$, we use $(\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$ to denote the assignment $(\!|\cdot|\!)_0$ defined by

$$\begin{aligned}\forall\mathbf{c}\in\mathcal{C}, (\!|\mathbf{c}|\!)_0 =& (\!|\mathbf{c}|\!)_1 = (\!|\mathbf{c}|\!)_2\\ \forall\mathrm{g}\in\mathcal{F}, (\!|\mathrm{g}|\!)_0 =& (\!|\mathrm{g}|\!)_1 \wedge (\!|\mathrm{g}|\!)_2\end{aligned}$$

where $\alpha\wedge\beta$ denotes the greatest lower bound of $\{\alpha,\beta\}$ in the point-wise order. Then we have the following proposition.

Proposition 198. *Let f be a program and let $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$ be two compatible quasi-interpretations for it. Then $(\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$ is also a quasi-interpretation for f.*

To establish this Proposition, we need some preliminary Lemmas. We will continue to use the denotation $(\!|\cdot|\!)_0 = (\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$.

Lemma 199. *For any g of f we have that $(\!|g|\!)_0$ is monotone and satisfies the subterm property.*

Proof. To prove monotonicity, we will assume $\vec{x} \leq \vec{y}$ for the product ordering. Then, for $i = 1$ or 2

$$(\!|\mathrm{g}|\!)_0(\vec{x}) = (\!|\mathrm{g}|\!)_1(\vec{x}) \wedge (\!|\mathrm{g}|\!)_2(\vec{x}) \leq (\!|\mathrm{g}|\!)_i(\vec{x}) \leq (\!|\mathrm{g}|\!)_i(\vec{y}),$$

using the monotonicity of $(\!|\mathrm{g}|\!)_i$. Since this is true for $i = 1$ and 2, we have

$$(\!|\mathrm{g}|\!)_0(\vec{x}) \leq (\!|\mathrm{g}|\!)_1(\vec{y}) \wedge (\!|\mathrm{g}|\!)_2(\vec{y}) = (\!|\mathrm{g}|\!)_0(\vec{y}).$$

It is also easy to check that $(\!|\mathrm{g}|\!)_0$ satisfies the subterm property. □

Lemma 200. *Let t be a term. We have $(\!|t|\!)_0 \leq (\!|t|\!)_i$ for $i = 1, 2$.*

Proof. The proof is by induction on t, using the definition of $(\!|\mathrm{g}|\!)_0$ and $(\!|\mathbf{c}|\!)_0$, together with the monotonicity property of Lemma 199. □

Lemma 201. *Let $g(p_1, \cdots, p_n) \to t$ be an equation of the program f. We have*

$$(\!|g|\!)_0 \circ ((\!|p_1|\!)_0, \cdots, (\!|p_n|\!)_0) \geq (\!|t|\!)_0.$$

Proof. As patterns only contain constructor and variable symbols, by the definition of $(\!|\cdot|\!)_0$, if p is a pattern, we have $(\!|p|\!)_0 = (\!|p|\!)_1 = (\!|p|\!)_2$. Let $i = 1$ or 2. We have

$$\begin{aligned} (\!|\mathrm{g}|\!)_i((\!|p_1|\!)_i(\vec{x}), \cdots, (\!|p_n|\!)_i(\vec{x})) &\geq (\!|t|\!)_i(\vec{x}) \quad \text{(because } (\!|\cdot|\!)_i \text{ is a quasi-interpretation)} \\ &\geq (\!|t|\!)_0(\vec{x}). \quad \text{(from Lemma 200)} \end{aligned}$$

So

$$(\!|\mathrm{g}|\!)_i((\!|p_1|\!)_0(\vec{x}), \cdots, (\!|p_n|\!)_0(\vec{x})) \geq (\!|t|\!)_0(\vec{x}),$$

since $(\!|p_j|\!)_i = (\!|p_j|\!)_0$. We write

$$\vec{y} = ((\!|p_1|\!)_0(\vec{x}), \cdots, (\!|p_n|\!)_0(\vec{x})).$$

Since $(\!|\mathrm{g}|\!)_1(\vec{y}) \geq (\!|t|\!)_0(\vec{x})$ and $(\!|\mathrm{g}|\!)_2(\vec{y}) \geq (\!|t|\!)_0(\vec{x})$, by the definition of $(\!|\mathrm{g}|\!)_0$, we get $(\!|\mathrm{g}|\!)_0(\vec{y}) \geq (\!|t|\!)_0(\vec{x})$, which concludes the proof. □

We can now proceed with the proof of Proposition 198.

Proof of Proposition 198. We need to check that the conditions of Definitions 179 and 180 of assignment and quasi-interpretation, respectively, are satisfied. Observe that Lemma 199 ensures that $(\!|\cdot|\!)_0$ satisfies the Weak Monotonicity and Subterm conditions, and Lemma 201 ensures it satisfies the condition with respect to the equations of the program. The Additivity condition is satisfied because $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$ are compatible and satisfy this condition by assumption. Finally, the Polynomial condition is satisfied because if $(\!|b|\!)_1$ and $(\!|b|\!)_2$ are bounded by polynomials, then so is $(\!|b|\!)_1 \wedge (\!|b|\!)_2$. Therefore, $(\!|\cdot|\!)_0$ is a quasi-interpretation. □

Proposition 202. *Let f be a program and $\mathcal{Q}$ be a non-empty family of compatible quasi-interpretations for f. Then $\wedge_{(\!|\cdot|\!)\in\mathcal{Q}}(\!|\cdot|\!)$ is a quasi-interpretation for f. Therefore, maximal compatible families of quasi-interpretations for f have an inferior semi-lattice structure for $\leq$.*

Proof. It is sufficient to generalise Lemmas 199 and 201 to the case of an arbitrary family $\mathcal{Q}$ and to apply the same argument as in the proof of Proposition 198. □

Remark 203. Note that if, starting from two quasi-interpretations $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$, we define the assignment $(\!|\cdot|\!) = (\!|\cdot|\!)_1 \vee (\!|\cdot|\!)_2$ in a similar way to $(\!|\cdot|\!)_1 \wedge (\!|\cdot|\!)_2$ with the point-wise lowest upper bound, then, in general, $(\!|\cdot|\!)$ is not a quasi-interpretation. Indeed, the following counter-example shows that the statement of Proposition 198 is not satisfied in this case.

Take $\mathtt{f}, \mathtt{g}, \mathtt{h} \in \mathcal{F}$, $\mathbf{c} \in \mathcal{C}$ and the equation

$$\mathtt{g}(\mathbf{c}(x)) \to \mathtt{f}(\mathtt{h}(\mathtt{g}(x))).$$

Now consider $(\!|\cdot|\!)_1$ and $(\!|\cdot|\!)_2$ given by

$$\begin{aligned}
(\!|\mathtt{g}|\!)_1(x) &= x\\
(\!|\mathtt{f}|\!)_1(x) &= x+1\\
(\!|\mathtt{h}|\!)_1(x) &= x\\
(\!|\mathtt{g}|\!)_2(x) &= x\\
(\!|\mathtt{f}|\!)_2(x) &= x\\
(\!|\mathtt{h}|\!)_2(x) &= x+1.
\end{aligned}$$

Thus we have,

$$\begin{aligned}
(\!|\mathtt{g}|\!)(x) &= x\\
(\!|\mathtt{f}|\!)(x) &= x+1\\
(\!|\mathtt{h}|\!)(x) &= x+1.
\end{aligned}$$

Hence, $(\!|\mathtt{g}(c(x))|\!) = x+1$, but $(\!|\mathtt{f}(h(\mathtt{g}(x)))|\!) = x+2$.

7 Extending the P-criterion

Blind abstraction suggests we consider not only the PPO ordering from the P-criterion, but also an extension that is invariant under the blinding map, *viz.* the EPPO ordering (see Subsection 4.3). It is thus natural to ask whether EPPO enjoys the same property as PPO. We prove in this section that if we consider programs over strings, with EPPO we can still bound the size of the call-dag, and thus generalise the P-criterion. We will also consider the *bounded-value property*, which is an extension of the notion of QI. Here we bound the number of nodes in the call-dag with a given precedence, and then Proposition 177 bounds the total number of nodes in the call-dag.

In this Section we restrict consideration to deterministic programs handling strings over a finite alphabet (that is, words), so we take $\mathcal{C} = \{\mathbf{s}_0, \cdots, \mathbf{s}_m, \mathbf{nil}\}$, where each $\mathbf{s}_i$ has arity 1 and **nil** has arity 0. We call these programs over unary constructors.

Example 204. We will first show that EPPO is (intensionally) strictly more powerful than PPO. It is quite obvious that any program terminating by PPO also terminates by EPPO. The following program works overs binary integers (that is, there are two successors) represented with least significant bit first (that is, $\mathbf{s}_0\mathbf{s}_1\mathbf{nil}$ corresponds to the integer 10, in other words, 2). The program computes (in a slightly contrived way) x^y using a 'fast exponentiation algorithm' in base 4, that is, using the recurrence $x^{4y} = ((x^y)^2)^2$, and similar rules for other cases (we will omit the base cases here). The rules for multiplication and squaring are quite standard, so they are not included here. The equation in parentheses following each rule explains what it means.

$$\begin{array}{lcl}
\mathtt{pow}(x, \mathbf{s}_0\mathbf{s}_0 y) \to & \mathtt{sq}(\mathtt{sq}(\mathtt{pow}(x,y))) & (x^{4y} = ((x^y)^2)^2)\\
\mathtt{pow}(x, \mathbf{s}_1\mathbf{s}_0 y) \to & \mathtt{mult}(x, \mathtt{sq}(\mathtt{pow}(x, \mathbf{s}_0 y))) & (x^{4y+1} = (x^{2y})^2 \times x)\\
\mathtt{pow}(x, \mathbf{s}_0\mathbf{s}_1 y) \to & \mathtt{sq}(\mathtt{mult}(x, \mathtt{pow}(x, \mathbf{s}_0 y))) & (x^{4y+2} = (x^{2y} \times x)^2)\\
\mathtt{pow}(x, \mathbf{s}_1\mathbf{s}_1 y) \to & \mathtt{mult}(x, \mathtt{sq}(\mathtt{mult}(x, \mathtt{pow}(x, \mathbf{s}_0 y)))) & (x^{4y+3} = (x^{2y} \times x)^2 \times x)
\end{array}$$

This program does not terminate by PPO since the last rule is not ordered by PPO (because $\mathbf{s}_0 y$ is not comparable with $\mathbf{s}_1\mathbf{s}_1 y$). However, the blind abstraction of this program does terminate by PPO, and the program itself terminates by EPPO. Indeed, with a fair precedence, $\mathbf{s}_0$ and $\mathbf{s}_1$ have the same precedence, hence $\mathbf{s}_0 y \prec_{eppo} \mathbf{s}_1\mathbf{s}_1 y$.

Fact 3. Since we are working over words (unary constructors), patterns are either constructor terms (that is, words), or have the form $p = \mathbf{s}_1(\mathbf{s}_2 \cdots \mathbf{s}_n(x) \cdots)$ where the $\mathbf{s}_i$ for $1 \le i \le n$ are constructors. In the second case, we will write $p = \Omega(x)$ with $\Omega = \mathbf{s}_1\mathbf{s}_2 \cdots \mathbf{s}_n$.

The length of a pattern is the length of the corresponding word, so $|p| = |\Omega|$.

Proposition 205. *In a program terminating by PPO or EPPO, the only calls at the same precedence that can occur are of the form*

$$\mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}_1(q_1^1, \cdots, q_m^1), \ldots, \mathtt{g}_p(q_1^p, \cdots, q_l^p)]$$

where $C[\cdot]$ *is some context,* $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}_k$*, and* p_i *and* q_j^k *are patterns. Moreover, each variable appearing in a* q_i^k *appears in* p_i*.*

Proof. The claim is a direct consequence of the termination ordering. □

Since we will only consider individual calls, we will put in the context all but one of the $\mathtt{g}_k$: $\mathtt{f}(p_1,\cdots,p_n) \to C[\mathtt{g}(q_1,\cdots,q_m)]$.

Definition 206 (Production size)**.** Let $\mathtt{f}$ be a program terminating by EPPO. Let

$$\mathtt{f}(p_1,\cdots,p_n) \to C[\mathtt{g}(q_1,\cdots,q_m)]$$

be a call in it where $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$. The *production size* of this call is $\max_{1\le i\le m}\{|q_i|\}$. The production size of an equation is the greatest production size of any call (at the same precedence) in it. That is, if we have an equation

$$e = \mathtt{f}(p_1,\cdots,p_n) \to C[\mathtt{g}_1(q_1^1,\cdots,q_m^1),\ldots,\mathtt{g}_p(q_1^p,\cdots,q_l^p)]$$

where $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}_k$ for $1 \le k \le p$, then its production size is

$$K_e = \max_{1\le k\le p, 1\le j\le l} |q_j^k|.$$

The production size of a function symbol is the maximum production size of any equation defining a function with the same precedence:

$$K_{\mathtt{h}} = \max_{\mathtt{g}\approx_{\mathcal{F}}\mathtt{h}} \max_{e=\mathtt{g}(\ldots)\to t} K_e.$$

Definition 207 (Normality)**.** Let $\mathtt{f}$ be a program. A function symbol $\mathtt{h}$ in it is *normal* if the patterns in the definitions of functions with the same precedence are bigger than its production size:

$$\forall \mathtt{g} \approx_{\mathcal{F}} \mathtt{h}, \forall \mathtt{g}(q_1,\cdots,q_m) \to r \in \mathcal{E}, |p_i| K_{\mathtt{h}}.$$

A program $\mathtt{f}$ is *normal* if all the function symbols in it are normal.

If a program is normal, this means that during recursive calls every constructor produced at a given moment will be consumed by the following pattern matching.

Lemma 208. *An EPPO-program can be changed into an equivalent normal program (* normalised*) with at most an exponential growth* in the size of the program.

The exponential is in the difference between the size of the biggest production and the size of the smallest pattern (with respect to each precedence).

Proof sketch. The idea is to extend the small pattern matchings so that their length reaches the length of the biggest production. This is illustrated by the following example:

$$\begin{aligned} \mathtt{f}(\mathtt{s}_1(\mathtt{s}_1(\mathtt{s}_1(x)))) &\to \mathtt{f}(\mathtt{s}_0(\mathtt{s}_0(x))) \\ \mathtt{f}(\mathtt{s}_0(x)) &\to \mathtt{f}(x). \end{aligned}$$

In this case, the biggest production has size 2, but the shortest pattern matching only has size 1. We can normalise the program as follows:

$$\begin{aligned} \mathtt{f}(\mathtt{s}_1(\mathtt{s}_1(\mathtt{s}_1(x)))) &\to \mathtt{f}(\mathtt{s}_0(\mathtt{s}_0(x))) \\ \mathtt{f}(\mathtt{s}_0(\mathtt{s}_0(x))) &\to \mathtt{f}(\mathtt{s}_0(x)) \\ \mathtt{f}(\mathtt{s}_0(\mathtt{s}_1(x))) &\to \mathtt{f}(\mathtt{s}_1(x)). \end{aligned}$$

Even if the process does extend productions as well as patterns, and increases the number of equations, it does terminate because only the smallest patterns, hence the smallest productions (due to termination ordering) are extended in this way. □

Notice that the exponential growth in Lemma 208 is indeed in the initial size of the program and does not depend on the size of any input. Since the size of the call-dag is bounded by the size of the inputs, this does not hamper the polynomial bound on execution time. The normalisation process of Lemma 208 preserves termination by PPO and EPPO and does not decrease the time complexity. Hence, bounding the time complexity of the normalised program is sufficient to bound the time complexity of the initial program. We will only consider normal programs in the following.

Let $\mathtt{f}$ be a program and $\mathtt{g}$ be a function. We will enumerate all the occurrences of symbols of same precedence as $\mathtt{g}$ in the right-hand side of $\mathtt{f}$ and label them $\mathtt{g}_1,\cdots,\mathtt{g}_n$. This is simply an enumeration, and not a renaming of the symbols, and if a given symbol appears several times (in several equations or in the same one), it will be given several labels (one for each occurrence) with this enumeration. Now, a path in the call-dag staying only at the same precedence as $\mathtt{g}$ is canonically identified by a word over $\{\mathtt{g}_1,\cdots,\mathtt{g}_n\}$. We -write $\eta \overset{\omega}{\rightsquigarrow} \mu$, where ω is a word, to denote the fact that the state η is an ancestor of μ and ω is the path between them.

Lemma 209. *Let $\eta_1 = (f, v_1, \cdots, v_n, v)$ and $\eta_2 = (g, u_1, \cdots, u_m, u)$ be two states such that $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ and both function symbols have the same precedence. Then $|v_i| \geq |u_i|$ for $1 \leq i \leq n$ and there exists j such that $|v_j| > |u_j|$.*

Proof. The claim is a consequence of the termination proof by EPPO. □

The following Proposition is a consequence of Lemma 209.

Proposition 210. *Let $\eta = (f, v_1, \cdots, v_n, v)$ be a state. Any branch in the call-dag starting from η has at most $n \times (max_{1 \leq i \leq n}|v_i|)$ nodes with the same precedence as f.*

Lemma 211. *Suppose we have labels α, β and γ, and nodes in the call dag such that $\eta \overset{\alpha}{\rightsquigarrow} \eta_1 \overset{\beta}{\rightsquigarrow} \mu_1 \overset{\gamma}{\rightsquigarrow} \xi_1$ and $\eta \overset{\beta}{\rightsquigarrow} \eta_2 \overset{\alpha}{\rightsquigarrow} \mu_2 \overset{\gamma}{\rightsquigarrow} \xi_2$. Then $\xi_1 = \xi_2$.*

Proof. Let us write $\xi_1 = (\mathtt{f}, s_1, \cdots, s_n, s)$, $\xi_2 = (\mathtt{g}, t_1, \cdots, t_m, t)$. Since labels are unique, we have $\mathtt{f} = \mathtt{g}$, and thus $n = m$. It is sufficient to show that the ith components are the same for $1 \leq i \leq n$. We use v, v_1, u_1, p_1, v_2, u_2, q_2 to denote the ith parameters of η, η_1, μ_1, ξ_1, η_2, μ_2, ξ_2, respectively. Since we are working on words, an equation e has, with respect to the ith argument, the form

$$\mathtt{f}(\ldots, \Omega_e(x), \ldots) \to C[\mathtt{g}(\ldots, \Upsilon_e(x), \ldots)],$$

and the fact that the program is normal implies that $|\Upsilon_e| \leq |\Omega_e|$ (previous Lemma). Using a similar denotation for the equation corresponding to the label α, we have

$$\mathtt{f}(\ldots, \Omega_\alpha(x), \ldots) \to C[\mathtt{g}(\ldots, \Upsilon_\alpha(x), \ldots)].$$

In this case, we write $\Omega_\alpha(x) \hookrightarrow \Upsilon_\alpha(x)$ as a short-hand notation, and define Ω_β, Υ_β, Ω_γ, Υ_γ similarly. So, in our case, we have

$$\begin{aligned}
v &= \Omega_\alpha(x_1) \hookrightarrow \Upsilon_\alpha(x_1) = v_1 = \Omega_\beta(y_1) \hookrightarrow \Upsilon_\beta(y_1) \\
&= u_1 = \Omega_\gamma(z_1) \hookrightarrow \Upsilon_\gamma(z_1) = p_1 \\
v &= \Omega_\beta(x_2) \hookrightarrow \Upsilon_\beta(x_2) = v_1 = \Omega_\alpha(y_2) \hookrightarrow \Upsilon_\alpha(y_2) \\
&= u_2 = \Omega_\gamma(z_2) \hookrightarrow \Upsilon_\gamma(z_2) = q_2
\end{aligned}$$

Because of normalisation, $|\Omega_\beta| \geq |\Upsilon_\alpha|$. Hence, y_1 is a suffix of x_1, which is itself a suffix of v. Similarly, z_1 and z_2 are suffixes of v. Since they are both suffixes of the same word, it is sufficient to show that they have the same length in order to show that they are identical:

$$\begin{aligned}
|x_1| =& |v| - |\Omega_\alpha| \\
|y_1| =& |v| - |\Omega_\beta| = |v| - |\Omega_\alpha| + |\Upsilon_\alpha| - |\Omega_\beta| \\
|z_1| =& |v| - |\Omega_\alpha| + |\Upsilon_\alpha| - |\Omega_\beta| + |\Upsilon_\beta| - |\Omega_\gamma| \\
|z_2| =& |v| - |\Omega_\beta| + |\Upsilon_\beta| - |\Omega_\alpha| + |\Upsilon_\alpha| - |\Omega_\gamma|.
\end{aligned}$$

Hence $z_1 = z_2$, so $q_1 = q_2$. □

We say that two words have the *same commutative image* if there exists a permutation on the order of letters mapping the first word to the second.

Proposition 212. *Let ω_1, ω_2 be words and α be a label such that $\eta \overset{\omega_1}{\rightsquigarrow} \mu_1 \overset{\alpha}{\rightsquigarrow} \xi_1$ and $\eta \overset{\omega_2}{\rightsquigarrow} \mu_2 \overset{\alpha}{\rightsquigarrow} \xi_2$. If ω_1 and ω_2 have the same commutative image, then $\xi_1 = \xi_2$.*

Proof. This is a generalisation of the previous proof, and not an induction on it. If $\omega_1 = \alpha_1, \cdots, \alpha_n$, then the size in the last node is

$$|x'| = |q| = \sum |\Omega_{\alpha_i}| + \sum |\Upsilon_{\alpha_i}| - |\Omega_\alpha|,$$

which only depends on the commutative image of ω_1. □

So, when using the semantics with memoisation, the number of nodes (at a given precedence) in the call-dag is roughly equal to the number of paths *modulo commutativity.* So any path can be associated with the vector whose components are the number of occurrences of the corresponding label in it.

Proposition 213. *Let $\mathcal{T}$ be a call-dag and $\eta = (f, v_1, \cdots, v_n, v)$ be a node in it. Let $I = n \times (\max |v_j|)$ and M be the number of function symbols with the same precedence as f. The number of descendants of η in $\mathcal{T}$ with the same precedence as f is bounded by $(I + 1)^M$, which is a polynomial in $|\eta|$.*

Proof. Any descendant of η with the same precedence can be identified with a word over $\{\mathtt{f}^1, \cdots, \mathtt{f}^M\}$. By Proposition 210, we know that these words have length at most I, and by Proposition 212, we know that it is sufficient to consider these words modulo commutativity. Modulo commutativity, words can be identified with vectors with as many components as the number of letters in the alphabet and whose sum of components is equal to the length of the word. Let D_i^n be the number of elements from $\mathbb{N}^n$ whose sum is i. This is the number of words of length i over an n-ary alphabet modulo commutativity. It is clear that $D_{i-1}^n \leq D_i^n$ (take all the n-tuples whose sum is $i-1$, then add 1 to the first component and you obtain D_{i-1}^n different n-tuples whose sum of components is i). Now, to count D_i^n, we choose a value j for the first component, then find $n-1$ components whose sum is $i-j$. There are D_{i-j}^{n-1} such elements.

$$D_i^n = \sum_{0\leq j\leq i} D_{i-j}^{n-1} = \sum_{0\leq j\leq i} D_j^{n-1} \leq (i+1)\times D_i^{n-1} \leq (i+1)^{n-1}\times D_i^1 \leq (i+1)^n.$$

This concludes the proof. □

Definition 214 (Bounded values)**.** A program $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}\rangle$ has polynomially bounded values if and only if for every n-ary function symbol $\mathtt{g} \in \mathcal{F}$, there is a polynomial $p_{\mathtt{g}} : \mathbb{N} \to \mathbb{N}$ such that for every state η' appearing in a call tree for $\eta = (\mathtt{g}, v_1, \cdots, v_n)$, we have $|\eta'| \leq p_{\mathtt{g}}(|\eta|)$.

Proposition 215. *Any deterministic program f over unary constructors that*

1. *terminates by EPPO and*
2. *has polynomially bounded values*

is executable in polynomial time with the memoisation semantics. Conversely, any function in FP can be represented by a program of this kind.

Since admitting a QI implies having polynomially bounded values, we have that a program over unary constructors terminating by EPPO and admitting a QI is polytime.

Proof. Proposition 213 bounds the size of the call dag by rank. The bounded-values property bounds the size of nodes in the call dag. So we can apply Proposition 166 to bound the size of any derivation. The converse is obtained from the P-criterion (Theorem 184). □

Theorem 216. *Let f be a deterministic program over unary constructors terminating by EPPO. Then the following two conditions are equivalent:*

1. *f has polynomially bounded values;*
2. *f is polytime in the call-by-value semantics with memoisation.*

Note that the interest in this last result is rather theoretical, as we do not provide here a new way to determine if a program admits polynomially bounded values (admitting a QI is a sufficient condition). However, it does exactly delineate those EPPO programs over unary constructors that are polytime in the semantics with memoisation.

Proof. The implication (1) ⇒ (2) is given by Proposition 215 For the converse, it is sufficient to observe that a state appearing in the call dag also appears in the final cache. Since the size of any term in the cache is bounded by the size of the proof (because we need to perform as many Constructor rules as needed to construct the term), it is polynomially bounded because the program is polytime. □

In fact, we conjecture that the statements of Proposition 215 and Theorem 216 hold not only for programs with unary constructors, but also for programs with arbitrary constructors, but we leave further investigations of this question for future works.

Remark 217. The notion of a sup-interpretation [MP09] has been introduced to generalise quasi-interpretations. However, not all programs admitting a sup-interpretation satisfy the bounded-values condition. Indeed sup-interpretations only relate the interpretation of a term to the interpretation of its normal form (see [MP09], Section 4.2, Definition 4.7), while the notion of a program with polynomially bounded values requires that, in addition, all intermediates states of the call-tree are also bounded. As a counter-example, consider the program defined by

$$\begin{aligned}
\mathtt{f}(0, y) &\to 0\\
\mathtt{f}(\mathtt{s}x, y) &\to \mathtt{f}(x, \mathtt{db}(y))\\
\mathtt{db}(0) &\to 0\\
\mathtt{db}(\mathtt{s}(x)) &\to \mathtt{s}(\mathtt{s}(\mathtt{db}(x)))
\end{aligned}$$

This program can be given a sup-interpretation for all its symbols,

$$\begin{aligned}\theta(\mathtt{f})(X,Y) =& X\\ \theta(\mathtt{db})(X) =& 2X,\end{aligned}$$

and, as expected,

$$\begin{aligned}\theta(\mathtt{s})(X) =& X+1\\ \theta(0) =& 0\end{aligned}$$

However, the final call of $\mathtt{f}$ during the execution of $\mathtt{f}(\mathtt{s}^n(0),0)$ is $\mathtt{f}(0,\mathtt{s}^m(0))$ with $m=2^n$, so it is exponentially large, and the program does not have polynomially bounded values.

To handle this, sup-interpretations come together with the notions of fraternities (to detect recursive calls) and weight (which, having the subterm property of QI, tame the recursive calls). As stated in [MP09], Theorem 5.3, programs with the *quasi-friendly* property (that is, sup-interpretation plus other conditions) also have the bounded-values property.

On the other hand, any program satisfying the bounded-values properties admits a sup-interpretation, that is to say, a sup-interpretation can be attributed to all its function symbols. More precisely, these sup-interpretations can be chosen to be polynomials. It is sufficient for that to take for any function symbol $\mathtt{g}$ of the program a polynomial $p_{\mathtt{g}}$ as given by Definition 214.

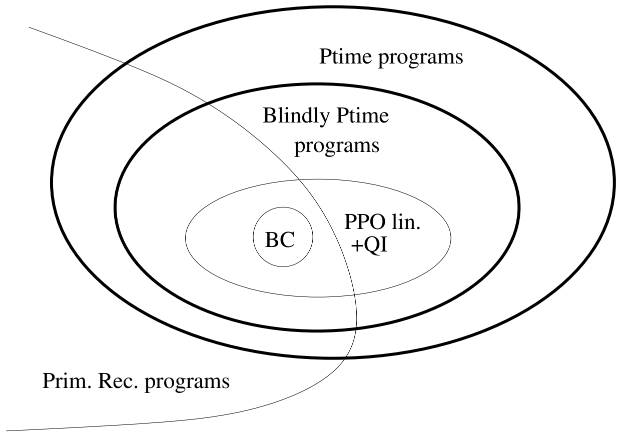


Figure 10: Subclasses of programs

8 Conclusions

We have introduced blind abstractions of first-order functional programs and exploited them to help us gain an understanding of the intensional expressive power of quasi-interpretations. In particular, we have shown that linear programs that admit product path-ordering and uniform quasi-interpretation are necessarily blindly polytime. This has suggested an extension of product path-ordering for which we have proved that the P-criterion still holds. Finally, we have delineated a property (polynomially bounded values) that describes those deterministic programs admitting product path-orderings that are executable in polynomial time.

We have introduced the notion of blindly polytime programs and shown that this notion can be used to compare ICC systems in a more general way than by exhibiting examples caught by one and not by the other. The comparison between the Bellantoni–Cook BC class and our 'blind P-criterion' is illustrated in Figure 10.

Further work includes investigations into the conditions characterising the class of programs captured by quasi-interpretations. In particular, the question is still open as to whether being blindly polytime is a necessary and sufficient condition for a program to have a (polynomially bounded) sup-interpretation or to be quasi-friendly [MP09], provided some sort of path-ordering for it exists.

Another direction is to try to find notions similar to blindly polytime that would help comparing and classifying ICC systems. We have seen that being blindly polytime is a condition independent of the primitive recursion scheme. In order to better understand the relative power of ICC systems, it would be useful to have other similar classes of programs and thus complete the picture in Figure 10.

Part II

Equivalences between Programs

Chains, Antichains, and Complements in Infinite Partition Lattices

James Emil Avery, Jean-Yves Moyen, Pavel Ruzicka, Jakob Grue Simonsen

Algebra Universalis, under revision

Abstract:

We consider the partition lattice Π_κ on any set of transfinite cardinality κ and properties of Π_κ whose analogues do not hold for finite cardinalities. Assuming the Axiom of Choice we prove: (I) the cardinality of any maximal well-ordered chain is always exactly κ; (II) there are maximal chains in Π_κ of cardinality $> \kappa$; (III) if, for every cardinal $\lambda < \kappa$, we have $2^\lambda < 2^\kappa$, there exists a maximal chain of cardinality $< 2^\kappa$ (but $\geq \kappa$) in Π_{2^κ}; (IV) every non-trivial maximal antichain in Π_κ has cardinality between κ and 2^κ, and these bounds are realized. Moreover we can construct maximal antichains of cardinality $\max(\kappa, 2^\lambda)$ for any $\lambda \leq \kappa$; (V) all cardinals of the form κ^λ with $0 \leq \lambda \leq \kappa$ occur as the number of complements to some partition $\mathcal{P} \in \Pi_\kappa$, and only these cardinalities appear. Moreover, we give a direct formula for the number of complements to a given partition; (VI) Under the Generalized Continuum Hypothesis, the cardinalities of maximal chains, maximal antichains, and numbers of complements are fully determined, and we give a complete characterization.

Let κ be a cardinal and let S be a set of cardinality κ. The set of partitions of S forms a lattice when endowed with the binary relation $\leq$, called *refinement*, defined by $\mathcal{P} \leq \mathcal{Q}$ if and only if each block of $\mathcal{P}$ is a subset of a block of $\mathcal{Q}$. This lattice is called the *partition lattice* on S, and is denoted $\Pi(S)$. By the standard correspondence between partitions and equivalence relations, it follows that $\Pi(S)$ is isomorphic to the lattice $\mathrm{Equ}(S)$ of equivalence relations on S ordered by set inclusion on $S \times S$.

As the particulars of S do not affect the order-theoretic properties of $\Pi(S)$ we shall without loss of generality restrict our attention to the lattice $\Pi_\kappa = \Pi(\kappa)$. Initiated by a seminal paper by Ore [Ore42], many of the properties of Π_κ that hold for arbitrary cardinals κ are well-known. Indeed, it is known that Π_κ is complete, matroid (hence atomistic and semimodular), non-modular (hence non-distributive) for $\kappa \geq 4$, relatively complemented (hence complemented), and simple [Bir40, §8-9], [RS92], [Grä03, Sec. IV.4].

For properties depending on κ, only a few results exist in the literature for infinite κ. Czédli has proved that if there is no inaccessible cardinal $\leq \kappa$ then the following holds: If $\kappa \geq 4$, Π_κ is generated by four elements [Czé96a], and if $\kappa \geq 7$, Π_κ is (1+1+2)-generated [Czé99] (for $\kappa = \aleph_0$, slightly stronger results hold [Czé96b]). It appears that no further results are known, beyond those holding for all cardinalities, finite or infinite. The aim of the present work is to prove a number of results concerning Π_κ that depend on κ being an infinite cardinal.

1 Preliminaries and notation

We work in ZF with the Axiom of Choice (AC). As usual, a set S is *well-ordered* if and only if it is totally ordered and every non-empty subset of S has a least element. Throughout the paper, we use von Neumann's characterization of ordinals: a set S is an ordinal if and only if it is strictly well-ordered by $\subsetneq$ and every element of S is a subset of S. The *order type* of a well-ordered set S is the (necessarily unique) ordinal α that is order-isomorphic to S. Cardinals and ordinals are denoted by Greek letters $\alpha, \beta, \delta, \gamma, \ldots$ for ordinals and $\kappa, \lambda, \ldots$ for cardinals. We denote by ω_κ the initial ordinal of κ, and by $|\alpha|$ the cardinality of α. The cardinality of a set is denoted $|S|$ and its powerset is denoted $\mathscr{P}(S)$. For a cardinal κ, we denote by κ^+ its successor and by κ^- its predecessor cardinal. Note that κ^- is defined only if κ is a successor cardinal.

Many standard results on cardinal arithmetic can be found in [HSW99], among other places, and are used frequently throughout the proofs.

Recall that a *chain* in a poset $(\mathbf{P}, \leq)$ is a subset of $\mathbf{P}$ that is totally ordered by $\leq$. Similarly, an *antichain* in $(\mathbf{P}, \leq)$ is a subset of $\mathbf{P}$ such that any two distinct elements of the subset are $\leq$-incomparable. A chain (respectively, antichain) in $(\mathbf{P}, \leq)$ is *maximal* if no element of $\mathbf{P}$ can be added to the chain without losing the property of being a chain (respectively, antichain). Observe that if $\mathbf{P}$ contains a bottom element, $\bot$, or a top element, $\top$, it belongs to any maximal chain. A chain $\mathbf{C}$ in $(\mathbf{P}, \leq)$ is *saturated* if, for any two elements $\mathcal{Q} < \mathcal{S}$ of the chain, there is no element $\mathcal{R} \in \mathbf{P} \setminus \mathbf{C}$ such that $\mathcal{Q} < \mathcal{R} < \mathcal{S}$ and $\mathbf{C} \cup \{\mathcal{R}\}$ is a chain; notably, a chain containing $\bot$ and $\top$ is maximal if and only if it is saturated. We say that a chain is *endpoint-including* if it contains a least and a greatest element, not necessarily equal to $\bot$ and $\top$, respectively. By the Maximal Chain Theorem [Hau14], every chain in a poset is contained in a maximal chain.

We denote partitions (and equivalences) of κ by capital italic Roman letters $\mathcal{P}, \mathcal{Q}, \ldots$, and denote subsets of Π_κ such as chains and antichains by capital boldface letters $\mathbf{C}, \mathbf{D}, \ldots$; If $\mathcal{P} = \{B_\delta\}$ is a partition, we call its elements, B_δ, *blocks*. It is easily seen that $\bot = \{\, \{\gamma\} \;\mid\; \gamma \in \kappa \,\}$ and $\top = \{\kappa\}$; that is, the set of all singleton subsets of κ, respectively the singleton set containing all elements of κ.

As is usual, if $\mathcal{P}, \mathcal{Q} \in \Pi_\kappa$, we write $\mathcal{P} \prec \mathcal{Q}$ if $\mathcal{P} < \mathcal{Q}$ and no $\mathcal{R} \in \Pi_\kappa$ exists such that $\mathcal{P} < \mathcal{R} < \mathcal{Q}$. Furthermore, $\mathcal{P} \preceq \mathcal{Q}$ denotes that either $\mathcal{P} \prec \mathcal{Q}$ or $\mathcal{P} = \mathcal{Q}$.

It follows that $\mathcal{P} \prec \mathcal{Q}$ if and only if $\mathcal{Q}$ can be obtained by merging exactly two distinct blocks of $\mathcal{P}$. If $\mathbf{X}$ is a subset of Π_κ, we write $\mathcal{P} \prec_{\mathbf{X}} \mathcal{Q}$ if $\mathcal{P}, \mathcal{Q} \in \mathbf{X}$ with $\mathcal{P} < \mathcal{Q}$, and there exists no $\mathcal{R} \in \mathbf{X}$ such that $\mathcal{P} < \mathcal{R} < \mathcal{Q}$. A subset $\mathbf{X} \subseteq \Pi_\kappa$ is called *covering* if $\mathcal{P} \prec_{\mathbf{X}} \mathcal{Q}$ implies $\mathcal{P} \prec \mathcal{Q}$.

A block B induces an equivalence relation on κ, defined by $\delta \equiv_B \gamma$ if and only if both $\delta, \gamma \in B$, and a partition $\mathcal{P}$ naturally induces an equivalence relation defined by $\delta \equiv_{\mathcal{P}} \gamma$ if and only if there is a block $B \in \mathcal{P}$ with $\delta \equiv_B \gamma$. Conversely, any equivalence relation corresponds to the partition whose blocks are the maximal sets of equivalent elements. This one-to-one correspondence allows us to consider a partition as its corresponding equivalence relation when convenient, and vice versa.

If $\mathcal{P} \in \Pi_\kappa$ contains exactly one block B with $|B| \geq 2$ and the remaining blocks are all singletons, we call $\mathcal{P}$ a *singular* partition, following Ore [Ore42].

If $\mathbf{C} \subseteq \Pi_\kappa$, then its greatest lower bound $\wedge\mathbf{C}$ is the partition that satisfies $x \equiv_{\wedge\mathbf{C}} y$ if and only if $x \equiv_{\mathcal{P}} y$ for all $\mathcal{P} \in \mathbf{C}$. That is, the blocks of $\wedge\mathbf{C}$ are all the nonempty intersections whose terms are exactly one block from every partition $\mathcal{P} \in \mathbf{C}$. Conversely, its least upper bound $\vee\mathbf{C}$ is the partition such that $\gamma \equiv_{\vee\mathbf{C}} \delta$ if and only if there exists a finite sequence of partitions $\mathcal{P}^1, \ldots, \mathcal{P}^k \in \mathbf{C}$ and elements $\beta^0, \ldots, \beta^k, \in \kappa$ such that $\gamma = \beta^0 \equiv_{\mathcal{P}^1} \beta^1 \equiv_{\mathcal{P}^2} \beta^2 \equiv_{\mathcal{P}^3} \cdots \equiv_{\mathcal{P}^k} \beta^k = \delta$. A set of partitions is called *complete* if it contains both the least upper bound and greatest lower bound of all its subset, and *closed* if this is true for every nonempty subset, i.e., a closed set need not include $\bot$ and $\top$.

Finally, the *cofinality* $\mathrm{cf}(\kappa)$ of an infinite cardinal κ is the least cardinal λ such that a set of cardinality κ can be written as a union of λ sets of cardinality strictly smaller than κ: $\mathrm{cf}(\kappa) = \min \left\{ |I| \;\middle|\; \kappa = \left|\bigcup_{i\in I} A_i\right| \wedge \forall i \in I, |A_i| < \kappa \right\}$. Since $\kappa = \bigcup_{i\in\kappa}\{i\}$, we always have $\mathrm{cf}(\kappa) \leq \kappa$.

If $\mathrm{cf}(\kappa) = \kappa$, then the cardinal κ is called *regular*, otherwise it is called *singular*. Under AC, which is assumed throughout this paper, every infinite successor cardinal is regular. König's Theorem [Kön05] implies $\mathrm{cf}(2^\kappa) > \kappa$, and we additionally have $2^\kappa \leq 2^\lambda$ whenever $\kappa < \lambda$. By Easton's theorem [Eas70], these are the only two constraints on permissible values for 2^κ when κ is regular and when only ZFC is assumed. In contrast, when the Generalized Continuum Hypothesis (GCH) is assumed, cardinal exponentiation is completely determined.

For infinite κ, Π_κ has cardinality 2^κ which provides a weak upper bound on the cardinality of its subsets, in particular maximal chains, maximal antichains, and sets of complements.

2 Results

We summarize here the main contributions of the paper.

Theorem (Well-ordered chains: Theorem 232)**.** *Let κ be any cardinal. The cardinality of a maximal well-ordered chain in Π_κ is always exactly κ.*

Theorem (Long chains: Theorem 235)**.** *Let κ be an infinite cardinal. There exist chains of cardinality $> \kappa$ in Π_κ.*

Theorem (Short chains: Theorem 250)**.** *Let κ be an infinite cardinal such that for every cardinal $\lambda < \kappa$ we have $2^\lambda < 2^\kappa$. Then there exists a maximal chain of cardinality $< 2^\kappa$ (but $\geq \kappa$) in Π_{2^κ}.*

Theorem (Antichains: Theorems 251 and 253)**.** *Let κ be an infinite cardinal. Each non-trivial maximal antichain in Π_κ has cardinality between κ and 2^κ, and these bounds are tight (there exists maximal anti-chains with each of these two cardinalities).*

Theorem (Complements: Theorems 260, 262, and 270)**.** *Let κ be an infinite cardinal. All cardinals of the form κ^λ with $0 \leq \lambda \leq \kappa$ occur as the number of complements to some partition $\mathcal{P} \in \Pi_\kappa$, and these are the only cardinalities the set of complements can have.*

For non-trivial partitions $\mathcal{P} \notin \{\bot, \top\}$, the number of complements is between κ and 2^κ, i.e. κ^λ with $1 \leq \lambda \leq \kappa$. The number of complements to $\mathcal{P}$ is 2^κ unless (i) $\mathcal{P}$ contains exactly one block B of cardinality κ, and (ii) $|\kappa \setminus B| < \kappa$. If $\mathcal{P}$ contains one block B of size κ, then $\mathcal{P}$ has $\kappa^{|\kappa\setminus B|}$ complements.

Theorem (Full characterizations under GCH, Theorem 271)**.** *Under the Generalized Continuum Hypothesis, when κ is an infinite cardinal:*

1. *Any maximal well-ordered chain in Π_κ always has cardinality κ.*
2. *Any general maximal chain in Π_κ has cardinality*
   (a) *κ^-, κ, or κ^+ (and all three are always achieved) if κ is a successor cardinal; and*
   (b) *either κ or κ^+ (and both are achieved) if κ is a limit cardinal.*
3. *Any non-trivial maximal antichain in Π_κ has cardinality either κ or κ^+, and both are achieved.*
4. *Any non-trivial partition has either κ or κ^+ complements. $\mathcal{P} \notin \{\bot, \top\}$ has κ complements if and only if (i) $\mathcal{P}$ contains exactly one block, B, of cardinality κ, and (ii) $|\kappa \setminus B| < \mathrm{cf}(\kappa)$; otherwise, $\mathcal{P}$ has κ^+ complements.*

3 Some basic properties

3.1 Saturated chains in complete lattices

In this section, we prove a few properties of chains on complete lattices (not necessarily the partition lattice) that will be used in the later sections.

Definition 218. Let $\mathbf{C}$ be a chain in a complete lattice $\mathbf{L}$. Define the *lower*, respectively *upper*, subchain relative to $x \in \mathbf{L}$ as $\mathbf{C}_x^- = \{ y \in \mathbf{C} \mid y < x \}$ and $\mathbf{C}_x^+ = \{ y \in \mathbf{C} \mid x < y \}$.

Lemma 219. *Let $\mathbf{C}$ be a closed chain in a complete lattice $\mathbf{L}$, and $x \in \mathbf{L}$ such that $\mathbf{C} \cup \{x\}$ is a chain, and the sets $\mathbf{C}_x^-$ and $\mathbf{C}_x^+$ are nonempty.*

- *If $x \notin \mathbf{C}$, then $\vee\mathbf{C}_x^- \prec_{\mathbf{C}} \wedge\mathbf{C}_x^+$.*
- *If $x \in \mathbf{C}$, then either (i) $\vee\mathbf{C}_x^- = x = \wedge\mathbf{C}_x^+$, (ii) $\vee\mathbf{C}_x^- \prec_{\mathbf{C}} \wedge\mathbf{C}_x^+$, or (iii) $\vee\mathbf{C}_x^- \prec_{\mathbf{C}} x \prec_{\mathbf{C}} \wedge\mathbf{C}_x^+$.*

This also implies that when $x \in \mathbf{C}$, $\vee\mathbf{C}_x^- \preceq_{\mathbf{C}} x \preceq_{\mathbf{C}} \wedge\mathbf{C}_x^+$.

This is easily proved by checking each case. The key points are: i) Because the chain $\mathbf{C}$ is closed, there exists no $y \in \mathbf{C}$ with $\vee\mathbf{C}_x^- < y < \wedge\mathbf{C}_x^+$; and ii) $x \in \mathbf{C} \Rightarrow \vee\mathbf{C}_x^- \le x \le \wedge\mathbf{C}_x^+$.

Lemma 220. *An endpoint-including chain $\mathbf{C}$ in a complete lattice $\mathbf{L}$ is saturated if and only if it is closed and covering; and it is maximal if and only if it is complete and covering.*

Proof. Let in the following $\mathbf{C}$ be a chain in a complete lattice $\mathbf{L}$, such that $\mathbf{C}$ has minimal element $c_{\min}$ and maximal element $c_{\max}$. Since $\mathbf{C}$ is totally ordered, $c_{\min} = \wedge\mathbf{C}$ is its unique least element, and $c_{\max} = \vee\mathbf{C}$ is its unique greatest element.

Saturated implies closed. Assume that there exists a nonempty subset $\mathbf{D} \subseteq \mathbf{C}$ with greatest lower bound $\wedge\mathbf{D} \notin \mathbf{C}$. By construction,

$$c_{\min} = \wedge\mathbf{C} < \wedge\mathbf{D} < \vee\mathbf{C} = c_{\max} \tag{.1}$$

Let x be an arbitrary element of $\mathbf{C}$. Since $\mathbf{C}$ is a chain, either $d \le x$ for some $d \in \mathbf{D}$, which implies $\wedge\mathbf{D} \le x$; or $x < d$ for all $d \in \mathbf{D}$ which implies $x \le \wedge\mathbf{D}$. It follows that $\mathbf{C} \cup \wedge\mathbf{D}$ is a chain. This together with Equation (.1) contradicts that $\mathbf{C}$ is saturated. Dually we prove that $\vee\mathbf{D} \in \mathbf{C}$ for every nonempty $\mathbf{D} \subseteq \mathbf{C}$. Hence $\mathbf{C}$ is saturated only if it is closed.

Saturated implies covering. Assume $\mathbf{C}$ is not covering, i.e. there exists $x, z \in \mathbf{C}$ with $x \prec_{\mathbf{C}} z$ but $x \not\prec_{\mathbf{L}} z$. The second relation implies that there exists $y \in \mathbf{L} \setminus \mathbf{C}$ with $x < y < z$. Because $x \prec_{\mathbf{C}} z$, every other element of $\mathbf{C}$ is either smaller than x or larger than z, hence comparable with y, whereby $\mathbf{C} \cup \{y\}$ is a chain. Hence $\mathbf{C}$ is saturated only if it is covering.

Closed and covering implies saturated. Assume that $\mathbf{C}$ is closed and covering, and choose any $x \in \mathbf{L}$ with $c_{\min} < x < c_{\max}$ for which $\mathbf{C} \cup \{x\}$ is still a chain. Then $c_{\min} \in \mathbf{C}_x^-$ and $c_{\max} \in \mathbf{C}_x^+$, so both sets are nonempty. If $x \notin \mathbf{C}$, Lemma 219 implies $\vee\mathbf{C}_x^- \prec_{\mathbf{C}} \wedge\mathbf{C}_x^+$, and, because $\mathbf{C}$ is covering, $\vee\mathbf{C}_x^- \prec \wedge\mathbf{C}_x^+$. But $\vee\mathbf{C}_x^- \le x \le \wedge\mathbf{C}_x^+$, so this implies $x = \vee\mathbf{C}_x^-$ or $x = \wedge\mathbf{C}_x^+$, contradicting $x \notin \mathbf{C}$, as both are in $\mathbf{C}$. Hence, $\mathbf{C}$ is saturated.

Maximal is equivalent to complete and covering. A maximal chain is a saturated chain that contains $\top$ and $\bot$; and a complete sublattice is a closed sublattice that contains $\top$ and $\bot$, yielding the lemma's second statement. □

3.2 Meets and joins of chains in Π_κ

Definition 221. Let $\mathbf{C}$ be a chain in Π_κ. We define the *lower* and *upper* subchains relative to $x, y \in \kappa$ as $\mathbf{C}_{x,y}^- = \{\mathcal{P} \in \mathbf{C} \mid x \not\equiv_{\mathcal{P}} y\}$ and $\mathbf{C}_{x,y}^+ = \{\mathcal{P} \in \mathbf{C} \mid x \equiv_{\mathcal{P}} y\}$;

Lemma 222. *Let $\mathbf{C}$ be a chain in Π_κ and $\mathbf{D}$ be any set of κ-partitions.*

1. *$x \equiv_{\wedge\mathbf{D}} y$ if and only if $x \equiv_{\mathcal{P}} y$ for all $\mathcal{P} \in \mathbf{D}$;*
2. *$x \equiv_{\vee\mathbf{C}} y$ if and only if $x \equiv_{\mathcal{P}} y$ for some $\mathcal{P} \in \mathbf{C}$;*

(3) *If $\mathbf{C}$ is closed, given elements $x, y \in \kappa$ for which $\mathbf{C}_{x,y}^-$ and $\mathbf{C}_{x,y}^+$ are both nonempty, we have*
   - *(a) $\wedge\mathbf{C}_{x,y}^+ \in \mathbf{C}_{x,y}^+$,*
   - *(b) $\vee\mathbf{C}_{x,y}^- \in \mathbf{C}_{x,y}^-$,*
   - *(c) $\vee\mathbf{C}_{x,y}^- \prec_{\mathbf{C}} \wedge\mathbf{C}_{x,y}^+$ (and, if $\mathbf{C}$ is covering, $\vee\mathbf{C}_{x,y}^- \prec \wedge\mathbf{C}_{x,y}^+$).*

Proof.

1. This is the definition of $\wedge\mathbf{D}$.
2. By definition of $x \equiv_{\vee\mathbf{C}} y$, there exist *finite* sequences $\{x^i\}$ and $\{\mathcal{P}^i\}$ such that $x = x^0 \equiv_{\mathcal{P}^1} x^1 \equiv_{\mathcal{P}^2} x^2 \equiv_{\mathcal{P}^3} \cdots \equiv_{\mathcal{P}^n} x^n = y$. But because $\mathbf{C}$ is totally ordered, the finite set $\{\mathcal{P}^i\} \subset \mathbf{C}$ has a greatest element $\mathcal{P}^k$, and thus $x_0 \equiv_{\mathcal{P}^k} x_1 \equiv_{\mathcal{P}^k} \cdots \equiv_{\mathcal{P}^k} x_n$, whereby $x \equiv_{\mathcal{P}^k} y$.

(3) (a) follows immediately from (1), and (b) from the negation of (2). For (c), notice that (a) and (b) imply $\vee\mathbf{C}_{x,y}^- \ne \wedge\mathbf{C}_{x,y}^+$, whereby $\vee\mathbf{C}_{x,y}^- < \wedge\mathbf{C}_{x,y}^+$. Because every $\mathcal{P} \in \mathbf{C}$ lies either in $\mathbf{C}_{x,y}^-$ or $\mathbf{C}_{x,y}^+$, no $\mathcal{P} \in \mathbf{C}$ can have the property $\vee\mathbf{C}_{x,y}^- < \mathcal{P} < \wedge\mathbf{C}_{x,y}^+$, whereby $\vee\mathbf{C}_{x,y}^- \prec_{\mathbf{C}} \wedge\mathbf{C}_{x,y}^+$. The final statement is simply the definition of covering.

□

Corollary 223. *Given a non-empty chain $\mathbf{C}$, each block of $\vee\mathbf{C}$ is the union, and each block of $\wedge\mathbf{C}$ is the intersection, of an increasing sequence of blocks, one from each $\mathcal{P} \in \mathbf{C}$.*

Proof. Fix $x \in \kappa$, and for each $\mathcal{P} \in \mathbf{C}$ let $B_\mathcal{P}$ be the block of $\mathcal{P}$ containing x. Let B (resp. B') be the block of $\wedge\mathbf{C}$ (resp. $\vee\mathbf{C}$) that contain x. The second equivalence in each case are Lemma 222(1-2). The statement for the greatest lower bound is derived as

$$y \in B \Leftrightarrow x \equiv_{\wedge\mathbf{C}} y \Leftrightarrow \forall\mathcal{P} \in \mathbf{C}, x \equiv_\mathcal{P} y \Leftrightarrow \forall\mathcal{P} \in \mathbf{C}, y \in B_\mathcal{P} \Leftrightarrow y \in \bigcap_{\mathcal{P}\in\mathbf{C}} B_\mathcal{P}$$

and for the least upper bound as

$$y \in B' \Leftrightarrow x \equiv_{\vee\mathbf{C}} y \Leftrightarrow \exists\mathcal{P} \in \mathbf{C}, x \equiv_\mathcal{P} y \Leftrightarrow \exists\mathcal{P} \in \mathbf{C}, y \in B_\mathcal{P} \Leftrightarrow y \in \bigcup_{\mathcal{P}\in\mathbf{C}} B_\mathcal{P}$$

□

Notice that the choice of x, hence also of the $B_\mathcal{P}$ in the union, is far from unique. These lemmas essentially state that "nothing happens when going to the limit". All the equivalences between elements that are present in the limit (*e.g.*, the join) were actually already here in some partition of the chain. Notably, if the chain is well-ordered, the "merge" between two elements must happen between a (partition indexed by an) ordinal and its successor, and does not suddenly appear at a (partition indexed by a) limit ordinal.

3.3 Restriction

Definition 224 (Restriction)**.** Given a partition $\mathcal{P}$ of κ, we define its *restriction to* $\lambda \leq \kappa$ as $\mathcal{P}\sqcap\lambda = \{ B \cap \lambda \mid B \in \mathcal{P} \} \setminus \{\emptyset\}$, which is a partition of λ. Similarly, a set $\mathbf{D}$ of partitions restricts to a subset of Π_λ as $\mathbf{D}\sqcap\lambda = \{ \mathcal{P} \sqcap \lambda \mid \mathcal{P} \in \mathbf{D} \}$.

Lemma 225. *(i) $\top_\kappa \sqcap \lambda = \top_\lambda$; (ii) $\bot_\kappa \sqcap \lambda = \bot_\lambda$; (iii) $\mathcal{P} < \mathcal{Q}$ implies $\mathcal{P}\sqcap\lambda \leq \mathcal{Q}\sqcap\lambda$; (iv) if $\mathbf{C}$ is a chain, so is $\mathbf{C}\sqcap\lambda$; (v) if $\mathcal{P}, \mathcal{Q}$ are comparable, then $\mathcal{P}\sqcap\lambda < \mathcal{Q}\sqcap\lambda$ implies $\mathcal{P} < \mathcal{Q}$; and (vi) $\mathcal{P} \prec \mathcal{Q}$ implies $\mathcal{P}\sqcap\lambda \preceq \mathcal{Q}\sqcap\lambda$.*

All these Facts can be checked easily by direct applications of the definitions.

Lemma 226. *If $\mathbf{C}$ is a chain in Π_κ and $\lambda \leq \kappa$, then $(\vee\mathbf{C})\sqcap\lambda = \vee(\mathbf{C}\sqcap\lambda)$, and $(\wedge\mathbf{C})\sqcap\lambda = \wedge(\mathbf{C}\sqcap\lambda)$.*

Proof. Let $\lambda \leq \kappa$, and $\mathbf{C}$ be a chain in Π_κ. If $\mathbf{C}$ is empty, then the result follows directly from Lemma 225(i-ii) and the definition of meet and join on the empty set.

Assume now that $\mathbf{C}$ is nonempty, and fix $x \in \lambda$. For each $\mathcal{P} \in \mathbf{C}$, let $B_\mathcal{P}$ be the block of $\mathcal{P}$ containing x, and $B_{\tilde{\mathcal{P}}}$ be the block of $\tilde{\mathcal{P}} = \mathcal{P} \sqcap \lambda$ containing x.
Greatest lower bound. Let B be the block in $\wedge\mathbf{C}$, and B_λ be the block in $\wedge(\mathbf{C}\sqcap\lambda)$, that contains x. Applying Corollary 223 to both B_λ and to B yields

$$B_\lambda = \bigcap_{\tilde{\mathcal{P}}\in\mathbf{C}\sqcap\lambda} B_{\tilde{\mathcal{P}}} = \bigcap_{\mathcal{P}\in\mathbf{C}} B_{\tilde{\mathcal{P}}} = \bigcap_{\mathcal{P}\in\mathbf{C}} (B_\mathcal{P} \cap \lambda) = (\bigcap_{\mathcal{P}\in\mathbf{C}} B_\mathcal{P}) \cap \lambda = B \cap \lambda$$

Note that restriction is not injective, hence we may have $\mathcal{P} \neq \mathcal{P}'$ but $\mathcal{P}\sqcap\lambda = \mathcal{P}'\sqcap\lambda$. In this case, $B_{\tilde{\mathcal{P}}} = B_{\tilde{\mathcal{P}}'}$, thus the second equality above is correct.

As this holds for every $x \in \lambda$, each block of $\wedge(\mathbf{C}\sqcap\lambda)$ is the restriction of a block of $\wedge\mathbf{C}$, therefore $\wedge(\mathbf{C}\sqcap\lambda) = (\wedge\mathbf{C})\sqcap\lambda$.
Least upper bound: The proof is symmetrical with key equalities being:

$$B_\lambda = \bigcup_{\tilde{\mathcal{P}}\in\mathbf{C}\sqcap\lambda} B_{\tilde{\mathcal{P}}} = \bigcup_{\mathcal{P}\in\mathbf{C}} B_{\tilde{\mathcal{P}}} = \bigcup_{\mathcal{P}\in\mathbf{C}} (B_\mathcal{P} \cap \lambda) = (\bigcup_{\mathcal{P}\in\mathbf{C}} B_\mathcal{P}) \cap \lambda = B \cap \lambda$$

□

Lemma 227. *Restriction preserves well-order and completeness of chains.*

Proof. Let $\lambda \leq \kappa$ and $\mathbf{C}$ be a chain in Π_κ. By Lemma 225(iv), $\mathbf{C}\sqcap\lambda$ is a chain in Π_λ. Let $\mathbf{D}_\lambda \subseteq \mathbf{C}\sqcap\lambda$ be any, possibly empty, subset. We have $\mathbf{D}_\lambda = \mathbf{D}\sqcap\lambda$, with $\mathbf{D} = \{ \mathcal{P} \in \mathbf{C} \mid \mathcal{P}\sqcap\lambda \in \mathbf{D}_\lambda \}$. Recall that subsets of chains are chains themselves, hence by Lemma 226, $\vee\mathbf{D}_\lambda = \vee(\mathbf{D}\sqcap\lambda) = (\vee\mathbf{D})\sqcap\lambda$, and $\wedge\mathbf{D}_\lambda = \wedge(\mathbf{D}\sqcap\lambda) = (\wedge\mathbf{D})\sqcap\lambda$.
Well-order: If $\mathbf{C}$ is well-ordered, then $\wedge\mathbf{D} \in \mathbf{D}$ is its least element. By construction of $\mathbf{D}$, $\wedge\mathbf{D}_\lambda = (\wedge\mathbf{D})\sqcap\lambda \in \mathbf{D}_\lambda$ which is thus the least element of $\mathbf{D}_\lambda$. Hence, $\mathbf{C}\sqcap\lambda$ is well-ordered.
Completeness: If $\mathbf{C}$ is complete, then $\wedge\mathbf{D} \in \mathbf{C}$. Hence, $\wedge\mathbf{D}_\lambda = (\wedge\mathbf{D})\sqcap\lambda \in \mathbf{C}\sqcap\lambda$. Similarly, $\vee\mathbf{D}_\lambda \in \mathbf{C}\sqcap\lambda$. Thus, $\mathbf{C}\sqcap\lambda$ is complete. □

Lemma 228. *Restriction preserves maximality of chains.*

Proof. Because of the nature of the proof, we explicitly tell in which set of partitions the inequalities hold (even when it is the whole lattice). We abusively write $\mathcal{P} <_\lambda \mathcal{Q}$ (and so on) instead of $\mathcal{P} <_{\Pi_\lambda} \mathcal{Q}$, for the sake of clarity.

Let $\lambda \leq \kappa$ and $\mathbf{C}$ be a maximal chain in Π_κ. It is complete and covering by Lemma 220. Hence, $\mathbf{C} \sqcap \lambda$ is a complete chain due to Lemma 227 and we only need to show that it is also covering.

Let $\tilde{\mathcal{P}}, \tilde{\mathcal{Q}} \in \mathbf{C} \sqcap \lambda$ such that $\tilde{\mathcal{P}} \prec_{\mathbf{C}\sqcap\lambda} \tilde{\mathcal{Q}}$. By construction, there exist $x, y \in \lambda$ such that $x \not\equiv_{\tilde{\mathcal{P}}} y$ and $x \equiv_{\tilde{\mathcal{Q}}} y$, whereby $\tilde{\mathcal{P}} \in (\mathbf{C} \sqcap \lambda)^-_{x,y}$ and $\tilde{\mathcal{Q}} \in (\mathbf{C} \sqcap \lambda)^+_{x,y}$. Note that because both x and y are in λ, we have $(\mathbf{C} \sqcap \lambda)^-_{x,y} = \mathbf{C}^-_{x,y} \sqcap \lambda$ and $(\mathbf{C} \sqcap \lambda)^+_{x,y} = \mathbf{C}^+_{x,y} \sqcap \lambda$.

By definition of restrictions, there exist (non-unique) $\mathcal{P}, \mathcal{Q} \in \mathbf{C}$ such that $\tilde{\mathcal{P}} = P \sqcap \lambda$ and $\tilde{\mathcal{Q}} = \mathcal{Q} \sqcap \lambda$, and we must have $x \not\equiv_{\mathcal{P}} y$ and $x \equiv_{\mathcal{Q}} y$, whereby $\mathcal{P} \in \mathbf{C}^-_{x,y}$ and $\mathcal{Q} \in \mathbf{C}^+_{x,y}$. Since $\mathbf{C}$ is maximal, it is complete and covering, hence Lemma 222(3)(c) yields $\vee\mathbf{C}^-_{x,y} \prec_\kappa \wedge\mathbf{C}^+_{x,y}$. Because they differ on $x, y \in \lambda$, we have $(\vee\mathbf{C}^-_{x,y}) \sqcap \lambda \prec_\lambda (\wedge\mathbf{C}^+_{x,y}) \sqcap \lambda$ by Lemma 225(vi), hence Lemma 226 yields $\vee\left(\mathbf{C}^-_{x,y} \sqcap \lambda\right) \prec_\lambda \wedge\left(\mathbf{C}^+_{x,y} \sqcap \lambda\right)$, i.e., $\vee(\mathbf{C} \sqcap \lambda)^-_{x,y} \prec_\lambda \wedge(\mathbf{C} \sqcap \lambda)^+_{x,y}$.

By completeness of $\mathbf{C} \sqcap \lambda$, it contains both $\vee(\mathbf{C} \sqcap \lambda)^-_{x,y}$ and $\wedge(\mathbf{C} \sqcap \lambda)^+_{x,y}$, thus $\vee(\mathbf{C} \sqcap \lambda)^-_{x,y} \prec_{\mathbf{C}\sqcap\lambda} \wedge(\mathbf{C} \sqcap \lambda)^+_{x,y}$. Because $\tilde{\mathcal{P}} \in (\mathbf{C} \sqcap \lambda)^-_{x,y}$, we have $\tilde{\mathcal{P}} \leq_\lambda \vee(\mathbf{C} \sqcap \lambda)^-_{x,y}$ and similarly, $\wedge(\mathbf{C} \sqcap \lambda)^+_{x,y} \leq_\lambda \tilde{\mathcal{Q}}$,

Because $\tilde{\mathcal{P}} \prec_{\mathbf{C}\sqcap\lambda} \tilde{\mathcal{Q}}$ by definition, we must have $\tilde{\mathcal{P}} = \vee(\mathbf{C} \sqcap \lambda)^-_{x,y} \prec_{\mathbf{C}\sqcap\lambda} \wedge(\mathbf{C} \sqcap \lambda)^+_{x,y} = \tilde{\mathcal{Q}}$, therefore $\tilde{\mathcal{P}} \prec_\lambda \tilde{\mathcal{Q}}$ and $\mathbf{C} \sqcap \lambda$ is covering. Being both complete and covering, $\mathbf{C} \sqcap \lambda$ is maximal. ☐

4 Well-ordered chains in Π_κ

For finite $\kappa = n$, it is immediate that any maximal chain in Π_n has cardinality n: Each step reduces the number of blocks by 1, whereby going from $\perp$ with n blocks to $\top$ with 1 block requires $n-1$ steps, hence n elements in the chain. For $n \geq 3$, maximal chains are not unique. In this section, we show that maximal well-ordered chains in Π_κ *always* have cardinality κ, whether κ is finite or infinite.

If a maximal chain in Π_κ is well-ordered of order type α, then α is a successor ordinal.[1]For clarity, we will write the order type of a maximal chain as $\alpha + 1$ to emphasize that it is a successor ordinal. Because $|\alpha| = |\alpha + 1|$, this has no impact on the cardinality of the chain. Such chains can be written as $\mathbf{C} = \{\, \mathcal{P}_\beta \mid \beta \leq \alpha \,\}$ — or as $\mathbf{C} = \{\, \mathcal{P}_\beta \mid \beta < \alpha + 1 \,\}$ to emphasize the order type — with $\mathcal{P}_0 = \perp$ and $\mathcal{P}_\alpha = \top$.

Lemma 229. *Let κ be an infinite cardinal. If a chain in Π_κ is well-ordered of order type α, then $|\alpha| \leq \kappa$.*

Note that we are here speaking of any well-ordered chain, not necessarily a maximal one, so its order type may be anything.

Proof. As Π_κ is isomorphic to $\mathrm{Equ}(\kappa)$, the set of equivalence relations on κ ordered by $\subseteq$, there is, for any chain of order type α in Π_κ, a chain of order type α in the poset $(\mathscr{P}(\kappa \times \kappa), \subseteq)$. Let $\mathbf{C} = \{\, \mathcal{E}_\beta \mid \beta < \alpha \,\}$ be such a chain and observe that for every β with $\beta + 1 < \alpha$ there exists at least one $\gamma_\beta \in \mathcal{E}_{\beta+1} \setminus \mathcal{E}_\beta$. As $\mathbf{C}$ is totally ordered under $\subseteq$, this implies $\gamma_\beta \notin \mathcal{E}_{\beta'}$ when $\beta' < \beta$, i.e. the γ_β do not repeat. Hence, $\{\, \gamma_\beta \mid \beta + 1 < \alpha \,\} \subseteq \kappa \times \kappa$ has cardinality $|\alpha|$, implying $|\alpha| \leq |\kappa \times \kappa| = \kappa$. ☐

Lemma 230. *Every well-ordered maximal chain of order type $\alpha + 1$ in Π_κ satisfies $\mathrm{cf}(\kappa) \leq |\alpha|$.*

Proof. Let $\mathbf{C} = \{\, \mathcal{P}_\beta \mid \beta < \alpha + 1 \,\}$ be a maximal well-ordered chain in Π_κ of order type $\alpha+1$. Consider the partitions in this chain that have at least one block of cardinality κ. Since $\top = \mathcal{P}_\alpha$, there is at least one such partition. Let δ be the least ordinal such that $\mathcal{P}_\delta$ contains a block of cardinality κ. It exists, because every non-empty set of ordinals has a least element. Let B^δ be a block of cardinality κ in $\mathcal{P}_\delta$.

By Lemma 219 and maximality of $\mathbf{C}$ we have $\vee\mathbf{C}^-_{\mathcal{P}_\delta} \preceq \mathcal{P}_\delta$, and $\vee\mathbf{C}^-_{\mathcal{P}_\delta} \in \mathbf{C}$. That is, we can write $\mathcal{P}_\gamma = \vee\mathbf{C}^-_{\mathcal{P}_\delta}$ either for $\gamma = \delta$ or $\gamma + 1 = \delta$. Consider for the sake of contradiction that $\mathcal{P}_\gamma \prec \mathcal{P}_\delta$. Then B^δ would be either a block of $\mathcal{P}_\gamma$ or the union of two such blocks, at least one of them with cardinality κ. Both cases contradict the hypothesis of δ being the smallest ordinal for which $\mathcal{P}_\delta$ contains a block of cardinality κ. Hence $\delta = \gamma$, and $\mathcal{P}_\delta = \vee\mathbf{C}^-_{\mathcal{P}_\delta} = \vee_{\beta<\delta}\mathcal{P}_\beta$.

By Corollary 223 there exists a sequence $\{B^\beta\}_{\beta<\delta}$ such that $B^\delta = \bigcup_{\beta<\delta} B^\beta$. By definition of δ, we have $\left|B^\beta\right| < \kappa$ for every $\beta < \delta$, and so by the definition of co-finality, $\mathrm{cf}(\kappa) \leq |\delta| \leq |\alpha|$ ☐

Corollary 231. *If κ is a regular cardinal, then every well-ordered maximal chain in Π_κ indexed by an ordinal α satisfies $|\alpha| = \kappa$.*

Theorem 232. *Let κ be an infinite cardinal. Every well-ordered maximal chain of order type $\alpha + 1$ in Π_κ satisfies $|\alpha| = \kappa$.*

[1] Any maximal chain in Π_κ has a maximal element, namely $\top$. But since a well-ordered set of limit-ordinal type has no maximal element, this implies that the order type of a well-ordered maximal chain must be a successor-ordinal.

Proof. Because of the previous results, we only need to check that for a singular κ, $|\alpha| \geq \kappa$. Indeed, the case of regular κ is handled by Corollary 231 and the upper bound is due to Lemma 229.
Cardinal arithmetic: It suffices to show that $\lambda \leq |\alpha|$ for every regular $\lambda < \kappa$: By standard cardinal arithmetic, the singular κ can be expressed as the sum of $\mathrm{cf}(\kappa)$ smaller successor (hence regular) cardinals λ_δ. If the above inequality holds, then:

$$\kappa = \sum_{\delta \leq \mathrm{cf}(\kappa)} \lambda_\delta \leq \sum_{\delta \leq \mathrm{cf}(\kappa)} |\alpha| = \mathrm{cf}(\kappa) \cdot |\alpha| = \max(\mathrm{cf}(\kappa), |\alpha|)$$

And because $\kappa > \mathrm{cf}(\kappa)$ by singularity, we can finally conclude that $\kappa \leq |\alpha|$.
Cardinality: Let $\mathbf{C}$ be a maximal, well-ordered chain in Π_κ. By Lemmas 227 and 228, $\mathbf{C} \sqcap \lambda$ is a maximal well-ordered chain in Π_λ for every regular $\lambda < \kappa$. Hence, by Corollary 231, we have $|\mathbf{C} \sqcap \lambda| = \lambda$. Moreover, by construction, $|\mathbf{C} \sqcap \lambda| \leq |\mathbf{C}| = |\alpha|$. Thus, for any regular $\lambda < \kappa$ we have $\lambda \leq |\alpha|$ and we can conclude that $\kappa \leq |\alpha|$, whereby $\kappa = |\alpha|$. □

Remark 233. Notice that any ordinal $\alpha + 1$ with $|\alpha| = \kappa$ appears as the order type of some maximal well-ordered chain in Π_κ. Indeed, the chain of singular partitions (of $\Pi_{\alpha+1}$) with non-singleton block $B_\beta = \{\, \delta \mid \delta \leq \beta \,\}$ works (in other words, it suffices to find a well-order of order type $\alpha + 1$ on κ, which exists by a cardinality argument, and add the elements to a single block in this order).

5 Long chains in Π_κ

For any nonempty $S \subseteq \kappa$, we define the partition

$$\mathrm{diag}(S) = \{S\} \cup \{\, \{\gamma\} \mid \gamma \in \kappa \setminus S \,\}.$$

When $|S| \geq 2$, $\mathrm{diag}(S)$ is the singular partition with non-singular block S.

Remark 234. It is easy to verify that $\mathrm{diag}(\cdot)$ is an order isomorphism between subsets of κ of size ≥ 2 and the singular partitions. In particular, chains in $(\mathscr{P}(\kappa), \subseteq)$ containing only sets of length ≥ 2 are mapped to chains with the same cardinality in $(\Pi_\kappa, \leq)$.

Theorem 235. *Let κ be an infinite cardinal. There is a chain of cardinality $> \kappa$ in Π_κ. The chain may be chosen to be maximal.*

Proof. By a result of Sierpiński [Sie22] (see also [Har05, Thm. 4.7.35]), there is a chain $\mathbf{D}'$ in the poset $(\mathscr{P}(\kappa), \subseteq)$ of cardinality $\lambda > \kappa$. There is at most one singleton element $\{\alpha\} \in \mathbf{D}'$, and as λ is infinite, $\mathbf{D} = \mathbf{D}' \setminus \{\emptyset, \{\alpha\}\}$ is still a chain of cardinality λ.

Then, by Remark 234, the set $\mathbf{C} = \{\, \mathrm{diag}(S) \mid S \in \mathbf{D} \,\}$ is a chain in Π_κ of cardinality $\lambda > \kappa$. By the Maximal Chain Theorem, any chain in a poset is contained in a maximal chain, and the result follows. □

For the special case of $\kappa = \aleph_0$, we may obtain existence of a maximal chain in $\Pi_{\aleph_0}$ of cardinality $2^{\aleph_0}$ without use of (G)CH:

Lemma 236. *There is a maximal chain of cardinality $2^{\aleph_0}$ in $\Pi_{\aleph_0}$.*

Proof. For each $r \in \mathbb{R}$, define the left Dedekind cut $D_r = \{\, q \in \mathbb{Q} \mid q < r \,\}$ and note that $r < r'$ implies $D_r \subsetneq D_{r'}$ by density of $\mathbb{Q}$ in $\mathbb{R}$. Hence, the set of left Dedekind cuts is a chain in $\mathscr{P}(\mathbb{Q})$ of cardinality $|\mathbb{R}| = 2^{\aleph_0}$, all elements of which have cardinality ≥ 2, and consequently, by Lemma 234, the set of partitions $\{\, \mathrm{diag}(D_r) \mid r \in \mathbb{R} \,\}$ is a chain in $\Pi_{\aleph_0}$ of cardinality $2^{\aleph_0}$. By the Maximal Chain Theorem, this chain can be extended to a maximal chain; note that $2^{\aleph_0}$ is an upper bound on the cardinality of any chain in $\Pi_{\aleph_0}$. □

For arbitrary infinite κ we do not know whether existence of a maximal chain in Π_κ of cardinality 2^κ can be proved without assuming GCH. In addition, we do not know whether the rather heavy-handed application of the Maximal Chain Theorem (equivalent to the Axiom of Choice) in the proof of Theorem 235 is necessary to establish existence of a maximal chain.

6 Short chains

We now turn to the question of whether there are maximal chains of cardinality strictly less than κ in Π_κ. It is immediate that there are no maximal chains of finite cardinality in $\Pi_{\aleph_0}$. Indeed, each step in a maximal chain merges exactly two blocks. Hence, starting from $\perp$ which has infinitely many blocks, after a finite number of steps it is impossible to reach $\top$ or any other partition that has only finitely many blocks. For larger cardinals, there is a general construction that proves existence of short chains. In the special case where GCH is assumed, it proves existence of a

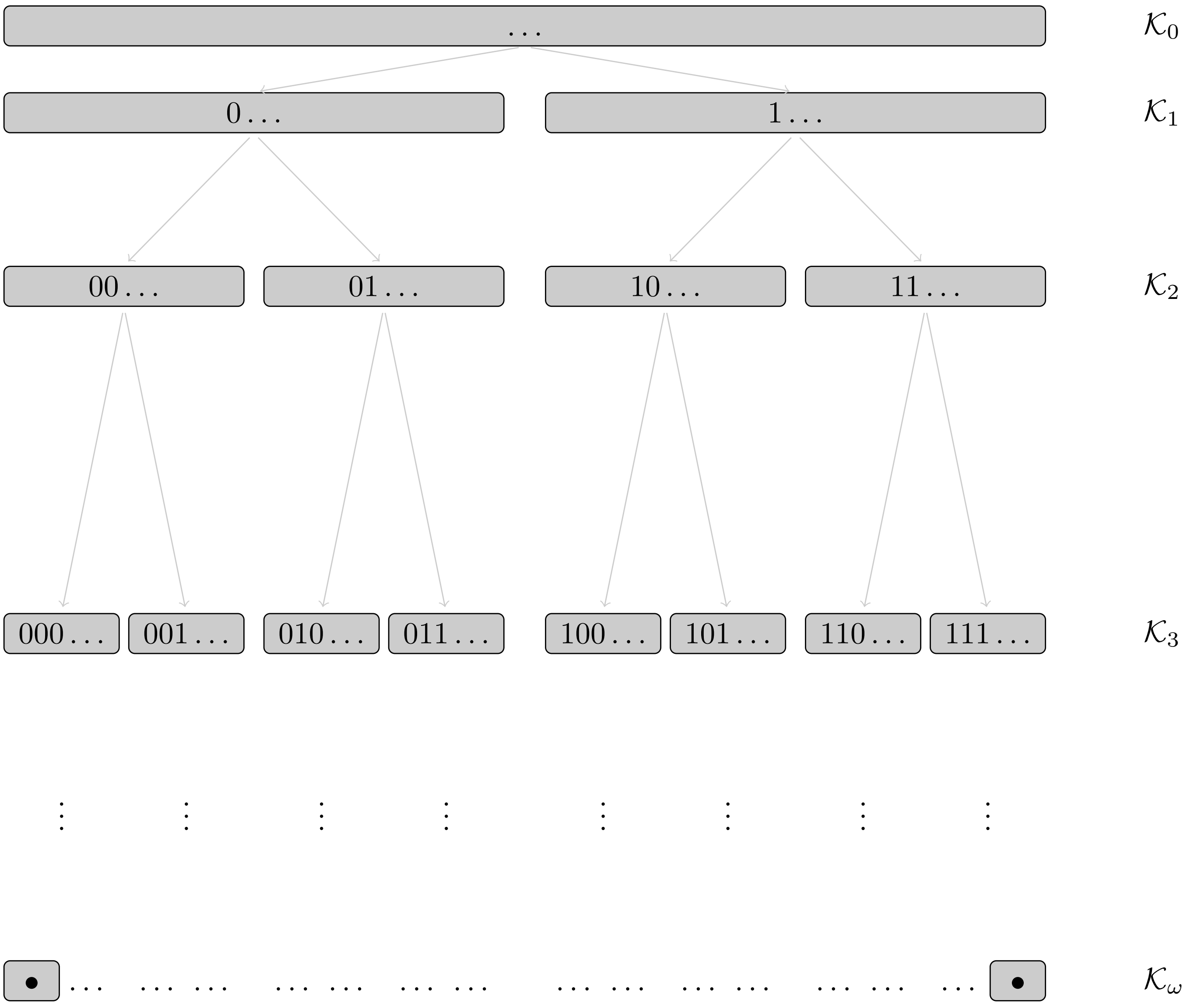


Figure 1: Keyframe partitions

maximal chain of cardinality κ^- for every successor cardinal κ. Because this construction is notationally cumbersome, we first explain the result for the particular case of $\kappa = 2^{\aleph_0}$, the cardinality of the continuum, in order to aid the reader's understanding.

6.1 Short chains in $\Pi_{2^{\aleph_0}}$

We build a maximal chain of size $\aleph_0$ in $\Pi_{2^{\aleph_0}}$. The proof boils down to the fact that there are countably many binary strings of finite length but uncountably many binary strings of countably infinite length, hence an infinite binary tree of depth ω has uncountably many leaves but only countably many inner nodes.

The construction proceeds in two steps:

1. Starting from $\top$, inductively construct a countable chain of "keyframe" partitions such that the n^{th} keyframe has 2^n blocks. The greatest lower bound of this chain will be $\bot$ with $2^{\aleph_0}$ singleton blocks.

2. Complete the chain into a maximal chain by adding "inbetween" partitions between the keyframes. We will need $2^n - 1$ extra partitions between the $(n-1)^{\text{st}}$ and the n^{th} keyframe.[2]

[2]"Keyframe" and "Inbetween" are terms from animation: A "keyframe" is a drawing that defines the start or end of a movement. The animation frames between keyframes–the "inbetweens"–are drawn to make the transitions between keyframes smooth.

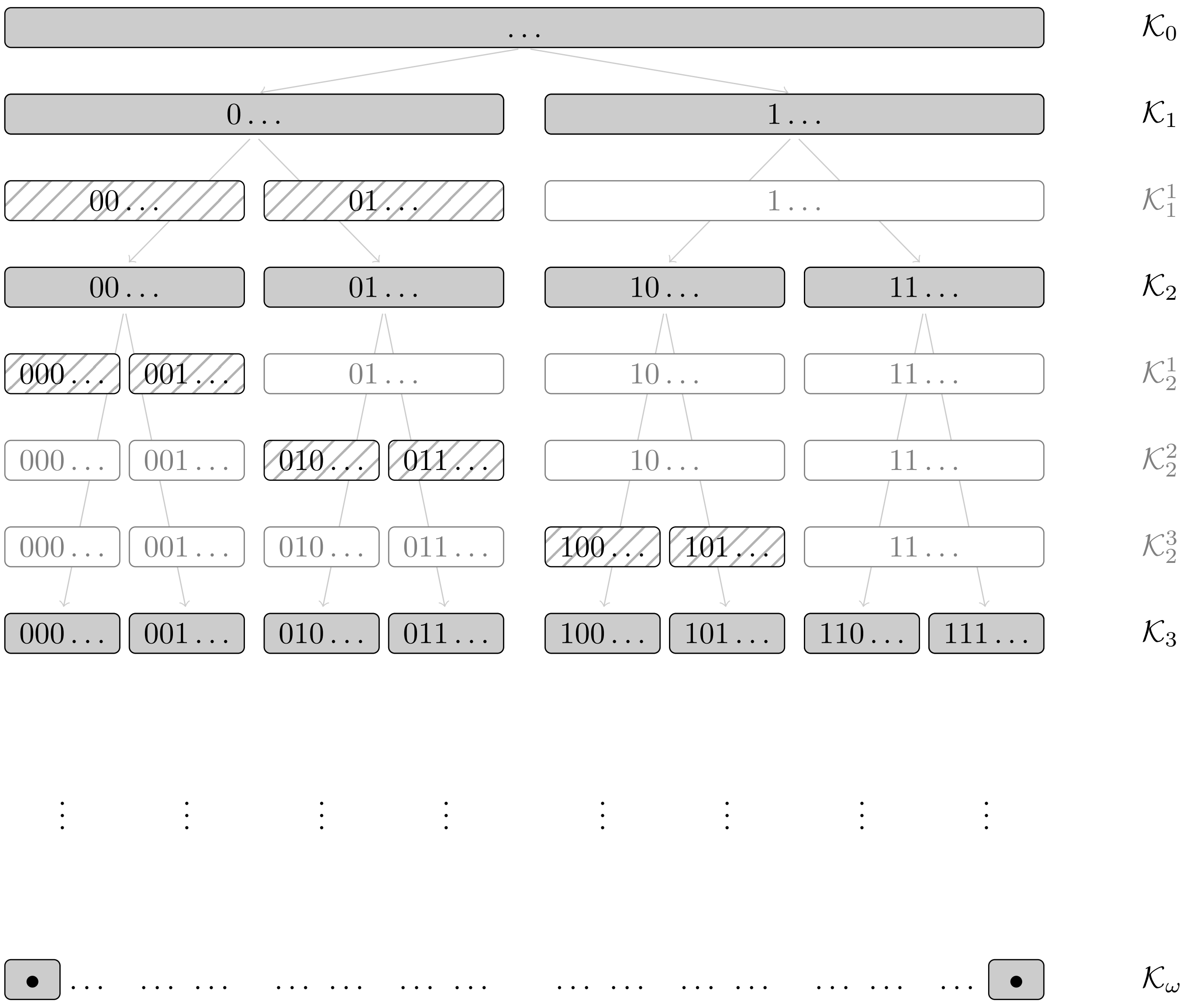


Figure 2: A short maximal chain

As a finite number of inbetween partitions are added for each n, only a countable number of partitions are added in total, and so the full chain has countable length.

Keyframes. Consider partitions of the set $\{0,1\}^\omega$ of countably infinite sequences of bits; note that the cardinality of $\{0,1\}^\omega$ is $2^{\aleph_0}$. Let $u \in \{0,1\}^\omega$ and let $[u]_k$ be the set of elements of $\{0,1\}^\omega$ with the same first k bits: $[u]_k = \{\, u_0u_1 \dots u_{k-1}v \mid v \in \{0,1\}^\omega \,\}$. Define the k^{th} keyframe to be $\mathcal{K}_k = \{\, [u]_k \mid u \in \{0,1\}^\omega \,\}$. Similarly, define $\mathcal{K}_\omega = \perp$.

Thus, the keyframes are as in Figure 1. Each level of the picture corresponds to one partition, with its blocks depicted. It is instructive to note that the set of blocks of all partitions form an infinite binary tree and that the number of nodes in each keyframe is finite (whence the corresponding partition has only finitely many blocks). The blocks of $\mathcal{K}_\omega$ constitute the bottom most level of the tree, and hence are the leaves of the infinite binary tree, of which there are $2^{|\omega|}$, and the total number of *keyframe partitions* corresponds to the number of levels in the infinite tree, and is thus clearly countable. The total number of internal nodes in the tree is $\sum_{\alpha<\omega} 2^{|\alpha|} = \aleph_0$.

Completing the chain. The set of keyframe partitions clearly form a chain, but just as clearly this chain is not maximal. Inbetween partitions must be added between the keyframes to obtain a maximal chain.

As seen in Figure 1, between the k^{th} and the $(k+1)^{\text{st}}$ keyframe, 2^k blocks have been split. In order to locally saturate the chain, it suffices to add $2^k - 1$ new partitions, each one with one more of the 2^k blocks cut in two. This will ensure that each partition is a successor to the previous one.

After adding these new partitions, the picture is now as shown in Figure 2. This chain is easily seen to be maximal. At each (non-keyframe) partition, the new blocks, resulting from splitting one of the blocks of the parent partition, are shown with hatching.

Since there are 2^k inbetween partitions between the k^{th} and $(k+1)^{\text{st}}$ keyframes, the total number of inbetween partitions is $\sum_{k<\omega} 2^k$, which is countable. Since there are also only countably many keyframes, the total number of partitions in the chain is countable.

6.2 Short chains: general construction for large cardinals

Let λ be any infinite cardinal and $\kappa = 2^\lambda$. We will build a maximal chain of cardinality λ in $\Pi_\kappa = \Pi_{2^\lambda}$.

Underlying set. Let S be the set of binary sequences indexed by the initial ordinal of λ. That is $S = \{0,1\}^{\omega_\lambda}$. S contains κ elements; we shall use $\Pi(S)$ as a representative of Π_κ.

Keyframes. Let $f \in S$ and $\delta \leq \omega_\lambda$. We denote by $[f]_\delta$ the set of sequences that agree with f on the initial δ elements:

$$[f]_\delta = \{\, g \in S \;\mid\; \forall \beta < \delta, f(\beta) = g(\beta) \,\}$$

Observe that $[f]_\delta$ is uniquely defined by a binary sequence indexed by δ. The bracketed notation $[f]$ is used to emphasize that $[f]$ is truly the equivalence class of f in the corresponding equivalence, even though everything is written in terms of partitions.

Let $\mathcal{K}_\delta = \{\, [f]_\delta \;\mid\; f \in S \,\}$. Then, $\mathcal{K}_\delta$ is a partition of S which we call a *keyframe partition*. From the definition, it follows that $\mathcal{K}_0 = \top$ and that $\mathcal{K}_{\omega_\lambda} = \bot$. Note that $\mathcal{K}_\delta$ consists of $2^{|\delta|}$ blocks and is in bijective correspondence with $\{0,1\}^\delta$. Let $\mathbf{K} = \{\, \mathcal{K}_\delta \;\mid\; \delta \leq \omega_\lambda \,\}$.

Lemma 237. *If $\alpha < \beta$ then $[f]_\beta \subsetneq [f]_\alpha$ and $\mathcal{K}_\beta < \mathcal{K}_\alpha$. Furthermore, $\mathbf{K}$ is a chain in Π_κ.*

Proof. The first point is immediate by definition of the refinement ordering: if f and g agree on their initial β elements, then they trivially agree on their initial α elements, so $[f]_\beta \subsetneq [f]_\alpha$. Since this holds for every $f \in S$, we further get $\mathcal{K}_\beta < \mathcal{K}_\alpha$. The second point is an immediate consequence of the first. ☐

Lemma 238. $\mathbf{K}$ *is closed.*

Proof. Let D be a set of ordinals and $\mathbf{D} = \{\, \mathcal{K}_\delta \;\mid\; \delta \in D \,\}$. Being a set of ordinals, D admits a minimal element α. Then $\mathcal{K}_\alpha$ is an upper bound for $\mathbf{D}$ and being part of it, must be the least upper bound.

Let β be the smallest ordinal not in D. If $\beta = \beta' + 1$ is a successor ordinal, then $\mathcal{K}_{\beta'}$ is both part of $\mathbf{D}$ and a lower bound, hence it's the greatest lower bound.

Suppose that β is a limit ordinal and that $\wedge\mathbf{D} > \mathcal{K}_\beta$. Then, there exists f, g such that $f \equiv_{\wedge\mathbf{D}} g$ but $f \not\equiv_{\mathcal{K}_\beta} g$. By definition of $\mathcal{K}_\beta$, there exists $\gamma < \beta$ with $f(\gamma) \neq g(\gamma)$. However, by definition of β, there must exists $\gamma' \in D$ with $\gamma \leq \gamma' < \beta$. $f(\gamma) \neq g(\gamma)$ thus implies $f \not\equiv_{\mathcal{K}_{\gamma'}} g$ which contradicts $f \equiv_{\wedge\mathbf{D}} g$. Therefore, $\mathbf{K}$ is closed. ☐

Locally saturating the chain. We fix $\delta < \omega_\lambda$ and construct a saturated chain bounded below by $\mathcal{K}_{\delta+1}$ and above by $\mathcal{K}_\delta$.

$\mathcal{K}_\delta$ has cardinality $2^{|\delta|}$. Fix, by the Axiom of Choice, a well-ordering $<_\delta$ of $\mathcal{K}_\delta$ with order type $\omega_{2^{|\delta|}}$. For $0 \leq \alpha < \omega_{2^{|\delta|}}$, let the block $S_{\delta,\alpha}$ be the α^{th} element of $\mathcal{K}_\delta$ according to this ordering.

For each $0 \leq \alpha < \omega_{2^{|\delta|}}$, define the set $[f]^\alpha_\delta$ as

$$[f]^\alpha_\delta = \begin{cases} [f]_{\delta+1} & \text{if } [f]_\delta <_\delta S_{\delta,\alpha} \\ [f]_\delta & \text{if } S_{\delta,\alpha} \leq_\delta [f]_\delta \end{cases}$$

and let $\mathcal{K}^\alpha_\delta = \{\, [f]^\alpha_\delta \;\mid\; f \in S \,\}$. Observe that the sets $[f]^\alpha_\delta$ are pairwise disjoint and that their union is S. Thus, $\mathcal{K}^\alpha_\delta$ is a partition of S (an "inbetween" partition), obtained by splitting the initial α blocks of $\mathcal{K}_\delta$. By construction there are $2^{|\delta|}$ inbetween partitions between $\mathcal{K}_\delta$ and $\mathcal{K}_{\delta+1}$.

Lemma 239. *For all $f \in S$, $[f]^0_\delta = [f]_\delta$. Thus $\mathcal{K}^0_\delta = \mathcal{K}_\delta$.*

Proof. By construction, $S_{\delta,0}$ is minimal for $<_\delta$, thus it is impossible for $[f]_\delta$ to be strictly smaller. ☐

Lemma 240. *Let $\alpha < \beta$. Then for each f, $[f]^\beta_\delta \subseteq [f]^\alpha_\delta$.*

Proof. By cases depending on $<_\delta$. First note that $\alpha < \beta$ implies $S_{\delta,\alpha} <_\delta S_{\delta,\beta}$ by construction.

1. If $[f]_\delta <_\delta S_{\delta,\alpha} <_\delta S_{\delta,\beta}$ then $[f]^\beta_\delta = [f]_{\delta+1} = [f]^\alpha_\delta$.

2. If $S_{\delta,\alpha} \leq_\delta [f]_\delta <_\delta S_{\delta,\beta}$ then $[f]^\beta_\delta = [f]_{\delta+1} \subsetneq [f]_\delta = [f]^\alpha_\delta$.

3. If $S_{\delta,\alpha} <_\delta S_{\delta,\beta} \leq_\delta [f]_\delta$ then $[f]^\beta_\delta = [f]_\delta = [f]^\alpha_\delta$.

☐

Lemma 241. *Let $\alpha < \beta$. Then $\mathcal{K}^\beta_\delta < \mathcal{K}^\alpha_\delta$.*

Proof. By Lemma 240, each block of $\mathcal{K}_\delta^\beta$ is contained in a block of $\mathcal{K}_\delta^\alpha$, thus $\mathcal{K}_\delta^\beta \leq \mathcal{K}_\delta^\alpha$.

Since $\alpha < \beta$, there exists f such that $S_{\delta,\alpha} \leq_\delta [f]_\delta <_\delta S_{\delta,\beta}$. Hence there are blocks of $\mathcal{K}_\delta^\beta$ that are strictly contained in blocks of $\mathcal{K}_\delta^\alpha$, and thus the inequality is indeed strict. □

Let $\mathbf{C}'_\delta = \{\, \mathcal{K}_\delta^\alpha \mid 0 \leq \alpha < \omega_{2^{|\delta|}} \,\}$ and $\mathbf{C}_\delta = \mathbf{C}'_\delta \cup \{\mathcal{K}_{\delta+1}\}$.

Corollary 242. $\mathbf{C}_\delta$ *is a chain with minimum* $\mathcal{K}_{\delta+1}$ *and maximum* $\mathcal{K}_\delta = \mathcal{K}_\delta^0$.

Note that including $\mathcal{K}_{\delta+1}$ in $\mathbf{C}'_\delta$ (that is, defining only $\mathbf{C}_\delta$) appears to be more natural, but because it is both an element of $\mathbf{C}_\delta$ and of $\mathbf{C}_{\delta+1}$, it becomes notationally awkward when taking unions of chains and computing the cardinality of such unions.

Lemma 243. $\mathbf{C}_\delta$ *is closed.*

Proof. Let D be a set of ordinals and $\mathbf{D} = \{\, \mathcal{K}_\delta^\alpha \mid \alpha \in D \,\}$. Being a set of ordinals D admits a minimal element α'. Then $\mathcal{K}_\delta^{\alpha'}$ is an upper bound for $\mathbf{D}$ and being part of it, must be the least upper bound.

Let β be the smallest ordinal not in D. If $\beta = \beta' + 1$ is a successor ordinal, then $\mathcal{K}_\delta^{\beta'}$ is both part of $\mathbf{D}$ and a lower bound, hence it's the greatest lower bound.

Suppose that β is a limit ordinal and that $\mathcal{K}_{\delta+1} \leq \mathcal{K}_\delta^\beta < \wedge\mathbf{D} \leq \mathcal{K}_\delta$. Then, there exists f, g such that $f \not\equiv_{\mathcal{K}_\delta^\beta} g$ but $f \equiv_{\wedge\mathbf{D}} g$ hence $f \equiv_{\mathcal{K}_\delta} g$. The later implies $[f]_\delta = [g]_\delta$, therefore we must have $[f]_\delta <_\delta S_{\delta,\beta}$ in order to have $[f]_\delta^\beta = [f]_{\delta+1} \neq [g]_{\delta+1} = [g]_\delta^\beta$.

Now, because β is limit, there exists γ with $[f]_\delta <_\delta S_{\delta,\gamma} < S_{\delta,\beta}$ and by definition of β, we can choose $\gamma \in D$. However, this implies $[f]_\delta^\gamma = [f]_{\delta+1} \neq [g]_{\delta+1} = [g]_\delta^\gamma$ and thus $f \not\equiv_{\mathcal{K}_\delta^\gamma} g$, contradicting $f \equiv_{\wedge\mathbf{D}} g$. Therefore, $\mathbf{C}_\delta$ is closed. □

For any $f \in S$ let $f_0 \in [f]_\delta$ (resp. $f_1 \in [f]_\delta$) be the binary sequence such that $f_0(\delta') = 0$ (resp. $f_1(\delta') = 1$) for all $\delta' \geq \delta$. Notice that $[f_0]_\delta = [f_1]_\delta$ but $[f_0]_{\delta+1} \neq [f_1]_{\delta+1}$.

For any $g \in [f]_\delta$, either $g(\delta) = 0$ and $[g]_{\delta+1} = [f_0]_{\delta+1}$, or $g(\delta) = 1$ and $[g]_{\delta+1} = [f_1]_{\delta+1}$. Thus, $[f]_\delta = [f_0]_{\delta+1} \cup [f_1]_{\delta+1}$.

Lemma 244. *For every* $\alpha < \omega_{2^{|\delta|}}$, $\mathcal{K}_\delta^{\alpha+1} \prec \mathcal{K}_\delta^\alpha$.

Proof. Fix δ and α. Let $f \in S$ and consider the block $[f]_\delta^\alpha$ of $\mathcal{K}_\delta^\alpha$. There are three cases:

1. If $[f]_\delta <_\delta S_{\delta,\alpha} <_\delta S_{\delta,\alpha+1}$ then $[f]_\delta^\alpha = [f]_\delta^{\alpha+1} = [f]_{\delta+1}$
2. If $S_{\delta,\alpha} <_\delta S_{\delta,\alpha+1} \leq_\delta [f]_\delta$ then $[f]_\delta^\alpha = [f]_\delta = [f]_\delta^{\alpha+1}$.
3. If $[f]_\delta = S_{\delta,\alpha} <_\delta S_{\delta,\alpha+1}$ then either $[f]_\delta^{\alpha+1} = [f_0]_{\delta+1}$ or $[f]_\delta^{\alpha+1} = [f_1]_{\delta+1}$.

(1) and (2) follow immediately from the definition. For case (3), we first obtain $[f]_\delta^\alpha = [f]_\delta$ and $[f]_\delta^{\alpha+1} = [f]_{\delta+1}$ by definition. But since $[f]_\delta = [f_0]_{\delta+1} \cup [f_1]_{\delta+1}$, we must have either $f \in [f_0]_{\delta+1}$ or $f \in [f_1]_{\delta+1}$. Hence $[f]_\delta^{\alpha+1} = [f]_{\delta+1}$ is either $[f_0]_{\delta+1}$ or $[f_1]_{\delta+1}$, as desired. By the above, every block of $\mathcal{K}_\delta^\alpha$ is a block of $\mathcal{K}_\delta^{\alpha+1}$ except for a single block (the unique block for which $[f]_\delta = S_{\delta,\alpha}$), which is obtained by taking the union of exactly two blocks of $\mathcal{K}_\delta^{\alpha+1}$. Hence, $\mathcal{K}_\delta^{\alpha+1} \prec \mathcal{K}_\delta^\alpha$. □

Lemma 245. $\mathbf{C}_\delta$ *is covering.*

Proof. Let $\mathcal{P} \prec_{\mathbf{C}_\delta} \mathcal{Q}$ be two partitions of $\mathbf{C}_\delta$. By definition, we must have $\mathcal{Q} = \mathcal{K}_\delta^\alpha$ for some α, therefore $\mathcal{P} = \mathcal{K}_\delta^{\alpha+1}$. By the previous lemma, we know that $\mathcal{K}_\delta^{\alpha+1} \prec \mathcal{K}_\delta^\alpha$, hence $\mathbf{C}_\delta$ is covering. □

Proposition 246. $\mathbf{C}_\delta$ *is a saturated chain in* Π_κ, *with minimum* $\mathcal{K}_{\delta+1}$ *and maximum* $\mathcal{K}_\delta$.

Proof. Applying previous results, we find that $\mathbf{C}_\delta$ is an endpoint-including (Corollary 242), closed (Lemma 243) and covering (Lemma 245) chain. Therefore, by Lemma 220, it is saturated. □

Completing the chain. In the following, let $\mathbf{C} = \{\bot\} \cup \left(\bigcup_{0 \leq \delta < \omega_\lambda} \mathbf{C}'_\delta\right)$.

Theorem 247. $\mathbf{C}$ *is a maximal chain in* Π_κ.

Proof. By Proposition 246, $\mathbf{C}_\delta = \mathbf{C}'_\delta \cup \{\mathcal{K}_{\delta+1}\}$ is a saturated chain with minimum $\mathcal{K}_{\delta+1}$ and maximum $\mathcal{K}_\delta$, whereby each $\mathbf{C}'_\delta$ is a chain. Noting that $\delta < \gamma$ implies $\mathcal{K}_\gamma < \mathcal{K}_\delta$ by Lemma 237, every element of $\mathbf{C}'_\delta$ is comparable to every element of $\mathbf{C}'_\gamma$. Hence, $\mathbf{C}$ is a chain.

Suppose that there exists a partition $\mathcal{P} \notin \mathbf{C}$ such that $\mathbf{C} \cup \{\mathcal{P}\}$ is still a chain. We obviously have $\mathcal{P} \notin \mathbf{K} \subset \mathbf{C}$ and because $\mathbf{K}$ is closed (Lemma 238) we can apply Lemma 219 and conclude that $\vee\mathbf{K}_\mathcal{P}^- \prec_\mathbf{K} \wedge\mathbf{K}_\mathcal{P}^+$. Because they are both keyframes, we must have $\wedge\mathbf{K}_\mathcal{P}^+ = \mathcal{K}_\beta$ for some ordinal, hence $\vee\mathbf{K}_\mathcal{P}^- = \mathcal{K}_{\beta+1}$. This entails $\mathcal{K}_{\beta+1} < \mathcal{P} < \mathcal{K}_\beta$, but we now by Proposition 246 that the chain is saturated between these two. Hence, $\mathbf{C}$ is maximal. □

Length of the chain. Recall that $\mathbf{C}'_\delta$ has cardinality $2^{|\delta|}$.

Lemma 248. *$\mathbf{C}$ has cardinality $\sum_{\delta<\omega_\lambda} 2^{|\delta|}$.*

Proof. $|\mathbf{C}| = \left|\bigcup_{\delta<\omega_\lambda} \mathbf{C}'_\delta \cup \{\bot\}\right| = \left|\bigcup_{\delta<\omega_\lambda} \mathbf{C}'_\delta\right| = \sum_{\delta<\omega_\lambda} |\mathbf{C}_\delta| = \sum_{\delta<\omega_\lambda} 2^{|\delta|}$ □

Lemma 249. *Let λ be a cardinal such that for any $\mu < \lambda$, we have $2^\mu < 2^\lambda$. Then $\sum_{\delta<\omega_\lambda} 2^{|\delta|} < 2^\lambda$.*

Proof. If $\delta < \omega_\lambda$, then $|\delta| < \lambda$, thus by hypothesis, $2^{|\delta|} < 2^\lambda$.

Assume now, for contradiction, that $\sum_{\delta<\omega_\lambda} 2^{|\delta|} \geq 2^\lambda$. Recall that $\mathrm{cf}(2^\lambda) = \inf\left\{ |I| \mid 2^\lambda = \left|\bigcup_{i\in I} A_i\right| \wedge \forall i \in I, |A_i| < 2^\lambda \right\}$, which is the same as $\mathrm{cf}(2^\lambda) = \inf\left\{ |I| \mid 2^\lambda = \sum_{i\in I} \mu_i \wedge \forall i \in I, \mu_i < 2^\lambda \right\}$.

It then follows that $\mathrm{cf}(2^\lambda) \leq |\omega_\lambda| = \lambda$. But by a standard consequence of König's Theorem [Kön05] we always have $\mathrm{cf}(2^\lambda) > \lambda$ under the Axiom of Choice. □

We can now prove our main result on short chains.

Theorem 250. *Let λ be an infinite cardinal such that for every cardinal $\mu < \lambda$ we have $2^\mu < 2^\lambda$. Then there exists a maximal chain of cardinality $< 2^\lambda$ (but $\geq \lambda$) in Π_{2^λ}.*

Proof. By Theorem 247 and Lemma 248, there is a maximal chain in Π_{2^λ} of cardinality $\sum_{\delta<\omega_\lambda} 2^{|\delta|}$. By Lemma 249 we have $\sum_{\delta<\omega_\lambda} 2^{|\delta|} < 2^\lambda$.

Consequently, the chain is of cardinality $< 2^\lambda$ (but $\geq \lambda$). □

For the case of $\lambda = \aleph_0$, we trivially have $2^n < 2^{\aleph_0}$ for all $n < \aleph_0$, and thus there is a countable maximal chain in $\Pi_{2^{\aleph_0}}$. The condition is also satisfied for all strong limit cardinals.

7 Antichains and maximal antichains

An *antichain* in $(\mathbf{P}, \leq)$ is a subset $\mathbf{A} \subseteq \mathbf{P}$ in which no two distinct elements of $\mathbf{P}$ are $\leq$-comparable. An antichain is *maximal* if adding an element to it results in a set that is not an antichain. Observe that in a poset, the *trivial* antichains $\{\bot\}$ and $\{\top\}$ are always maximal antichains.

There is no known tight bound on the cardinality of maximal antichains in Π_n for finite n, but some asymptotic results are known [Can98, BH02]. The cardinality of a maximal antichain in Π_n is $\Theta\left(n^a (\log n)^{-a-1/4} S(n, K_n)\right)$ where $a = (2 - e\log 2)/4$ and $S(n, K_n) = \max_k \left\{ {n \atop k} \right\}$ is the largest Stirling number of the second kind for fixed n.

Theorem 251. *Let κ be infinite. There is a maximal antichain of cardinality κ and a maximal antichain of cardinality 2^κ in Π_κ.*

Proof. Atoms of Π_κ are singular partitions with the non-singleton block containing only two elements and are thus in bijection with the set of two-element subsets of α, hence there are κ atoms. By atomicity of Π_κ, atoms are mutually incomparable (they form an anti-chain) and every other non-$\bot$ element lies over an atom (the anti-chain is maximal).

Similarly, co-atoms are two blocks partitions of the form $\{A, \kappa \setminus A\}$. By picking the subset not containing 0, they are in bijection with non-empty subsets of $\kappa \setminus \{0\}$, hence there are 2^κ co-atoms. By co-atomicity of Π_κ, co-atoms form a maximal anti-chain. □

Since Π_κ itself has cardinality 2^κ, there is no anti-chain of a bigger size. The proof that there is no maximal anti-chain shorter than κ is slightly more invoved.

Theorem 252. *No non-trivial maximal anti-chain in Π_κ has cardinality less than κ.*

Proof. Let $\lambda < \kappa$ and $\mathbf{A} = \{ \mathcal{A}_\delta \mid \delta < \omega_\lambda \}$ be a non-trivial anti-chain of cardinality λ in Π_κ. We will show that $\mathbf{A}$ is not maximal by building a partition out of it but yet not comparable with any of its elements.

Small partitions. For each $\mathcal{A}_\delta$, we build the set S_δ of all the elements that are not in a singleton block in it:

$$S_\delta = \{ x \in \kappa \mid \text{There exists } y \neq x \text{ with } x \equiv_{\mathcal{A}_\delta} y \}$$

Now, we call partition $\mathcal{A}_\delta$ *small* if $|S_\delta| \leq \lambda$, that is there are less than λ elements in all the non-singleton blocks of $\mathcal{A}_\delta$. Given an anti-chain $\mathbf{A}$, either it contains some small partitions or not.

No small partitions. Suppose that $\mathbf{A}$ contains no small partition. Since it is not trivial, we know that $\top \notin \mathbf{A}$. Hence, for each $\mathcal{A}_\delta \in \mathbf{A}$ we can pick an element x_δ which is not in the same block as 0: $0 \not\equiv_{\mathcal{A}_\delta} x_\delta$. Note that the x_δ are not necessarily distinct. Now, build the singular partition $\mathcal{P}$ whose non-singleton block is $P_0 = \{0\} \cup \{ x_\delta \mid \delta < \omega_\lambda \}$. We claim that $\mathcal{P} \notin \mathbf{A}$ but $\mathbf{A} \cup \{\mathcal{P}\}$ is still an anti-chain, hence $\mathbf{A}$ is not maximal.

Because P_0 has cardinality at most λ and all the rest is singletons, $\mathcal{P}$ is small. Thus it is not in $\mathbf{A}$ as we are in the case where $\mathbf{A}$ contains no small partition. Moreover, it is not possible that $\mathcal{A}_\delta \leq \mathcal{P}$. Indeed, because $\mathcal{P}$ contains only one non-singleton block, P_0, the comparison can hold only if P_0 contains S_δ, the union of all the non-singleton blocks of $\mathcal{A}_\delta$. But S_δ has cardinality larger than λ (because $\mathcal{A}_\delta$ is not small) while P_0 has cardinality at most λ. Hence, $S_\delta \not\subset P_0$ and $\mathcal{A}_\delta \nleq \mathcal{P}$.

Next, we have by construction $0 \equiv_{\mathcal{P}} x_\delta$ but $0 \not\equiv_{\mathcal{A}_\delta} x_\delta$. Thus, $\mathcal{P} \nleq \mathcal{A}_\delta$ for all $\delta < \omega_\lambda$.

So, if $\mathbf{A}$ contains no small partition, we can build a partition $\mathcal{P}$ that can expand the anti-chain, hence $\mathbf{A}$ is not a maximal anti-chain.

With small partitions. We now look at the case where $\mathbf{A}$ does contain small partitions. Let Δ be the set of indices of small partitions: $\Delta = \{\, \delta \mid |S_\delta| \leq \lambda \,\}$. Note that by construction, since $|\mathbf{A}| = \lambda$, we have $|\Delta| \leq \lambda$. Finally, let S be the union of all non-singleton blocks among all the small partitions in $\mathbf{A}$: $S = \bigcup_{\delta\in\Delta} S_\delta$.

Arithmetic. It follows from cardinal arithmetic that $|S| \leq \lambda$, indeed:

$$|S| = \left|\bigcup_{\delta\in\Delta} S_\delta\right| \leq \sum_{\delta\in\Delta} |S_\delta| \leq \sum_{\delta\in\Delta} \lambda = |\Delta| \times \lambda \leq \lambda \times \lambda = \lambda$$

Construction. Since $|S| \leq \lambda < \kappa$, we have $|\kappa \setminus S| = \kappa$. Thus, in $\kappa \setminus S$, we can pick λ different elements y_δ, one for each $\delta < \omega_\lambda$. Now, we claim that for each $\delta < \omega_\lambda$ we can find a $x_\delta \in S$ with $x_\delta \not\equiv_{\mathcal{A}_\delta} y_\delta$. Indeed, if that were not the case then the block of $\mathcal{A}_\delta$ containing y_δ would also contains the entirety of S and thus any of the small partitions in $\mathbf{A}$ would be smaller than $\mathcal{A}_\delta$, contradicting the fact that $\mathbf{A}$ is an anti-chain.

Note that the y_δ are distinct by hypothesis, and that $x_\delta \neq y_\delta$ because the former is in S and the latter is not. However, it is possible the x_δ are not distinct.

We now build the equivalence (and partition) $\mathcal{Q}$ generated by the relation $\{\, (x_\delta, y_\delta) \mid \delta < \omega_\lambda \,\}$. Because of the non unicity of the x_δ, $\mathcal{Q}$ may contain blocks with more than two elements, indeed transitivity may link some of the pairs together. However, because of the unicity of the y_δ, and the separation of the x_δ and y_δ, we know that each non-singleton block of $\mathcal{Q}$ contains exactly one x_δ and one or more y_δ.

Non-maximality. Let us now show that $\mathcal{Q}$ can extend $\mathbf{A}$. Firstly, because we have, by construction, $x_\delta \equiv_{\mathcal{Q}} y_\delta$ but $x_\delta \not\equiv_{\mathcal{A}_\delta} y_\delta$ that implies that $\mathcal{Q} \nleq \mathcal{A}_\delta$; notably $\mathcal{Q} \neq \mathcal{A}_\delta$, thus $\mathcal{Q} \notin \mathbf{A}$.

Next, consider a small $\mathcal{A}_\delta$. By construction of S, there is at least one non-singleton block of $\mathcal{A}_\delta$ that is contained in S (as S is precisely the union of such blocks for all the small partitions). On the other hand, each non-singleton block of $\mathcal{Q}$ contains exactly one x_δ and all its other elements are y_δ, chosen out of S. Hence, each block of $\mathcal{Q}$ contains at most one element of S and thus cannot contain any non-singleton block of $\mathcal{A}_\delta$, hence $\mathcal{A}_\delta \nleq \mathcal{Q}$.

Finally, consider a not small $\mathcal{A}_\delta$. By construction, the union of all the non-singleton blocks of $\mathcal{Q}$ is $\{\, x_\delta \mid \delta < \omega_\lambda \,\} \cup \{\, y_\delta \mid \delta < \omega_\lambda \,\}$ which has cardinality $\leq \lambda$ while the union of all non-singleton blocks of $\mathcal{A}_\delta$ has cardinality $> \lambda$ (as it is not a small partition). Thus, it is not possible that the non-singleton blocks of $\mathcal{A}_\delta$ are somehow contained in the non-singleton blocks of $\mathcal{Q}$ and $\mathcal{A}_\delta \nleq \mathcal{Q}$.

To conclude, $\mathcal{Q}$ is not in the anti-chain and is not comparable with its elements, hence $\mathbf{A}$ is not maximal.

Thus, any anti-chain of cardinality strictly less than κ is not maximal, and any maximal anti-chain in Π_κ has cardinality at least κ. □

The Theorems here tell us that any maximal anti-chain in it has cardinality between κ and 2^κ and since both of the bounds occur as cardinality of a maximal anti-chain, they are as tight as one can get. However, it is also possible to build maximal anti-chain of cardinality between these bounds.

Remark 253. Let $\lambda < \kappa$ be two infinite cardinals. Given a maximal anti-chain $\mathbf{A}$ in Π_λ, we can construct a maximal anti-chain $\mathbf{B}$ in Π_κ as follows:

For each partition $\mathcal{A} \in \mathbf{A}$, let $\mathcal{A}'$ be a partition of κ constructed from $\mathcal{A}$ by adding all the singleton blocks $\{x\}, x \in \kappa \setminus \lambda$ and let $\mathbf{A}'$ be the collection of all these. Next, let $\mathcal{P}_{\alpha,\beta}$ be the singular partition of Π_κ with non-singleton block $\{\alpha, \beta\}$ and define

$$\mathbf{B} = \mathbf{A}' \cup \{\, \mathcal{P}_{\alpha,\beta} \mid \alpha < \beta < \omega_\kappa \text{ and } \lambda \leq |\beta| \,\}$$

B is an anti-chain. Indeed, the $\mathcal{A}'$ are mutually incomparable because $\mathbf{A}$ is an anti-chain; the $\mathcal{P}_{\alpha,\beta}$ are mutually incomparable by construction; and a $\mathcal{A}'$ is not comparable with a $\mathcal{P}_{\alpha,\beta}$ because each of the non-singleton blocks of the former is included in λ while the non-singleton block of the latter contains an element of $\kappa \setminus \lambda$ (as we have only taken the β with $|\beta| \geq \lambda$).

B is maximal. Indeed, consider a partition $\mathcal{Q}'$ of Π_κ which is not in $\mathbf{B}$. If the elements of $\kappa \setminus \lambda$ are all in singleton blocks in $\mathcal{Q}'$ then its restriction to λ, $\mathcal{Q}$, is comparable with some $\mathcal{A}$ (by maximality of $\mathbf{A}$), and so $\mathcal{Q}'$ is comparable with $\mathcal{A}'$. On the other hand, if there is a non singleton block containing α, β with $\beta \in \kappa \setminus \lambda$ we immediately have $\mathcal{P}_{\alpha,\beta} \leq \mathcal{Q}$.

Because $\lambda < \kappa$, there are $\kappa - \lambda = \kappa$ different β with $\lambda \leq |\beta|$, thus there are κ different $\mathcal{P}_{\alpha,\beta}$ in $\mathbf{B}$. Because the $\mathcal{A}'$ and $\mathcal{P}_{\alpha,\beta}$ are distinct, $\mathbf{B}$ has cardinality $|\mathbf{A}'| + \kappa = \max(\kappa, |\mathbf{A}|)$.

Since we know, by Theorem 251, that we can build a maximal anti-chain of cardinality 2^λ in Π_λ, we can use this construction to build maximal anti-chains of cardinality $\max(\kappa, 2^\lambda)$ in Π_κ, for any value of $\lambda < \kappa$. Thus, we can possibly build maximal anti-chains of cardinalities other than κ or 2^κ.

8 Complements

Recall that in a bounded lattice L, elements $a, b \in L$ are *complements* if and only if $a \vee b = \top$ and $a \wedge b = \bot$. We denote by $\mathrm{compl}(\mathcal{P})$ the set of all complements to $\mathcal{P}$ in Π_κ. For finite $\kappa = n$, counting the number of elements in $\mathrm{compl}(\mathcal{P})$ is a difficult combinatorial problem. The best known estimate, due to Grieser [Gri91], is that if $\mathcal{P} = \{B_1, \dots, B_m\}$ is a partition in Π_n, then the number of complements $\mathcal{Q}$ of $\mathcal{P}$ satisfying $|\mathcal{Q}| = n - m + 1$ is $\prod_{i=1}^m |B_i| \cdot (n - m + 1)^{m-2}$.

In the following, we prove a succession of lemmas leading up to the main result of the section, Theorem 260. which gives a complete characterization of the counts of complements to partitions of infinite cardinals.

For later use, we first recall some fundamental results on cardinal arithmetic:

Lemma 254. *Let $(\kappa_i)_{i\in I}$ be a family of cardinals. The following hold:*

1. ([HSW99], Lemma 1.6.3(b.i)) *If $\kappa_i > 0$ for all $i \in I$, $I \neq \emptyset$ and at least one of the cardinals $|I|$ and κ_i (for some $i \in I$) is infinite, then*
$$\sum_{i\in I} \kappa_i = \max\{|I|, \sup\{\, \kappa_i \mid i \in I \,\}\} = |I| \cdot \sup\{\, \kappa_i \mid i \in I \,\}$$
2. ([HSW99], Lemma 1.6.15(a), Tarski), *If $\lambda \geq \aleph_0$ is a cardinal and $(\kappa_\alpha)_{\alpha<\lambda}$ is an increasing sequence of infinite cardinals, then*
$$\prod_{\alpha<\lambda} \kappa_\alpha = (\sup\{\, \kappa_\alpha \mid \alpha < \lambda \,\})^\lambda$$
3. ([HSW99], Lemma 1.6.15(d), Tarski) *If κ is an infinite cardinal, then*
$$2^\kappa = \left(\sup_{\lambda<\kappa} 2^\lambda\right)^{\mathrm{cf}(\kappa)}$$

Lemma 255. *Let κ be infinite and $\mathcal{P} \notin \{\bot, \top\}$ be a partition of κ. Then, $|\mathrm{compl}(\mathcal{P})| \geq 2^{|\mathcal{P}|}$.*

Proof. If $|\mathcal{P}|$ is finite, then there must exist at least one block B of cardinality κ in $\mathcal{P}$ (otherwise, since $\cup\mathcal{P} = \kappa$, we would have $\mathrm{cf}(\kappa)$ finite). Any singular partition whose non-singleton block contains exactly one element from each block of $\mathcal{P}$ is a complement to $\mathcal{P}$. Since there are κ choices for the element in B, there are at least κ complements to $\mathcal{P}$. Thus $|\mathrm{compl}(\mathcal{P})| \geq \kappa > 2^{|\mathcal{P}|}$.

Assume now that $|\mathcal{P}|$ is infinite. Since $\mathcal{P} \neq \bot$, we can choose a block B_0 from $\mathcal{P}$ containing distinct elements $\iota \neq \upsilon$. Write $\mathcal{P}' = \mathcal{P} \setminus \{B_0\}$, and note that $|\mathcal{P}'| = |\mathcal{P}|$. By the Axiom of Choice, we may select from each block $B \in \mathcal{P}'$ an element $\gamma(B)$. Given any subset $\mathcal{P}_1 \subseteq \mathcal{P}'$, let $\mathcal{P}_2 = \mathcal{P}' \setminus \mathcal{P}_1$. If we now define

$$\begin{aligned} Q_1 &= \{\iota\} \cup \{\, \gamma(B) \mid B \in \mathcal{P}_1 \,\} \\ Q_2 &= \{\upsilon\} \cup \{\, \gamma(B) \mid B \in \mathcal{P}_2 \,\} \\ \mathcal{Q}_s &= \{\, \{\gamma\} \mid \gamma \in \kappa \setminus (Q_1 \cup Q_2) \,\} \end{aligned}$$

then $\mathcal{Q} = \{Q_1, Q_2\} \cup \mathcal{Q}_s$ is a complement to $\mathcal{P}$, as can be verified as follows: (i) $\mathcal{Q}$ is a partition, since each element is included in exactly one block. (ii) $\mathcal{P} \wedge \mathcal{Q} = \bot$, since each block $Q \in \mathcal{Q}$ contains at most one element from each block in $\mathcal{P}$. (iii) Finally, $\mathcal{P} \vee \mathcal{Q} = \top$, i.e. $x \equiv_{\mathcal{P}\vee\mathcal{Q}} y$ for all $x, y \in \kappa$: Consider $x, y \in \kappa$, and write B_x, B_y for the blocks in $\mathcal{P}$ containing x and y, respectively. If $B_x, B_y \in \mathcal{P}_1 \cup \{B_0\}$, then there exists[3] a $\gamma_x \in Q_1 \cap B_x$ and a $\gamma_y \in Q_1 \cap B_y$. This yields

$$x \underset{B_x}{\equiv} \gamma_x \underset{Q_1}{\equiv} \gamma_y \underset{B_y}{\equiv} y$$

If instead $B_x \in \mathcal{P}_1 \cup \{B_0\}$ and $B_y \in \mathcal{P}_2$, there is a $\gamma_x \in Q_1 \cap B_x$, whereby

$$x \underset{B_x}{\equiv} \gamma_x \underset{Q_1}{\equiv} \iota \underset{B_0}{\equiv} \upsilon \underset{Q_2}{\equiv} \gamma(B_y) \underset{B_y}{\equiv} y$$

The remaining two cases, $B_x, B_y \in \mathcal{P}_2 \cup \{B_0\}$ and $B_x \in \mathcal{P}_2 \cup \{B_0\}$, $B_y \in \mathcal{P}_1$, are symmetrical. Thus $\mathcal{Q} \in \mathrm{compl}(\mathcal{P})$. Clearly, two different choices of the subset $\mathcal{P}_1 \subseteq \mathcal{P}'$ yields different complements $\mathcal{Q}$, whereby $|\mathrm{compl}(\mathcal{P})| \geq 2^{|\mathcal{P}|}$. □

[3]Namely, $\gamma_x = \gamma(B_x)$ if $B_x \in \mathcal{P}_1$ and $\gamma_x = \iota$ if $B_x = B_0$.

Lemma 256. *Let κ be infinite and $\mathcal{P} \notin \{\bot, \top\}$ be a partition of κ. If $\mathcal{P}$ contains no block of cardinality κ, then $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$*

Proof. If $\mathcal{P}$ contains no block of cardinality κ, then $|\mathcal{P}| \geq \mathrm{cf}(\kappa)$ (because $\kappa = \bigcup_{B\in\mathcal{P}} B$). Then Lemma 255 implies $|\mathrm{compl}(\mathcal{P})| \geq 2^{|\mathcal{P}|} \geq 2^{\mathrm{cf}(\kappa)}$. Thus, if κ is a regular cardinal, or if $|\mathcal{P}| = \kappa$, we immediately obtain $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$.

Assume now that κ is singular and $\mathrm{cf}(\kappa) \leq |\mathcal{P}| < \kappa$. We can construct a complement $\mathcal{Q}$ to $\mathcal{P}$ as follows:

1. Let A_0 be a set containing exactly one element $\gamma(B)$ from each block B of $\mathcal{P}$. Let $\mathcal{P}' = \{\, B \setminus A_0 \mid B \in \mathcal{P} \,\} \setminus \{\emptyset\}$, whereby $\cup\mathcal{P}' = \kappa \setminus A_0$.

2. Any partition $\mathcal{Q}$ of κ that contains A_0 as a block will have $\mathcal{P} \vee \mathcal{Q} = \top$: If $\beta \in B_1$ and $\delta \in B_2$ for $B_1, B_2 \in \mathcal{P}$, then $\beta \equiv_{B_1} \gamma(B_1) \equiv_{A_0} \gamma(B_2) \equiv_{B_2} \delta$. Hence, if a partition $\mathcal{Q}'$ of $\kappa \setminus A_0$ satisfies $|A' \cap B'| \leq 1$ for all $A' \in \mathcal{Q}', B' \in \mathcal{P}'$, then $\mathcal{Q} = \{A_0\} \cup \mathcal{Q}'$ is a complement to $\mathcal{P}$.

3. Because $|A_0| = |\mathcal{P}| < \kappa$, we have $|\kappa \setminus A_0| = \kappa$, and hence $\sum_{B\in\mathcal{P}'} |B| = |\cup\mathcal{P}'| = \kappa$. By Lemma 254(1), and using $|\mathcal{P}'| < \kappa$, we obtain
$$\kappa = \sum_{B\in\mathcal{P}'} |B| = \max\left\{ |\mathcal{P}'|\, , \sup_{B\in\mathcal{P}'} |B| \right\} = \sup_{B\in\mathcal{P}'} |B|$$

4. By definition of cofinality, there exists an increasing sequence $(\mu_\alpha)_{\alpha<\mathrm{cf}(\kappa)}$ of infinite cardinals strictly less than κ that sums to κ. Because κ is singular, Lemma 254(1) implies $\kappa = \sup_{\alpha<\mathrm{cf}(\kappa)} \mu_\alpha$.

   Since also $\kappa = \sup\{\, |B| \mid B \in \mathcal{P}' \,\}$, we can choose by AC for every $\alpha < \mathrm{cf}(\kappa)$ a $B_\alpha \in \mathcal{P}'$ such that $|B_\alpha| \geq \mu_\alpha$. The sequence $(|B_\alpha|)_{\alpha<\mathrm{cf}(\kappa)}$ clearly has supremum κ.

5. For every successor ordinal $\alpha + 1 < \mathrm{cf}(\kappa)$, split the block $B_{\alpha+1}$ into a small subset $B^-_{\alpha+1}$ of cardinality $|B_\alpha|$ and a large subset $B^+_{\alpha+1}$ of cardinality $|B_{\alpha+1}|$. Then for every limit ordinal α, define $B^+_\alpha = B_\alpha$. Now choose for every ordinal $\alpha < \mathrm{cf}(\kappa)$ a bijection $\sigma_\alpha \colon B^+_\alpha \to B^-_{\alpha+1}$; let $A^\beta_\alpha = \{\beta, \sigma_\alpha(\beta)\}$ and let $A_\alpha = \left\{\, A^\beta_\alpha \mid \beta \in B^+_\alpha \,\right\}$, it is a partition of $B^+_\alpha \cup B^-_{\alpha+1}$, consisting of two-element blocks.

6. Finally, let
$$S = \kappa \setminus \left(A_0 \cup \bigcup_{\alpha<\mathrm{cf}(\kappa)} B_\alpha \right).$$
   Then it is easy to verify that
$$\mathcal{Q} = \{\, \{\beta\} \mid \beta \in S \,\} \cup \{A_0\} \cup \bigcup_{\alpha<\mathrm{cf}(\kappa)} A_\alpha$$
   is a complement to $\mathcal{P}$. Indeed, each element of A_α contains exactly two elements from two different blocks B_α and $B_{\alpha+1}$ of $\mathcal{P}'$, and the other blocks are either A_0 or singletons. Thus, for any $A' \in \mathcal{Q}$ other than A_0 and any $B' \in \mathcal{P}'$, we have $|A' \cap B'| \leq 1$, and by Point (2) above, $\mathcal{Q}$ is a complement to $\mathcal{P}$.

For $\sigma_\alpha \neq \sigma'_\alpha$ there exists $\beta \in B^+_\alpha$ with $\sigma_\alpha(\beta) \neq \sigma'_\alpha(\beta)$ and hence $\{\beta, \sigma_\alpha(\beta)\} \neq \{\beta, \sigma'_\alpha(\beta)\}$, whence each choice of $(\sigma_\alpha)_{\alpha<\mathrm{cf}(\kappa)}$ yields a distinct complement. There are $\left|B^-_{\alpha+1}\right|^{\left|B^+_\alpha\right|} = |B_\alpha|^{|B_\alpha|} = 2^{|B_\alpha|}$ ways to choose each bijection σ_α. Since $\kappa = \sup_{\alpha<\mathrm{cf}(\kappa)} |B_\alpha|$, for each $\lambda < \kappa$, there exists α with $|B_\alpha| > \lambda$ hence $2^{|B_\alpha|} \geq 2^\lambda$. Consequently, $\sup_{\alpha<\mathrm{cf}(\kappa)} 2^{|B_\alpha|} \geq \sup_{\lambda<\kappa} 2^\lambda$. Then, Lemma 254(2) and (3) yields
$$|\mathrm{compl}(\mathcal{P})| \geq \prod_{\alpha<\mathrm{cf}(\kappa)} 2^{|B_\alpha|} = \left(\sup_{\alpha<\mathrm{cf}(\kappa)} 2^{|B_\alpha|} \right)^{\mathrm{cf}(\kappa)} \geq \left(\sup_{\lambda<\kappa} 2^\lambda \right)^{\mathrm{cf}(\kappa)} = 2^\kappa$$

As $2^\kappa = |\Pi_\kappa|$ is an upper bound to the number of complements to $\mathcal{P}$, we obtain $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$. □

Lemma 257. *Let κ be infinite and $\mathcal{P} \in \Pi_\kappa$ be any partition of κ. If $\mathcal{P}$ contains a block B of cardinality κ, then $|\mathrm{compl}(\mathcal{P})| = \kappa^\lambda$, where $\lambda = |\kappa \setminus B|$.*

Proof. Assume that $\mathcal{P}$ has a block B of cardinality κ, and write $\mathcal{P}$ as the disjoint union $\mathcal{P} = \{B\} \cup \mathcal{P}'$. Let $\bar{B} = \kappa \setminus B$, and observe that $\bar{B} = \cup\mathcal{P}'$. Denote $\lambda = |\cup\mathcal{P}'| = \left|\bar{B}\right| = |\kappa \setminus B| \leq \kappa$. If $\left|\bar{B}\right| = 0$, then $\mathcal{P} = \top$, which has exactly one complement, namely $\bot$, whereby $|\mathrm{compl}(\mathcal{P})| = 1 = \kappa^0$ as desired. Hence, in the following we can assume that $\mathcal{P} \neq \top$, such that $1 \leq \lambda \leq \kappa$.

Lower bound. To show $|\mathrm{compl}(\mathcal{P})| \geq \kappa^\lambda$, choose any injection $\sigma \colon \bar{B} \to B$, and write $B_\lambda = \sigma\left(\bar{B}\right)$, $B_s = B \setminus B_\lambda$. If we now define $A_\alpha = \{\alpha, \sigma(\alpha)\}$, then the partition $\mathcal{Q} = \left\{\, A_\alpha \mid \alpha \in \bar{B} \,\right\} \cup \{\, \{\beta\} \mid \beta \in B_s \,\}$ is a complement to $\mathcal{P}$, as is easily verified by checking each of the properties: (i) Every element of κ is either in $\bar{B}$, B_λ, or B_s, hence $\mathcal{Q}$ is a partition; (ii) each block of $\mathcal{Q}$ is either a singleton (from B_s) or a doubleton (one of the A_α) with one element in B and one out, hence it intersects each block of $\mathcal{P}$ in at most one point; (iii) any α and α' are linked through their images $\sigma(\alpha)$ and $\sigma(\alpha')$ in B.

There are κ^λ ways of choosing σ, and each way leads to a distinct complement. Hence, $|\mathrm{compl}(\mathcal{P})| \geq \kappa^\lambda$.

Upper bound. Observe that we have $|\bar{B}| \geq |\mathcal{P}'|$. Let now $\mathcal{Q}$ be any complement to $\mathcal{P}$. Then

1. $\mathcal{P} \wedge \mathcal{Q} = \bot$ implies that for every $A \in \mathcal{Q}$ and $B \in \mathcal{P}$, $|A \cap B| \leq 1$, and hence $|A| \leq |\mathcal{P}| \leq \lambda$.

2. Let $\mathcal{Q}' = \{\, A' \in \mathcal{Q} \mid |A'| \geq 2 \,\}$ be the set of non-singleton blocks of $\mathcal{Q}$. Since $|A' \cap B| \leq 1$, each A' must intersect $\bar{B}$ in at least one point. As blocks of a partition, the A' are pairwise disjoint, as are consequently these intersections. Hence choosing (by AC) an element in each $A' \cap \bar{B}$ defines an injection from $\mathcal{Q}'$ to $\bar{B}$, whereby $|\mathcal{Q}'| \leq \bar{B} \leq \lambda$.

3. An upper bound for the number of complements to $\mathcal{P}$ can then be found in the following way: Specifying $\mathcal{Q}'$ uniquely determines the complement $\mathcal{Q}$. $\mathcal{Q}$ has $|\mathcal{Q}'| \leq \lambda$ non-singleton blocks, each of size at most $|\mathcal{P}| \leq \lambda$, yielding $|\bigcup \mathcal{Q}'| \leq |\lambda \times \lambda| = \lambda$. A complement $\mathcal{Q}$ is fully specified by (i) the union $\bigcup \mathcal{Q}'$ of the non-singleton blocks; and (ii) the partition of these unions into blocks of $\mathcal{Q}'$. Write $\epsilon = |\bigcup \mathcal{Q}'|$. For each cardinality ϵ that $\bigcup \mathcal{Q}'$ can attain, there are at most $\lambda^\epsilon \leq \kappa^\epsilon$ ways to select the elements $\bigcup \mathcal{Q}'$ from $\bar{B}$. Since $\mathcal{Q}'$ is a partition of ϵ, there are at most $|\Pi_\epsilon| = 2^\epsilon$ distinct ways to partition these elements into blocks of $\mathcal{Q}'$. Letting now ϵ range over all potentially allowed cardinalities, i.e. all less than or equal λ, we find

$$|\mathrm{compl}(\mathcal{P})| \leq \sum_{\epsilon \leq \lambda} 2^\epsilon \cdot \lambda^\epsilon \leq \sum_{\epsilon \leq \lambda} 2^\epsilon \cdot \kappa^\epsilon = \sum_{\epsilon \leq \lambda} \kappa^\epsilon = \kappa^\lambda$$

Observe that the above applies both to finite and infinite $\lambda = |\kappa \setminus B|$. □

Corollary 258. *Let κ be infinite and $\mathcal{P} \notin \{\bot, \top\}$ be a partition of κ. If $\mathcal{P}$ contains two or more blocks of cardinality κ, then $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$.*

Proof. If two blocks, B_1 and B_2, have cardinality κ, then $|\kappa \setminus B_1| = \kappa$, and Lemma 257 yields $|\mathrm{compl}(\mathcal{P})| \geq \kappa^\kappa = 2^\kappa$. □

Lemma 259. *Let κ be infinite and $\mathcal{P} \notin \{\bot, \top\}$ be a partition of κ. Then, $\kappa \leq |\mathrm{compl}(\mathcal{P})| \leq 2^\kappa$.*

Proof. As $|\Pi_\kappa| = 2^\kappa$, the upper bound is immediate, and it suffices to prove the lower bound. There are two cases to consider: First, if $\mathcal{P}$ contains no block of cardinality κ, Lemma 256 yields $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$. Otherwise, if $\mathcal{P}$ does contain a block B of cardinality κ, Lemma 257 implies $|\mathrm{compl}(\mathcal{P})| = \kappa^{|\kappa \setminus B|}$. Because $\mathcal{P} \neq \top$, we have $|\kappa \setminus B| \geq 1$, whereby $|\mathrm{compl}(\mathcal{P})| \geq \kappa$. In both cases, $|\mathrm{compl}(\mathcal{P})| \geq \kappa$. □

We can now state our main theorem:

Theorem 260. *Let κ be infinite and let $\mathcal{P} \notin \{\bot, \top\}$ be a partition in Π_κ. Then*

1. $\kappa \leq |\mathrm{compl}(\mathcal{P})| \leq 2^\kappa$.

2. $|\mathrm{compl}(\mathcal{P})| \geq 2^{|\mathcal{P}|}$.

3. *If $\mathcal{P}$ contains no block of cardinality κ, then $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$.*

4. *If $\mathcal{P}$ contains a block B of cardinality κ, then $|\mathrm{compl}(\mathcal{P})| = \kappa^{|\kappa \setminus B|}$.*

5. *If $\mathcal{P}$ contains two or more blocks of cardinality κ, then $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$.*

Proof. (1)–(4) are Lemmas 259, 255, 256 and 257, respectively. (5) is Corollary 258. □

A consequence of Theorem 260 is that partitions with fewer complements than 2^κ must always have exactly one large block, and a sufficiently small number of elements remaining after removing it:

Corollary 261. *If $|\mathrm{compl}(\mathcal{P})| < 2^\kappa$, then $\mathcal{P}$ contains exactly one block B of size κ, and $|\kappa \setminus B| < \kappa$.*

For any infinite cardinal κ, both κ and 2^κ can be realized as $|\mathrm{compl}(\mathcal{P})|$ for some partition $\mathcal{P}$. In fact, Theorem 260 provides a complete characterization of the cardinals that can be realized as complement counts: it is precisely those cardinals of the form κ^λ, with $0 \leq \lambda \leq \kappa$ (and $1 \leq \lambda \leq \kappa$ when considering only non-trivial partitions), as the following corollary shows:

Corollary 262. *For any infinite cardinality κ, and every cardinal $0 \leq \lambda \leq \kappa$, there is a partition $\mathcal{P}_\lambda$ of κ for which $|\mathrm{compl}(\mathcal{P}_\lambda)| = \kappa^\lambda$.*

In particular, there exist partitions $\mathcal{P}$ and $\mathcal{R}$ in Π_κ with $|\mathrm{compl}(\mathcal{P})| = \kappa$, respectively $|\mathrm{compl}(\mathcal{R})| = 2^\kappa$.

No cardinal that is not of the form κ^λ can be realised as $|\mathrm{compl}(\mathcal{P})|$.

Proof. Theorem 260 fully describes the possible number of complements of non-trivial partition. Remembering that $\kappa^\kappa = 2^\kappa$ and that, for $\top$ and $\bot$, the number of complements is $1 = \kappa^0$, it is apparent that for any partition, the number of complements has the form κ^λ. Conversely, the cardinal κ^λ, $1 \leq \lambda \leq \kappa$, can be realised as the number of complements to the doubleton partition $\{B, \bar{B}\}$ with $|B| = \lambda$ and $|\bar{B}| = \kappa$, using Theorem 260(4). ☐

8.1 Orthocomplements

It is well-known that Π_κ is relatively complemented. To our knowledge, it has so far been unknown whether there exists an $n > 2$ such that Π_n is orthocomplemented. We prove that for any cardinality $\kappa > 2$ (finite or transfinite), Π_κ is *not* orthocomplemented.

Proposition 263 (Orthocomplements)**.** *If $\kappa > 2$, then Π_κ is not orthocomplemented.*

Proof. Orthocomplementation yields a bijection between atoms and co-atoms. Atoms are singular partitions whose non-singleton block is a pair, hence there are as many atoms as there are pairs, namely $\frac{\kappa\times(\kappa-1)}{2}$. Co-atoms are partitions with two blocks, a non-trivial subset of κ and its complement, hence there are as many co-atoms as half the number of non-trivial subsets, namely $\frac{2^\kappa-2}{2}$. For $\kappa \geq 3$, these numbers are different hence Π_κ cannot be orthocomplemented. For $\kappa = 3$, one can easily check that Π_3 is not orthocomplemented either (because it contains an odd number of elements). ☐

9 Results under GCH

Under the Generalized Continuum Hypothesis, the results from the previous sections all simplify greatly, and it is possible to obtain much stronger results.

9.1 Maximal chains under GCH

Under GCH, we can fully determine the possible cardinals for maximal chains in Π_κ, as we will see in Theorem 267 below.

Lemma 264. *Let κ be an infinite cardinal, let $\mathbf{C}$ be a maximal chain in Π_κ Given a partition $\mathcal{P} \in \mathbf{C}$, write $\mathcal{P}^+ = \wedge\mathbf{C}^+_\mathcal{P}$ (cf. Definition 218), and let*

$$\mathbf{C}^* = \left\{ \mathcal{P} \in \mathbf{C} \;\middle|\; \mathcal{P} \prec \mathcal{P}^+ \right\}$$

Then $\kappa \leq 2^{|\mathbf{C}^*|}$.

Proof. First note that, being maximal, $\mathbf{C}$ is end-point including, closed and covering by Lemma 220. Let $\mathcal{P}^-_{\alpha,\beta} = \vee\mathbf{C}^-_{\alpha,\beta}$ (cf. Definition 221). By Lemma 222, we have $\mathcal{P}^-_{\alpha,\beta} \prec \wedge\mathbf{C}^+_{\alpha,\beta}$ and by construction, $\mathbf{C}^+_{\alpha,\beta} = \mathbf{C}^+_{\mathcal{P}^-_{\alpha,\beta}}$. Hence, $\mathcal{P}^-_{\alpha,\beta} \in \mathbf{C}^*$.

We next construct a family of maps $\varphi_\alpha \colon \mathbf{C}^* \to \{0,1\}$ in the following way: Let $\mathcal{P} \in \mathbf{C}^*$. By definition, we have $\mathcal{P} \prec \mathcal{P}^+$, thus there exist an unique block $B_\mathcal{P}$ of $\mathcal{P}^+$ which is the union of two blocks of $\mathcal{P}$ (and all others are also blocks of $\mathcal{P}$). Choose, by axiom of choice, one of these two blocks as $B_{\mathcal{P},0}$.

Now, for each $\alpha \in \kappa$, we define $\varphi_\alpha \colon \mathbf{C}^* \to \{0,1\}$ by

$$\varphi_\alpha(\mathcal{P}) = \begin{cases} 0 & \text{if } \alpha \in B_{\mathcal{P},0} \\ 1 & \text{otherwise} \end{cases}$$

The previous construction shows that for any $\alpha \neq \beta$, (i) $\mathcal{P}^-_{\alpha,\beta} \in \mathbf{C}^*$; (ii) α and β are precisely in $B_{\mathcal{P}^-_{\alpha,\beta}}$; and (iii) they are in different blocks of $\mathcal{P}^-_{\alpha,\beta}$, that is one of them is in $B_{\mathcal{P}^-_{\alpha,\beta},0}$ and the other is not. Hence, $\varphi_\alpha(\mathcal{P}^-_{\alpha,\beta}) \neq \varphi_\beta(\mathcal{P}^-_{\alpha,\beta})$ and consequently, $\varphi_\alpha \neq \varphi_\beta$. Thus, the map $\alpha \mapsto \varphi_\alpha$ is injective from κ to $\{0,1\}^{\mathbf{C}^*}$, whereby $\kappa \leq 2^{|\mathbf{C}^*|}$. ☐

Proposition 265. *Let κ be any infinite cardinal. Under GCH, any maximal chain $\mathbf{C}$ in Π_κ has cardinality*

$$|\mathbf{C}| \geq \begin{cases} \kappa^- & \textit{if } \kappa \textit{ is a successor cardinal,} \\ \kappa & \textit{if } \kappa \textit{ is a limit cardinal.} \end{cases}$$

Proof. By Lemma 264, $\kappa \leq 2^{|\mathbf{C}^*|} \leq 2^{|\mathbf{C}|}$ for any maximal chain $\mathbf{C}$. Under GCH, $|\mathbf{C}|^+ = 2^{|\mathbf{C}|}$, whereby $\kappa \leq |\mathbf{C}|^+$. Assume now that $|\mathbf{C}| < \kappa$, i.e. $|\mathbf{C}| < \kappa \leq |\mathbf{C}|^+$. By definition of the successor relation, this implies $\kappa = |\mathbf{C}|^+$.

If κ is a limit cardinal, this is a contradiction, and so we must have $|\mathbf{C}| \geq \kappa$. If κ is a successor cardinal, it follows that $|\mathbf{C}| \geq \kappa^-$. □

By combining the previous results, we can tightly bound the cardinal of any maximal chain in Π_κ under GCH. In addition, the following simple restriction of Theorems 235 and 250 to the case when GCH is assumed, provides instances of long and short maximal chains that realize the bounds established above.

Corollary 266. *Let κ be a infinite cardinal. Under GCH, there exists a maximal chain of length κ^+ in Π_κ; and there exists a chain of length κ in Π_{κ^+}.*

Proof. If GCH is assumed, the first statement is an immediate consequence of Theorem 235; and the second statement follows, because the precondition in Theorem 250, $\lambda < \kappa$ only if $2^\lambda < 2^\kappa$, is always satisfied under GCH. □

Theorem 267. *Let κ be an infinite cardinal. Under GCH, the cardinality of any maximal chain in Π_κ is:*

- *κ^-, κ, or κ^+ (and all three are always achieved) if κ is a successor; and*
- *either κ or κ^+ (and both are achieved) if κ is a limit cardinal.*

Proof. The lower bounds are given by Proposition 265, and the upper bound is the cardinality of Π_κ itself.

Furthermore, each possible value is always realized: For any infinite cardinal κ, there always exists a well-founded maximal chain of cardinality κ. By Corollary 266, there also exists a chain of length κ^+, and for successor cardinals κ, a chain of length κ^-. □

The bounds can also be stated in a symmetrical fashion:

Corollary 268. *For any infinite cardinal κ (successor or limit), the cardinality of any maximal chain lies between* $\sup\{\lambda < \kappa\}$ *and* $\inf\{\lambda > \kappa\}$.

9.2 Maximal antichains under GCH

Corollary 269. *Under GCH, when κ is an infinite cardinal, the length of any maximal antichain in Π_κ is either κ or κ^+, and both are realized.*

Proof. This is the content of Theorems 251, 252, together with the fact that $|\Pi_\kappa| = 2^\kappa$, when the assumption $2^\kappa = \kappa^+$ is made. □

9.3 Complements under GCH

Under GCH, the number of complements $|\mathrm{compl}(\mathcal{P})|$ for partitions $\mathcal{P} \notin \{\top, \bot\}$ of an infinite cardinal κ is either κ or $\kappa^+ = 2^\kappa$. In addition, the simplified rules for arithmetic under GCH strengthen Theorem 260:

Theorem 270. *Let κ be an infinite cardinal, and $\mathcal{P} \notin \{\top, \bot\}$ be a partition of κ. Assuming GCH, then*

$$|\mathrm{compl}(\mathcal{P})| = \begin{cases} \kappa & \textit{if and only if exactly one block } B \in \mathcal{P} \textit{ has } |B| = \kappa, \\ & \textit{and } |\kappa \setminus B| < \mathrm{cf}(\kappa) \\ 2^\kappa & \textit{otherwise} \end{cases}$$

Proof. Consider the three cases:

1. First, if $\mathcal{P}$ contains either zero or at least two blocks of size κ. Then Theorem 260(2) or 260(5) yields $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$.

2. Next, assume $\mathcal{P}$ contains exactly one block B of size κ, and $|\kappa \setminus B| \geq \mathrm{cf}(\kappa)$. By Theorem 260(4), $|\mathrm{compl}(\mathcal{P})| = \kappa^{|\kappa\setminus B|} \geq \kappa^{\mathrm{cf}(\kappa)} > \kappa$. GCH, together with $|\mathrm{compl}(\mathcal{P})| \leq 2^\kappa$, then yields $|\mathrm{compl}(\mathcal{P})| = 2^\kappa$.

3. Finally, if $\mathcal{P}$ contains exactly one block B of size κ in $\mathcal{P}$, and $|\kappa \setminus B| < \mathrm{cf}(\kappa)$. Under GCH, $\kappa^\lambda = \kappa$ if and only if $1 \leq \lambda < \mathrm{cf}(\kappa)$. Together with Theorem 260(4) and $\mathcal{P} \neq \top$, this yields $|\mathrm{compl}(\mathcal{P})| = \kappa^{|\kappa\setminus B|} = \kappa$.

□

In Theorem 270, the assumption of GCH is necessary in the sense that it is easy to construct non-GCH examples that violate the result. For example, consider a model where $2^{\aleph_1} = 2^{\aleph_2} = \aleph_3$, which is consistent with ZFC by Easton's Theorem, and let $\kappa = \aleph_2$. Consider a partition $\mathcal{P} = \{B, B'\}$, where $|B| = \aleph_2$, and $|B'| = \aleph_1 < \mathrm{cf}(\aleph_2)$. Then $|\kappa \setminus B| = |B'| = \aleph_1$, so 260(4) yields $|\mathrm{compl}(\mathcal{P})| = \aleph_2^{\aleph_1} \geq 2^{\aleph_1}$. But in this model, $2^{\aleph_1} = 2^{\aleph_2} = 2^\kappa > \kappa$, despite $|\kappa \setminus B| < \mathrm{cf}(\kappa)$.

However, the result does hold for some classes of cardinals, regardless of whether or not GCH is assumed. In particular, whenever $\mu^\lambda < \kappa$ for all $\mu < \kappa$ and all $\lambda < \mathrm{cf}(\kappa)$. This is the case, for example, when κ is a strong limit cardinal.

9.4 Gathering all results under GCH

Collecting the results of this section gives us the following concise characterization of chains, antichains, and complements in infinite partition lattices under GCH:

Theorem 271. *Under GCH, when κ is an infinite cardinal:*

1. *Any maximal well-founded chain in Π_κ always has cardinality κ.*
2. *Any general maximal chain in Π_κ has cardinality*
   (a) *κ^-, κ, or κ^+ (and all three are always achieved) if κ is a successor cardinal; and*
   (b) *either κ or κ^+ (and both are achieved) if κ is a limit cardinal.*
3. *Any non-trivial maximal antichain in Π_κ has cardinality either κ or κ^+, and both are achieved.*
4. *Any non-trivial partition has either κ or κ^+ complements. $\mathcal{P} \notin \{\bot, \top\}$ has κ complements if and only if (i) $\mathcal{P}$ contains exactly one block, B, of cardinality κ, and (ii) $|\kappa \setminus B| < \mathrm{cf}(\kappa)$; otherwise, $\mathcal{P}$ has κ^+ complements.*

10 Acknowledgments

The authors thank Benoît Kloeckner for his many comments on a preliminary version of this article, and thank the anonymous referees for valuable suggestions. James Avery was supported by VILLUM FONDEN through the network for Experimental Mathematics in Number Theory, Operator Algebras, and Topology. Jean-Yves Moyen was partially supported by the ANR project "Elica" ANR-14-CE25-0005 and by the Marie Skłodowska–Curie action "Walgo", program H2020-MSCA-IF-2014, number 655222. Pavel Ruzicka was partially supported by the Grant Agency of the Czech Republic under the grant no. GACR 14-15479S. Jakob Grue Simonsen was partially supported by the Danish Council for Independent Research *Sapere Aude* grant "Complexity through Logic and Algebra" (COLA).

More intensional versions of Rice's Theorem

Jean-Yves Moyen, Jakob Grue Simonsen

Developments in Implicit Computational Complexity, 2016

Abstract:

Classic results in computability theory are almost invariably extensional: they concern the behaviour of partial recursive functions rather than the programs computing them. We provide generalised versions of two classic results Rice's Theorem and the Rice-Shapiro Theorem and demonstrate how they may be applied to study intensional properties such as time and space complexity. In particular we obtain simple proofs of several striking negative results about overapproximations of intensional properties, for example that any decidable property of programs that (strictly) contains the set of polytime program must contain programs of arbitrarily high time complexity.

1 Introduction

A cornerstone of computability theory is Rice's Theorem [Ric53]: any non-trivial extensional set of programs is undecidable (extensionality roughly means that the set depends only on the partial functions computed by the program). This very generic formulation allows to prove undecidability of a variety of sets *e.g.* "programs that, on input 0, return 42", "programs that compute a bijection" or "programs that compute a non-total function".

Rice's Theorem showcases a fundamental dichotomy between programs and the partial functions they compute: it gives an undecidability criterion for sets of programs, but these sets are defined by the function computed by the programs. Underlying this dichotomy and the Theorem is the notion of *extensional equivalence* "two programs are equivalent iff they compute the same function". Rice's Theorem basically tells us that this equivalence is undecidable.

Even after 60 years, scant research has been made in *intensional* analogues of Rice's Theorem, i.e. undecidability results concerning *how* programs compute rather than *what* they compute. One exception is Asperti's work on *complexity cliques* [Asp08]. Roughly, Asperti refines the extensional equivalence relation into the equivalence relation "two programs are equivalent if they compute the same function with comparable (up to big-Θ) complexity" and then states the corresponding result that this equivalence relation is also undecidable.

In this work, we generalise Rice's Theorem in a different direction. Rather than trying to refine the extensional equivalence relation and find other, more precise, ones that are still undecidable, we will first lift the strict extensionality of sets and then generalise to any kind of equivalence between programs. The first generalisation is to remove the strict extensionality of the sets considered. We will only ask that the studied sets accept all the programs computing one given function, but we put no condition on programs computing other functions (they may or not be in the set). This allows to consider more intensional sets of programs. The second generalisation is to completely change the equivalence relations considerd. We must still impose very general conditions on such relations (via so-called *switching families*). This allows both to refine the equivalence (*i.e.*, Asperti's Result fits perfectly in the generalised setting) and to consider completely different equivalences.

Our work has implications for Implicit Computational Complexity (ICC). Classical ICC provides sound but incomplete criteria for a given property (such as computing a PTIME function). The criteria are *sound* in the sense that being accepted certifies the complexity bound, but *incomplete* in the sense that some "good" programs are nonetheless rejected (but extensional completeness is usually required, e.g. every PTIME function is computed by some program accepted, but not all programs running in polynomial time are accepted). That is, a classical ICC criterion admit false negatives but guarantees the absence of false positives. In this sense, an ICC criterion can be seen as a certificate of good behaviour by the program. It is an *under*-approximation of the target set.

Consider the *non*-classical view: what if we try to characterise *over*-approximations rather than under-approximations? From an ICC point of view, that means a criterion that accepts false positives but no false negatives. Every program in the target set should be recognised but we will also accept some "bad eggs" in the process. The hope being that there will be few of them and that they won't be too bad. Typically, it would make sense to accept all the polytime programs plus "a few" exponential time ones. As long as there's not too many exponential ones and they're "only" exponential and not worse, this is still interesting. Typically, most accepted programs would run fast (e.g., in a couple of minutes) and some would be slower but not too slow.

Moreover, having over-approximation can be a good way to prove lower bounds on complexity. Classical complexity has been very good to provide upper bounds on complexity (e.g., SAT is in NP), but is bad at providing lower bounds (thus making separation results between complexity classes hard to prove). Classical ICC, via *under*-approximations, can only provide upper bounds: if the program is accepted, then it is guaranteed to have complexity at most the one of the criterion, but if the program is rejected, nothing is known. Having an *over*-approximation could help provide lower bounds: if the program is rejected, then it is guaranteed to have a "bad" behaviour.

However, our intensional versions of Rice's Theorem will show that over-approximations are patently not a viable path: There will, necessarily, always be *many* bad eggs and there will always be some *extremely* bad ones.

2 Notations and Rice's Theorem

We assume an unspecified, Turing-complete programming language, in the proofs we'll use an informal syntax for programs. We note $[\![\mathtt{p}]\!]$ the function computed by a program $\mathtt{p}$ and conversely $\mathcal{P}_f$ the set of all programs computing function f. We define the extensional equivalence, or Rice's equivalence, as $\mathtt{p}\mathfrak{R}\mathtt{q} \Leftrightarrow [\![\mathtt{p}]\!] = [\![\mathtt{q}]\!]$.

We say that a set of programs $\mathtt{P}$ is

- *non-trivial* if it is neither empty, nor the set of all programs.
- *extensional* if it is the union of classes of $\mathfrak{R}$;
- *partially extensional* for a set of functions F if it contains all the programs computing a function in F: $\mathcal{P}_F \subset \mathtt{P}$ (over approximation of $\mathcal{P}_F$). It does not need to be extensional for other functions;

- *extensionally complete* for a set a functions F if it can computes all these functions: $F \subset [\![\mathtt{P}]\!]$;
- *extensionally sound* for a set a functions F if it can compute only these functions: $[\![\mathtt{P}]\!] \subset F$ (under approximation of $\mathcal{P}_F$);
- an *ICC characterisation* of a set of functions F if it is both extensionally sound and complete for F, *i.e.* it computes exactly these functions: $[\![\mathtt{P}]\!] = F$;
- *extensionally universal* if it is extensionally complete for the set of computable partial functions (each computable partial function is computed by at least one program in `P`).

Note that any extensionally universal set of programs must be infinite as there are infinitely many computable functions.

Theorem 272 (Rice[Ric53])**.** *Any (non-trivial) extensional set of programs is undecidable.*

Sketch of proof. Let `p` be in the set and suppose that the infinite loop isn't. Consider the program `q'(x) = q(0); p(x)`. It computes the same thing as `p` (and hence is in the set) iff `q`(0) terminates (and loops otherwise), an undecidable property. □

Note for later reference, that Rice's Theorem could equivalently be formulated as: "Every decidable extensional set of programs is trivial".

3 The Rice and Rice-Shapiro Theorems, intensionally

We now provide intensional versions of two classic results; the proofs use simple tools from classical recursion theory, but are different from the proofs of the classical extensional versions.

3.1 Rice, intensional version

Definition 273 (Recursively separable)**.** Two sets A and B are said to be *recursively separable* (a concept due to Smullyan [Smu58]) if there is a decidable set C such that $A \subseteq C$ and $B \bigcap C = \emptyset$. The sets A and B are said to be *recursively inseparable* if they are not recursively separable.

That is, C may overapproximate A, but will never contain elements of B. A classical example of recursively inseparable sets (see [Pap94], Section 3.3) is:

Lemma 274. *Let* $A = \{\,\mathtt{p} \mid [\![\mathtt{p}]\!]\,(0) = 0\,\}$ *and* $B = \{\,\mathtt{p} \mid [\![\mathtt{p}]\!]\,(0) \notin \{0, \perp\}\,\}$*. They are recursively inseparable.*

Our first generalisation of Rice's Theorem is to replace the "extensional" condition by partial extensionality.

Theorem 275. *Any non-empty decidable partially extensional set of programs is extensionally universal.*

That is, any decidable over-approximation of an extensional set of programs contains at least one program for any computable partial function. Among other, it must contain at least one program that always loops. Thus, for example, a decidable set of programs capturing all the programs that compute the constant function $x \mapsto 0$ must also capture (at least) one program that always loops. There is no hope of reaching intensional completeness and keeping extensional soundness.

Note that, contrary to Rice's Theorem, we only require the set of program to be non-empty rather than non-trivial. Indeed, the trivial set of all programs is certainly decidable, partially extensional and extensionally universal.

Sketch of proof. Suppose that the set is partially extensional for $[\![\mathtt{p}]\!]$ but contains no program computing $[\![\mathtt{q}]\!]$. Consider program `r'(x) = if r(0)=0 then p(x) else q(x)`. It is in the set if `r(0) = 0` and out if `r(0)` terminates on another value, thus deciding the set would recursively separate A and B above. □

Corollary 276 (Rice's Theorem)**.** *Any (non-trivial) extensional set of programs is undecidable.*

Sketch of proof. An extensional and extensionally universal set must be the set of all programs. □

This Theorem has as immediate corollaries several folklore results that are usually explained by simple ad-hoc arguments (e.g. "adding arbitrarily many dummy `x:=x` instructions")–the theorem provides a uniform means of proving these. It is also more general than Rice's Theorem.

Corollary 277 (Many programs)**.** *For any computable function f, there are infinitely many programs computing it.*

Proof. If $\mathcal{P}_f$ is finite, it is decidable. By construction, it is partially extensional for f, hence by the Theorem it must be extensionally universal and thus infinite, an absurdity. □

Note that this is a bit less trivial than what it appears: The corollary states that it is not possible to design a programming language with only finitely many possible implementations of a given function without losing Turing completeness.

Corollary 278 (Arbitrary size)**.** *For any N, every computable function is computable by a program of size larger than N.*

Proof. Let $\mathtt{P} = \{\, \mathtt{p} \mid |\mathtt{p}| \leq N \,\}$. It is finite, hence by previous Corollary cannot contain all the programs computing a given function. □

3.2 Rice-Shapiro, intensional version

We can apply the same idea to the Rice-Shapiro Theorem Theorem [MS55, Sha56]:

Theorem 279 (Rice-Shapiro)**.** *Let F be a recursively enumerable set of computable functions. $f \in F \Leftrightarrow$ there exists $g \in F$ with finite domain D such that $\forall d \in D, f(d) = g(d)$.*

The "power" of this Theorem comes from the following construction: Suppose that f is in F. As $f \in F$, there exists g with finite domain D in F that agrees on f on this finite domain. Now, consider **any** function f' that also agrees on g on this finite domain. Using the $\Leftarrow$ direction of the Theorem, $f' \in F$. Thus, any function that agrees with f on a finite domain D must be in F and the only thing one can do to compare functions (especially to determine whether a given program computes a given function or not) is to check a finite number of inputs.

The intensional version is:

Theorem 280. *Let $\mathtt{P}$ be a recursively enumerable partially extensional set of programs. Then, for any computable function f, there exists a program $\mathtt{p} \in \mathtt{P}$ such that $[\![\mathtt{p}]\!]$ differs from f only on finitely many inputs.*

Note that $\mathtt{P}$ here is recursively enumerable and not necessarily decidable as in Rice's Theorem. For example, if $\mathtt{P}$ is a recursively enumerable set of programs partially extensional for a total function (e.g., the constant function $x \mapsto 0$), then by choosing f to be the totally undefined function, we can conclude that $\mathtt{P}$ must contain a program that loops on all but finitely many inputs, by choosing $f : x \mapsto 1$ we can conclude that $\mathtt{P}$ must also contain a program that return 1 on all but finitely many inputs.

4 Beyond Rice's Theorem

The second, and broader, generalisation we make is to choose other equivalence relations than the Rice relation $\mathfrak{R}$.

4.1 Switching families

Definition 281. Let S be a set and $\approx$ an equivalence relation on S. A *switching family compatible with* $\approx$ is a family $I = (\pi_s)_{s \in S}$ of computable total functions $\pi_s : S \times S \to S$ such that the sets $A_I = \{\, s \in S \mid \forall x, y. \pi_s(x, y) \approx x \,\}$ and $B_I = \{\, s \in S \mid \forall x, y. \pi_s(x, y) \approx y \,\}$ are recursively inseparable.

The definition implies that neither A_I nor B_I is decidable. Also, there may be elements in S neither in A_I nor in B_I such that $\pi_s(x, y)$ is not always equivalent to x and not always equivalent to y. Note that the classical projections $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$ are switching functions but in general switching functions do not need to ignore one of their arguments.

The constructions of $\mathtt{r}'$ in the proof of the Theorem 275 constitutes a switching family with $\mathtt{r}'(x) = \pi_{\mathtt{r}}(\mathtt{p}, \mathtt{q})(x) =$ `if r(0) then p(x) else q(x)`. We call this the *standard switching family.*

4.2 Rice's Theorem, second generalisation

Let $\mathfrak{P} = \{\mathtt{P}_1, \mathtt{P}_2, \dots\}$ be a partition of a set S and $\mathtt{P}$ be a subset of S, it is

- *compatible with* $\mathfrak{P}$ if it is the union of classes of $\mathfrak{P}$;
- *partially compatible* for a block $\mathtt{P}_k$ if it contains all the elements in that block: $\mathtt{P}_k \subset \mathtt{P}$. Note that it does not need to be compatible for other blocks;
- *complete* for a set of blocks $\mathtt{P}_{k_1}, \mathtt{P}_{k_2}, \dots$ if it contains at least one element of each of these blocks: $\mathtt{P} \bigcap \mathtt{P}_{k_i} \neq \emptyset$;

- *universal* if it is complete for $\mathfrak{P}$ itself: it contains at least one element in each block of the partition.

Theorem 282. *Let $\mathfrak{P}$ be a partition of a set S and $I = (\pi_s)_{s\in S}$ be a switching family compatible with it.*
Any non-empty decidable partially compatible subset of S is universal.

Sketch of proof. Let x be in a class included in the set and y be in a class that does not intersect it. Consider $s' = \pi_s(x, y)$. Deciding if s' is in the set would recursively separate A_I and B_I, which is impossible by definition of the switching family. □

4.3 Examples : computational complexity

We now come to our most striking example: any decidable set of of programs containing, for instance, the polynomial-time programs, must contain programs of arbitrarily high complexity.

Example 283 (Complexity)**.** Let Φ be a complexity measure in the sense of Blum. Let $\equiv_\Phi$ be the equivalence relation $\mathtt{p} \equiv_\Phi \mathtt{q}$ iff $\Phi_{\mathtt{p}} \in \Theta(\Phi_{\mathtt{q}})$.

The standard switching family is compatible with the above relation: when `r(0)` terminates it does so with a constant complexity, whence the global complexity is the one of `p` (resp. `q`) up to a constant additive factor.

Hence, any non-empty decidable set of programs partially compatible with $\equiv_\Phi$ is universal and must contain programs of arbitrarily high complexity.

This example can be specialised by making the partially compatible set more precise. Note that we consider sets of programs defined by properties of the programs themselves, and *not* necessarily by properties of the computed function (*i.e.* this is not the set of programs computing PTIME functions as it does not include inefficient programs that compute a "simple" function in a long time).

Example 284 (Polynomial time)**.** Let PPTIME be the set of *programs* whose runtime is polynomial. It is partially compatible with $\equiv_\Phi$ when Φ is the usual time complexity. Thus, any decidable set that includes all the *programs* computing in polynomial time must also include programs of arbitrarily high time complexity. Including not only exponential programs but also non primitive recursive ones and even non multiple recursive ones.

Any attempt at finding a decidable over-approximation of PPTIME is doomed to also contain many extremely "bad" programs.

Example 285 (Linear space)**.** Let PLINSPACE be the set of programs computing in linear space. It is partially compatible with $\equiv_\Phi$ when Φ is the usual space complexity. Thus, any decidable set that contains PLINSPACE must also contains programs of arbitrarily high space complexity.

Thus, even for sets not closed under composition, over-approximations cannot both be decidable and contain only programs of restricted space complexity.

Example 286 (Asperti-Rice, [Asp08])**.** Let $\mathfrak{A}$ be the equivalence relation $\mathtt{p}\mathfrak{A}\mathtt{q}$ iff $\big([\![\mathtt{p}]\!] = [\![\mathtt{q}]\!]$ and $\Phi_{\mathtt{p}} \in \Theta(\Phi_{\mathtt{q}})\big)$. The standard switching family is compatible with it (seeing relations as subset of couples, we have $\mathfrak{A} = (\mathfrak{R} \bigcap \equiv_\Phi)$, the standard switching family being compatible with both).

Hence, any non-empty decidable set of programs partially compatible with $\mathfrak{A}$ is universal. This generalises Asperti's result that the only decidable compatibles sets (*i.e.* "Complexity cliques" in his formalism) are the trivial ones.

More intensional versions of Rice's Theorem

Jean-Yves Moyen, Jakob Grue Simonsen
Long version of previous article

1 Introduction

A cornerstone of computability theory is Rice's Theorem [Ric53]: any non-trivial extensional set of programs is undecidable (extensionality means that the set depends only on the partial functions computed by the program). This very generic formulation allows to prove undecidability of a variety of sets *e.g.* "programs that, on input 0, return 42", "programs that compute a bijection" or "programs that compute a non-total function".

Rice's Theorem showcases a fundamental dichotomy between programs and the partial functions they compute: it gives an undecidability criterion for sets of programs, but these sets are defined by the function computed by the programs. Underlying this dichotomy and the Theorem is the notion of *extensional equivalence* "two programs are equivalent iff they compute the same function". Rice's Theorem basically tells us that this equivalence is undecidable.

Even after 60 years, scant research has been made in *intensional* analogues of Rice's Theorem, i.e. undecidability results concerning *how* programs compute rather than *what* they compute. One exception is Asperti's work on *complexity cliques* [Asp08]. Roughly, Asperti refines the extensional equivalence relation into the equivalence relation "two programs are equivalent if they compute the same function with comparable (up to big-Θ) complexity" and then states the corresponding result that this equivalence relation is also undecidable.

In this work, we generalise Rice's Theorem in a different direction. Rather than trying to refine the extensional equivalence relation and find other, more precise, ones that are still undecidable, we will first lift the strict extensionality of sets and then generalise to any kind of equivalence between programs. The first generalisation is to remove the strict extensionality of the sets considered. We will only ask that the studied sets accept all the programs computing one given function, but we put no condition on programs computing other functions (they may or not be in the set). This allows to consider more intensional sets of programs. The second generalisation is to completely change the equivalence relations considered. We must still impose very general conditions on such relations (via so-called *switching families*). This allows both to refine the equivalence (*i.e.*, Asperti's Result fits perfectly in the generalised setting) and to consider completely different equivalences.

Our work has implications for Implicit Computational Complexity (ICC). Classical ICC provides sound but incomplete criteria for a given property (such as computing a PTIME function). The criteria are *sound* in the sense that being accepted certifies the complexity bound, but *incomplete* in the sense that some "good" programs are nonetheless rejected (but extensional completeness is usually required, e.g. every PTIME function is computed by some program accepted, but not all programs running in polynomial time are accepted). That is, a classical ICC criterion admit false negatives but guarantees the absence of false positives. In this sense, an ICC criterion can be seen as a certificate of good behaviour by the program. It is an *under*-approximation of the target set.

Consider the *non*-classical view: what if we try to characterise *over*-approximations rather than under-approximations? From an ICC point of view, that means a criterion that accepts false positives but no false negatives. Every program in the target set should be recognised but we will also accept some "bad eggs" in the process. The hope being that there will be few of them and that they won't be too bad. Typically, it would make sense to accept all the polytime programs plus "a few" exponential time ones. As long as there's not too many exponential ones and they're "only" exponential and not worse, this is still interesting. Typically, most accepted programs would run fast (*e.g.*, in a couple of minutes) and some would be slower but not too slow (*e.g.*, run in a couple of hours or days). In non time-critical situations, such guarantee can be useful (in the same way that quicksort outperforms merge sort in average even though it has quadratic worse case; or that the simplex algorithm is remarkably efficient in practise despite its exponential worse case).

Moreover, having over-approximations can be a good way to prove lower bounds on complexity. Classical complexity has been very good to provide upper bounds on complexity (*e.g.*, SAT is in NP), but is bad at providing lower bounds (thus making separation results between complexity classes hard to prove). Classical ICC, via *under*-approximations, can only provide upper bounds: if the program is accepted, then it is guaranteed to have complexity at most the one of the criterion, but if the program is rejected, nothing is known. Having an *over*-approximation could help provide lower bounds: if the program is rejected, then it is guaranteed to have a "bad" behaviour.

Figure 1 describes the situation. We have an extensional set (the set of all programs computing a function in PTIME, irregular red shape). We know, by Rice's Theorem, that such an extensional set is undecidable. ICC aimed at finding an under-approximation of it (small blue circle) and to make it as large as possible (unfortunately, we know by experience that ICC under-approximations tend to be very small).

ICC also requires the under-approximation to be *extensionally complete*: it must contain one program for each function in PTIME. Thus we know, for example, that there is at least one program computing the sorting function in the under-approximation. However, we also know (by experience) that there are many programs computing the sorting function out of it (but still in the set of programs computing PTIME functions, obviously), these are false negatives. Note that this is not necessarily bad as many programs computing the sorting function are utterly inefficient (*e.g.* first compute Ackermann function on each value in the list to be sorted, then sort it) and it is rather good that we reject them...

Under-approximations, like classical complexity theory, are very good at providing upper-bounds. Any program in the under-approximation is proved to be in the target set and thus computes a PTIME function. This gives an

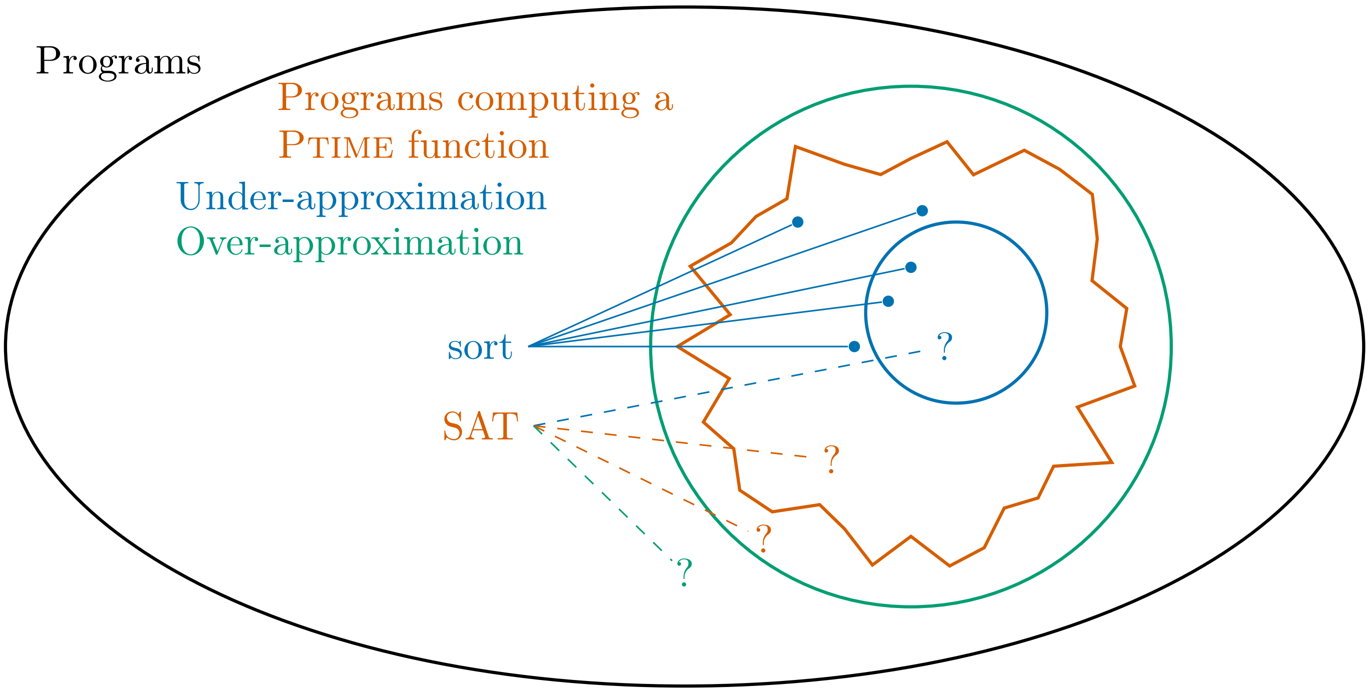


Figure 1: Under- and Over-approximations.

upper bound on the complexity of the function. If we could find a single program resolving, say, SAT, in the under-approximation (blue question mark) then we'd know for sure that this program computes a PTIME function, hence SAT would be PTIME.

Now, we want to investigate over-approximations, such as the large green circle. Finding programs that are in the over-approximation but not in the under-approximation (red question marks) does not improve our knowledge as we have no way to know whether they are in the target set or not (it may be a false positive). Obviously, making the over-approximation small and the under-approximation large will reduce the number of such cases and give a better understanding of the target set. Moreover, over-approximations are good to prove *lower bounds*, something at which classical complexity fails. Indeed, if we can find a single program computing SAT that is **not** in the over-approximation (green question mark), then we know for sure that this program is not in the target set and thus it **does not** compute a PTIME function. This would allow to prove that SAT is not in PTIME.

However, our intensional versions of Rice's Theorem will show that over-approximations are patently not a viable path: There will, necessarily, always be *many* bad eggs (false positives) and there will always be some *extremely* bad ones.

2 Preminaries and notations

2.1 Notations

We assume an unspecified, Turing-complete, programming language and note $\mathcal{P}$ the set of all programs. In the proofs we'll use an informal syntax for programs. We note $[\![\mathtt{p}]\!]$ the function computed by a program $\mathtt{p}$ (*i.e.*, semantics brackets) and conversely $\mathcal{P}_f$ the set of all programs computing function f. We write $[\![\mathtt{p}]\!](x) = \perp$ to denote that $\mathtt{p}$ does not terminate on input x.

We assume given a computable injective encoding of programs into natural numbers and we'll abusively write $\mathtt{p}$ do also design its encoding. Alternatively, this may be seen as programs accepting (source code of) programs as input. We also assume given an universal program, that is, given the source code of a program (or its encoding into a number), it is possible to simulate it on any input. Moreover, even if it is not always explicitly mentioned in the proofs, an *s-m-n* Theorem (or an equivalent one) is often used. All these are common ingredients of computability and hence part of any Turing-complete language.

We say that a set of programs $\mathtt{P}$ is

- *non-trivial* if it is neither empty, nor the set of all programs.
- *extensional* if it does not separate programs computing the same function: if $\mathtt{p} \in \mathtt{P}$ and $\mathtt{q} \notin \mathtt{P}$ then $[\![\mathtt{p}]\!] \neq [\![\mathtt{q}]\!]$;

- *partially extensional* for a set of functions F if it contains all the programs computing a function in F: $\mathcal{P}_F \subset \mathtt{P}$ (over approximation of $\mathcal{P}_F$). It does not need to be extensional and thus functions out of F may be computed by both programs in and out of $\mathtt{P}$;
- *extensionally complete* for a set a functions F if it can computes all these functions: $F \subset [\![\mathtt{P}]\!]$. It does not need to contain *all* the programs computing a function in F as long as it contains at least one program for each function in F;
- *extensionally sound* for a set a functions F if it can compute only these functions: $[\![\mathtt{P}]\!] \subset F$ (and thus $\mathtt{P} \subset \mathcal{P}_F$ is an under approximation of $\mathcal{P}_F$);
- an *ICC characterisation* of a set of functions F if it is both extensionally sound and complete for F, *i.e.* it computes exactly these functions: $[\![\mathtt{P}]\!] = F$;
- *extensionally universal* if it is extensionally complete for the set of computable partial functions (each computable partial function is computed by at least one program in $\mathtt{P}$).

Note that any extensionally universal set of programs must be infinite as there are infinitely many computable functions.

3 Rice's Theorem: background and previous work

3.1 Rice's Theorem and the extensional equivalence

Theorem 287 (Rice [Ric53])**.** *Any (non-trivial) extensional set of programs is undecidable.*

Proof. Let $\mathtt{P}$ be a non-trivial extensional set of programs and assume that it does not contains programs computing the never defined function (otherwise, work with its complement). Let $\mathtt{p} \in \mathtt{P}$ be a program in it.

Let $\mathtt{q}$ be any program and build program $\mathtt{q}'$ as follows: on input x, $\mathtt{q}'$ first simulates $\mathtt{q}$ on input 0 and then $\mathtt{p}$ on input x. Informally speaking, we have $\mathtt{q}'(x) = \mathtt{q}(0); \mathtt{p}(x)$.

If $\mathtt{q}$ terminates on input 0, then $\mathtt{q}'$ always computes the same thing as $\mathtt{p}$ and, by extensionality, must be in $\mathtt{P}$. On the other hand, if $\mathtt{q}$ does not terminate on input 0, then $\mathtt{q}'$ never terminates and thus, by hypothesis, is not in $\mathtt{P}$.

Thus, $\mathtt{q}' \in \mathtt{P}$ if and only if $\mathtt{q}$ terminates on input 0, which is not decidable. ☐

Note that the precise construction of $\mathtt{q}'$ heavily depends on the chosen language and its programming "bricks". However, be it function calls, universal machines or *s-m-n* Theorem, we know that given $\mathtt{p}$ and $\mathtt{q}$ (their Gödel's numbers, source codes, pre-compiled libraries, λ-term, ...) it is possible to build $\mathtt{q}'$ acting as required. Note also that even if $\mathtt{q}(0)$ is actually a constant, it can obviously not be pre-computed and thus $\mathtt{q}'$ **must** contain the call to $\mathtt{q}(0)$ in its code in order to get the proof working.

The power of Rice's Theorem is that it allows to easily prove undecidability of a variety of properties of programs, for instace:

- properties about termination on a given input, *e.g.* "programs which terminate on input 17";
- properties about precise results on a given input, *e.g.* "programs which return 42 on input 54";
- properties about larger input-output subsets, *e.g.* "programs that return an even integer on any prime input";
- more general properties on the input-output relation, *e.g.* "programs computing a monotonic function", "programs computing a bijection";

However, Rice's Theorem is limited to extensional properties. It shows undecidability of any (non-trivial) set of program behaviours that depend only on the input-output relation but it is unable to tell anything about *intensional* sets of programs, that depend on the actual program behaviour. That is, it provides undecidability results on "what is computed?" question but is unable to provide undecidability results on "how is it computed?".

But intensional behaviour of programs is important in practice: the time or space resource use of a program is an intensional property, as are security issues such as "will the program overwrite `/bin/init`?". When armed only with Rice's Theorem we still need to prove undecidability of these questions "by hand".

Now, the extensionality central to Rice's Theorem defines an equivalence relation between programs.

Definition 288 (The Rice Equivalence Relation)**.** We define the extensional equivalence, or *Rice equivalence*, as $\mathtt{p}\mathfrak{R}\mathtt{q} \Leftrightarrow [\![\mathtt{p}]\!] = [\![\mathtt{q}]\!]$, that is "two programs are equivalent iff they compute the same function".

With this in mind, an extensional set of programs is exactly the union of equivalence classes of the equivalence relation $\mathfrak{R}$. Moreover, computable functions now become just convenient names for the classes of this equivalence and programs can be seen as representatives of it. That is, f is not only a function but also the class of equivalence of all programs computing it and the notation $[\![\mathtt{p}]\!] = f$ means that the class of equivalence of $\mathtt{p}$ is exactly f. This notation is reminiscent to usual bracketed notations for classes of equivalences (where $[x]$ is the class of equivalence of x).

Rice's Theorem states that not only is $\mathfrak{R}$ itself undecidable but that any (non-trivial) union of classes of it is also undecidable. Equivalence relations (or, similarly, partitions) have a very nice order structure, the refinement ordering[1], and Rice's Theorem says that any (non-trivial) equivalence less precise than $\mathfrak{R}$ is undecidable. Part of the intensional quest is to ask what happen for equivalences which are more precise than $\mathfrak{R}$.

Theorem 289 (Rice's Theorem, equivalence relation version)**.** *Any (non-trivial) equivalence relation $\equiv$ such that $\mathfrak{R} \leq \equiv$ is undecidable.*

We shall momentarily generalise Rice's Theorem by utilizing the view of Rice's Theorem as a result on equivalence classes as follows. First, we shall require the target set to simply *contain* some classes of $\mathfrak{R}$, and not necessarily to be exactly the union of these (partial extensionality); next, we shall consider what happens if we completely change the equivalence relation and use some other equivalence than $\mathfrak{R}$.

3.2 The Asperti-Rice Theorem

Despite its venerable age, little work have been done to try and improve Rice's Theorem. A notable exception lies in the work of Asperti [Asp08]. We present here the result purely in term of equivalence relation.

Definition 290 (Asperti's equivalence)**.** Let Φ be a complexity measure in the sense of Blum [Blu67]. We define Asperti's equivalence as $\mathtt{p}\mathfrak{A}\mathtt{q} \Leftrightarrow [\![\mathtt{p}]\!] = [\![\mathtt{q}]\!] \wedge \Phi(\mathtt{p}) \in \Theta(\Phi(\mathtt{q}))$, that is "two programs are equivalent iff they compute the same function with the same complexity."

This equivalence is exactly Asperti's "similarity" (Def. 3 and Thm 4.). Next, Asperti introduce the notion of *complexity cliques* (Def. 5) which are exactly the unions of classes of $\mathfrak{A}$ (Thm. 8). Because we have already started to switch the vocabulary toward equivalences and classes, we don't need to define these cliques.

Theorem 291 (Asperti-Rice [Asp08])**.**
Any (non-trivial) union of classes of $\mathfrak{A}$ is undecidable.
or, said otherwise,
Any (non-trivial) equivalence less precise than $\mathfrak{A}$ is undecidable.

Strikingly, the proof is essentially the same as for Rice's Theorem. Especially, the reduction from the Halting problem is exactly the same and we just need to add a discussion on the complexity (which is here somewhat simpler and less formal than Asperti's original proof). We emphasise this similitude by greying out parts of the proof that are the same as for Rice's Theorem.

Proof. First, note that programs computing the never defined function must also have a never defined complexity. Hence, they form exactly one class of $\mathfrak{A}$.

Let $\mathtt{P}$ be a non-trivial union of classes of $\mathfrak{A}$ and assume that it does not contains programs computing the never defined function (otherwise, work with its complement). Let $\mathtt{p} \in \mathtt{P}$ be a program in it.

Let $\mathtt{q}$ be any program and build program $\mathtt{q}'$ as follows: on input x, $\mathtt{q}'$ first simulates $\mathtt{q}$ on input 0 and then $\mathtt{p}$ on input x. Informally speaking, we have $\mathtt{q}'(x) = \mathtt{q}(0); \mathtt{p}(x)$.

If $\mathtt{q}$ terminates on input 0, then $\mathtt{q}'$ always computes the same thing as $\mathtt{p}$; moreover, the complexity of $\mathtt{q}'$ is the (constant) complexity of $\mathtt{q}(0)$ (constant because it is computed on a fixed input) plus the complexity of $\mathtt{p}$[2]. Because the constant additive factor is absorbed in the Θ equivalence, we have $\Phi(\mathtt{q}') \in \Theta(\Phi(\mathtt{p}))$ and because they compute the same function we thus have $\mathtt{p}\mathfrak{A}\mathtt{q}'$, hence $\mathtt{q}' \in \mathtt{P}$.

On the other hand, if $\mathtt{q}$ does not terminate on input 0, then $\mathtt{q}'$ never terminates and thus, by hypothesis, is not in $\mathtt{P}$.

Thus, $\mathtt{q}' \in \mathtt{P}$ if and only if $\mathtt{q}$ terminates on input 0, which is not decidable. $\square$

Among other, Asperti's result is enough to prove the undecidability of complexity classes. Indeed, for example, the set of all programs computing in quadratic time ($\Theta(n^2)$) is clearly a non-trivial union of classes of $\mathfrak{A}$, hence undecidable. Similarly, the sets of polytime programs, of linear space programs, ... are undecidable.

As said, this proof boils down to the exact same argument as the proof of Rice's Theorem. In this sense, the title of Asperti's article "*The intensional Content of Rice's Theorem*" is uncannily precise: all this result was already

[1]The refinement ordering is defined by $\equiv_0 \leq \equiv_1$ iff $x \equiv_0 y$ implies $x \equiv_1 y$, see Subsection 3.4 for more details.

[2]This is where this version of the proof lacks formality while Asperti enforces *s-m-n* and *linear-composition* properties to ensure that the complexity of the sequencing is indeed small enough.

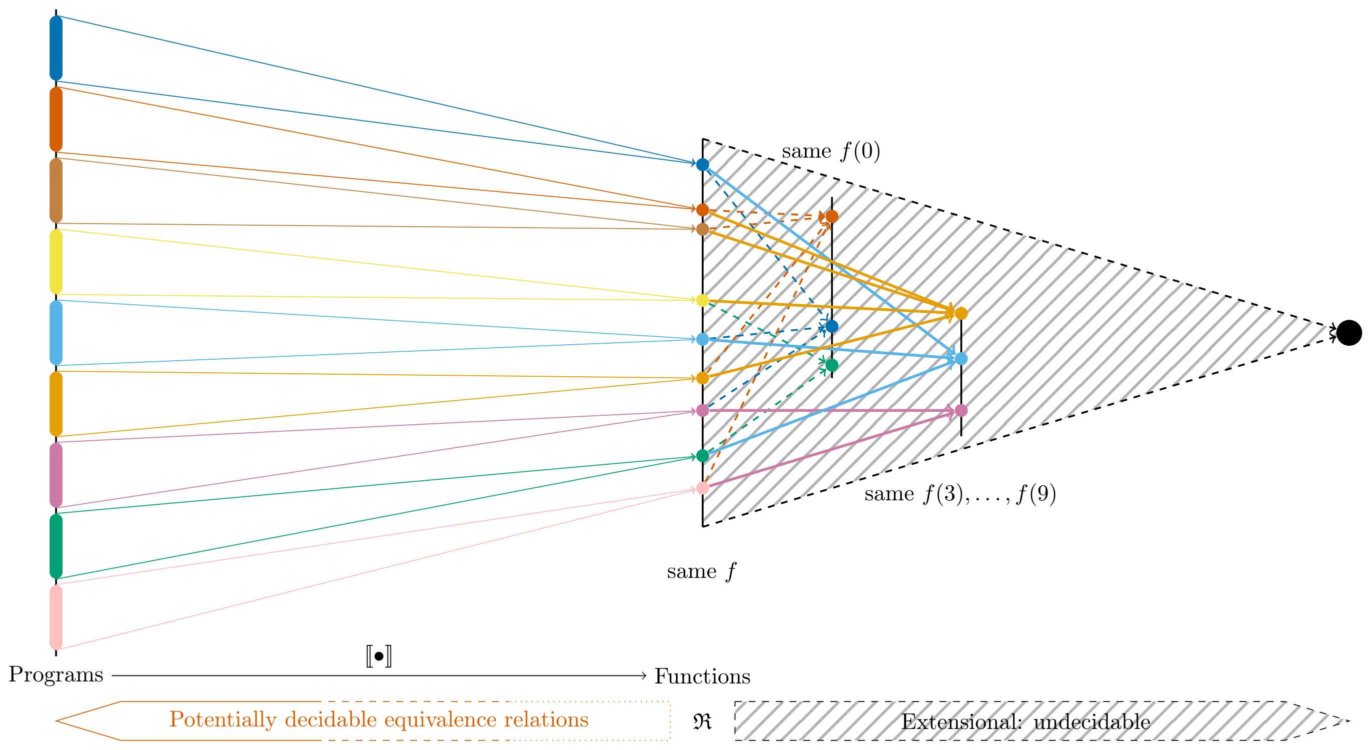


Figure 2: The extensional cone

contained in Rice's Theorem and it took us 50 years to realise it. This statement is certainly interesting from an history of science point of view in pointing the early focus on computability while complexity was of lesser interest. It also makes it a true *tour de force* to have been able to look through a fifty years old classical result and see that it actually was more powerful than globally believed.

The results in this paper will require a different proof. Thus, we are not only studying the content of an existing result but really giving new, more intensional, versions of it.

3.3 The extensional cone

The situation after Rice's Theorem can be illustrated as in Figure 2. On the left, we have the set of all programs and in the middle the set of computable functions. The semantics function, $[\![\bullet]\!]$, maps programs onto functions and several programs are mapped to the same function. It exactly defines the extensional equivalence, $\mathfrak{R}$, which is colour-coded in the picture: the green programs are all the programs that compute the green function and they form exactly one class of $\mathfrak{R}$. And by definition, the classes of $\mathfrak{R}$ (*i.e.* the computable functions) are the basic blocks with which we can build extensional sets: each extensional set is always exactly the union of some of these.

Now, Rice's Theorem states that any (non-trivial) extensional set is undecidable. Thus, each class of $\mathfrak{R}$ itself is undecidable. But the Theorem also states that no matter how we group them, the resulting union is undecidable. That is, we may make $\mathfrak{R}$ less precise by grouping all the programs that return the same value on input 0 (dashed lines), the resulting equivalence is undecidable by Rice's Theorem. Or we could group them if they return the same value on all inputs between 3 and 9 (bold lines), and still get an undecidable but completely different equivalence, and so on. . .

Actually, Rice's Theorem states that anything that lies within the hatched "extensional cone" on the right of the picture is undecidable as it is extensional.

On the other side, we know that what's on the extreme left is decidable as it is the (syntactical) equality between programs (*i.e.* the equivalence where no two different elements are actually equivalent). Thus, the first part of the intensional quest lies in this picture: if we are anywhere in the cone on the right of $\mathfrak{R}$, we are undecidable. But the left-end of the picture is decidable. So when we move to the left, something must become decidable at some point. When and how does it become decidable?

The situation with Asperti-Rice Theorem is now depicted in Figure 3. The boundary of undecidability has been pushed back to the left from $\mathfrak{R}$ to $\mathfrak{A}$. Actually, $\mathfrak{A}$ is more a family of equivalences rater than a single one because each Blum's complexity measure creates a new $\mathfrak{A}$. For the sake of clarity a single one is represented on the Figure. Asperti's equivalence is **not** extensional. Programs computing the same function are split by complexity. Hence, it really lies on the left of $\mathfrak{R}$ and gives some answers to the intensional quest. The question of when all this becomes

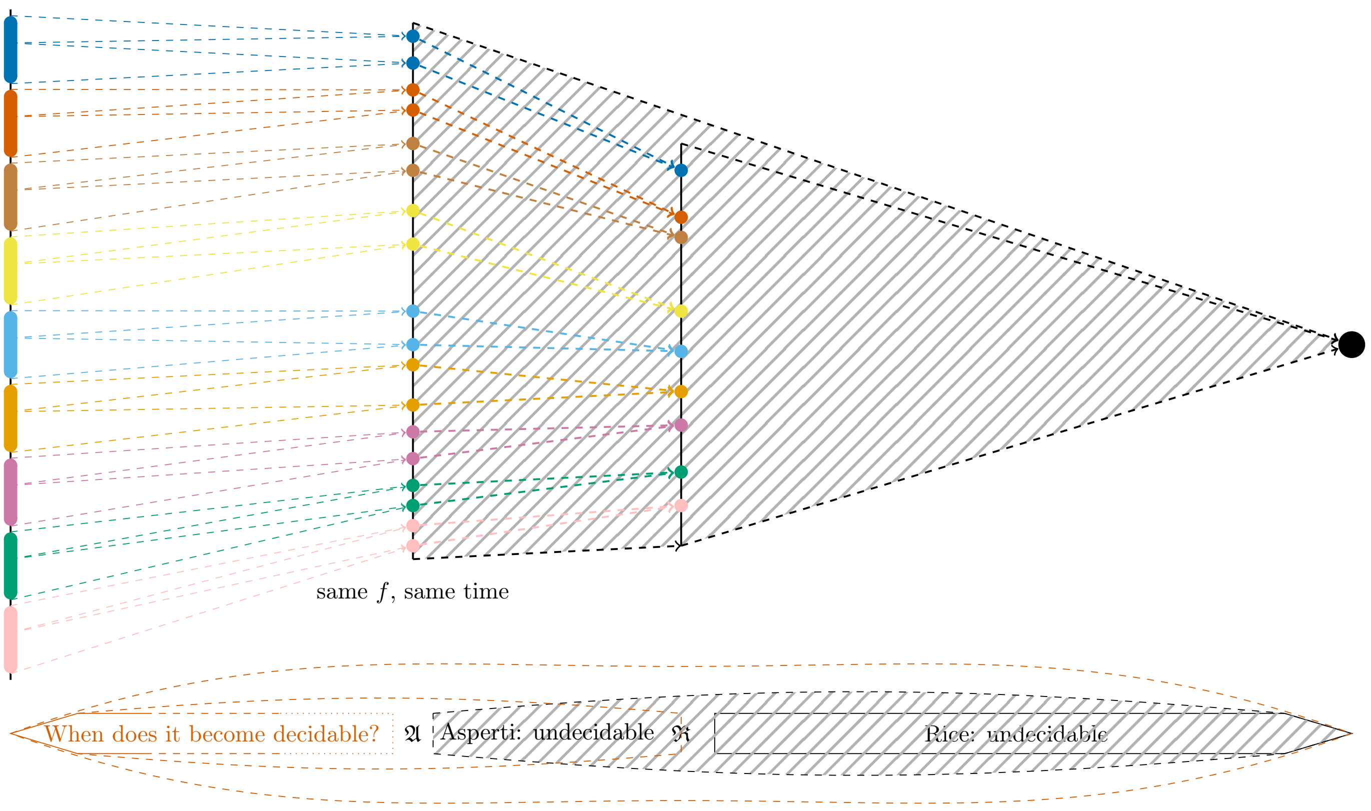


Figure 3: The extensional cone, revisited

decidable is now restricted to what's on the left of $\mathfrak{A}$.

Everything on the right of $\mathfrak{A}$ is undecidable by Asperti-Rice Theorem. This includes $\mathfrak{R}$ and the previous extensional cone and thus this is indeed a generalisation of Rice's Theorem. It is worth noticing that the new cone on the right of $\mathfrak{A}$ can, somehow, "circumvent" $\mathfrak{R}$. That is, it is possible to go from $\mathfrak{A}$ to the trivial equivalence by a chain (in the order theoretic sense) of equivalences that are all out of the extensional cone! For example, classes of $\mathfrak{A}$ can first be grouped purely by complexity, disregarding semantics.

This is the reason why, on the bottom, the hatched area of Asperti-Rice Theorem goes around the hatched arrow of Rice's Theorem. But the same thing happens on the left! It is, similarly, possible to go from syntactical equality to $\mathfrak{R}$ by circumventing $\mathfrak{A}$ for a long time (*e.g.*, group program by "same semantics, same size" first). It is also possible to go from on end to the other by a chain of equivalences that completely circumvent both $\mathfrak{A}$ and $\mathfrak{R}$, it is even possible to do so with a chain where every single equivalence is decidable (*e.g.* after numbering the programs in a decidable way, consider the equivalences with a single non-singleton class $[\![0; n]\!]$).

This brings the second part of the intensional quest. Not only do we want to push the boundary of undecidability as far to the left as possible, but we also want to study equivalences that do not lie on this semantics cone. The one-dimensional representation below our cone is poorly adapted to illustrate this.

3.4 The equivalence lattice

As mentioned previously, the set of equivalence relations on a set has a natural ordering, the *refinement* defined by $\equiv_0 \leq \equiv_1$ iff $x \equiv_0 y$ implies $x \equiv_1 y$. Said otherwise, this means that each class of $\equiv_0$ is a subset of a class of $\equiv_1$. Together with this ordering, the set of equivalences forms a *complete lattice* [Ore42] whose minimal element, $\perp$, is the equality and whose maximal element, $\top$, is the trivial equivalence with a single class (each element is equivalent to each other).

While it is not the main focus of this article, it is worth noticing that this lattice has a very rich structure [AMRS17] and interacts with decidability properties in complicated ways [MS].

To come back to Rice's and Asperti-Rice Theorems, expressed in the language of lattice, they now become:

Theorem 292 (Rice, Asperti)**.**
Every (non-trivial) equivalence in the principal filter at $\mathfrak{R}$ is undecidable.
Every (non-trivial) equivalence in the principal filter at $\mathfrak{A}$ is undecidable.

The fact that these results are so cleanly expressed with order theoretic vocabulary is in itself a strong clue that the study of the equivalences lattice can shed some light on the extensional/intensional dichotomy.

Similarly, once we are given a Blum complexity measure Φ, this defines an equivalence $\mathtt{p}\mathfrak{B}_\Phi\mathtt{q}$ iff $\Phi(\mathtt{p}) \in \Theta(\Phi(\mathtt{q}))$ (it is indeed an equivalence because Θ defines an equivalence between functions). Now, we have $\mathfrak{A} = \mathfrak{R} \wedge \mathfrak{B}_\Phi$ is *exactly* the meet of these two equivalences. Again, the simplicity of this relation in the order theoretic language tells us to dig into it. This fact was already noticed by Asperti as the proof that $\mathfrak{A}$ ("similarity") is indeed an equivalence (Thm. 4) simply points out that it is the intersection of two equivalences. Note that this implies that $\mathfrak{A} \leq \mathfrak{B}_\Phi$ hence the undecidability of $\mathfrak{B}_\Phi$ (and its principal filter) is a direct consequence of Asperti-Rice Theorem.

We shall not delve deeper into the lattice structure, but rather examine different equivalence relations. Especially, we shall consider equivalence relations not in the principal filter at either $\mathfrak{R}$ or $\mathfrak{A}$,and thus provide new results and describe new areas of undecidability in the lattice.

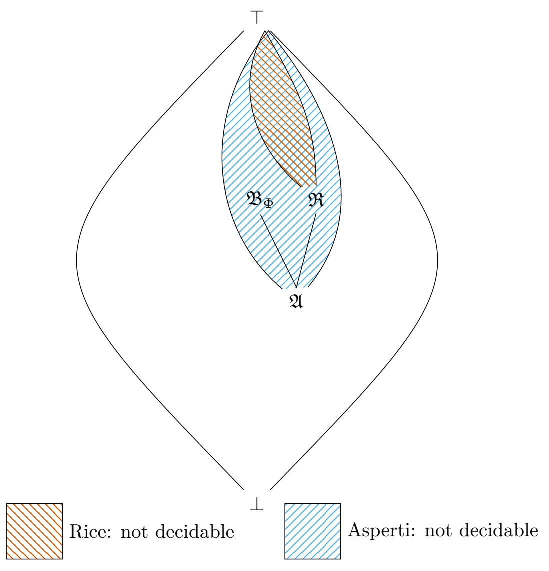


Figure 4: The equivalence lattice

4 Intensionality: first generalisation

The first generalisation we want to make is to consider target sets not as unions of classes of $\mathfrak{R}$ (extensional sets) but simply as containing some classes (partially extensional set), that is we want to speak about over-approximations of extensional sets.

Obviously, we are going to loose undecidability in the process. Indeed, a set such as "the set of programs that contain a loop" is partially extensional for the never defined function (as programs without loops always terminate), non-trivial, and decidable. However, we'll see that we can still get something interesting out of it.

If we closely examine the proof of Rice's Theorem, the argument consists in finding two functions, one in $\mathtt{P}$ (or rather all programs computing it are in $\mathtt{P}$) and one not in it, and then for each program $\mathtt{q}$ we can build a program $\mathtt{q}'$ which computes one of the two functions depending on a non-decidable property; we can then conclude by extensionality. That is, the proof actually requires only partial extensionality of $\mathtt{P}$ (for $[\![\mathtt{p}]\!]$) and of its complement (for the never defined function). Partial extensionality for other functions is not actually required in the proof and Rice's Theorem could be rephrased as "there is no non-trivial decidable set of programs that is partially extensional for a given function $[\![\mathtt{p}]\!]$ and whose complement is partially extensional for the never defined function". Hence, generalisation to partially extensional sets seem feasible.

While it is easy to still require partial extensionality for $[\![\mathtt{p}]\!]$ (this is what we want to talk about), require that $\overline{\mathtt{P}}$ is partially extensional for the infinite loop is a too strong restriction. It can be lifted and made more general by require partial extensionality for one other (unspecified) function. This in turn forces us to find an undecidable problem (on $\mathtt{q}$) which is not simply the halting problem (because we don't know who the "opponent" of $\mathtt{p}$ will be and whether it terminates or not)...

4.1 Separable sets

The notion of *recursively separable* sets was introduced by Smullyan [Smu58]. We introduce a more general version of separability. All sets are considered to be part of an unspecified "universe" (*e.g.*, sets of numbers, ...)

Definition 293 (Separable sets)**.** Let $\mathcal{S}$ be a family of sets.

Two sets A and B are said to be $\mathcal{S}$*-separable* if there exists two sets $C, D \in \mathcal{S}$ such that $A \subseteq C$ and $B \subseteq D$ and $C \bigcap D = \emptyset$.

The sets A and B are said to be $\mathcal{S}$*-inseparable* if they are not $\mathcal{S}$-separable.

That is, C and D are respectively over-approximations of A and B that have a nice property (being members of the family) but do not intersect.

If $\mathcal{S} = \text{REC}$, that is $\mathcal{S}$ is the family of all recursive (*i.e.* decidable) sets, then A and B are REC-(in)separable iff they recursively (in)separable.

For families closed under complement (which include the recursive sets or the PTIME sets), we can always choose $D = \overline{C}$ and thus we can completely get ride of D. However, the symmetrical notion is needed for families not closed under complement which, typically, include not only recursively enumerable or NPTIME sets, but also the open (or closed) sets of a topological space.

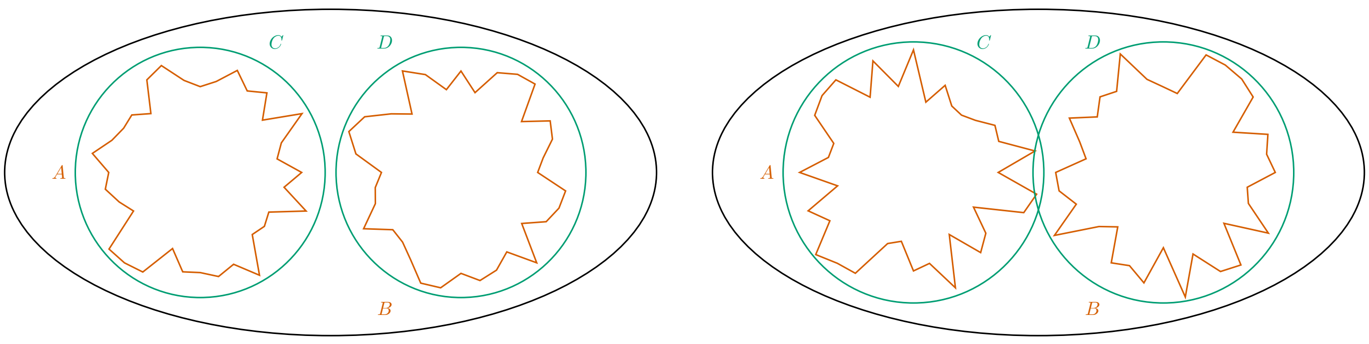


Figure 5: Left: two circle-separable sets Right: two circle-inseparable sets.

Also observe that the notion of separability is not specific to computability, complexity or computer science at large. For example, Figure 5 shows examples of circle-(in)separable sets, a purely geometrical property.

The following Lemma and Corollary are classical examples of recursively inseparable sets. See, for example, [Pap94], Section 3.3.

Lemma 294. *Let $A = \{\, \mathtt{p} \mid [\![\mathtt{p}]\!](\mathtt{p}) = 0 \,\}$ and $B = \{\, \mathtt{p} \mid [\![\mathtt{p}]\!](\mathtt{p}) \notin \{0, \perp\} \,\}$. Then, A and B are recursively inseparable.*

B is the set of programs that do terminate but with a result other than 0. Note that both A and B are RE sets, hence they are RE-separable. For recursive inseparability, because the recursive sets are closed under complement, we only look for a suitable C.

Proof. Assume that there is a decidable C that contains A and does not intersects B. Let $\mathtt{q}$ be a program that decides C and answers 1 if its input is in C and 0 otherwise.

Now, we have two cases:

- If $\mathtt{q} \in C$ then since $\mathtt{q}$ decides C, we must have $[\![\mathtt{q}]\!](\mathtt{q}) = 1$ but in this case, by definition of B, we have $\mathtt{q} \in B$ and by construction of the recursively separating C, $B \bigcap C = \emptyset$ hence $\mathtt{q} \notin C$, absurd.
- If $\mathtt{q} \notin C$ then since $\mathtt{q}$ decides C, we must have $[\![\mathtt{q}]\!](\mathtt{q}) = 0$ but in this case, by definition of A, we have $\mathtt{q} \in A$ and by construction of the recursively separating C, $A \subset C$ hence $\mathtt{q} \in C$, absurd.

Thus, no such decidable C exists and A and B are recursively inseparable. □

Corollary 295. *Let $A' = \{\, \mathtt{p} \mid [\![\mathtt{p}]\!](0) = 0 \,\}$ and $B' = \{\, \mathtt{p} \mid [\![\mathtt{p}]\!](0) \notin \{0, \perp\} \,\}$. They are recursively inseparable.*

Proof. Let $\mathtt{p}$ be a program and construct $\mathtt{p}'$ as the program which, on any input, simulate $\mathtt{p}$ on input $\mathtt{p}$, that is $\mathtt{p}'(x) = \mathtt{p}(\mathtt{p})$ for all x. Note that, obviously, the call $\mathtt{p}(\mathtt{p})$ is **not** pre-computed and has to be made each time $\mathtt{p}'$ is used, hence the construction of $\mathtt{p}'$ is indeed feasible.

Thus, we have $[\![\mathtt{p}']\!](0) = [\![\mathtt{p}]\!](\mathtt{p})$. By construction, $\mathtt{p}' \in A' \Leftrightarrow \mathtt{p} \in A$ and $\mathtt{p}' \in B' \Leftrightarrow \mathtt{p} \in B$.

Now, if A' and B' were recursively separable by C', then A and B would be recursively separable by $C = \{\, \mathtt{p} \mid \mathtt{p}' \in C' \,\}$ which would be decidable because the construction of $\mathtt{p}'$ is. □

4.2 Partially extensional sets

We can now give a first intensional generalisation of Rice's Theorem.

Theorem 296. *Any (non-empty), decidable, partially extensional set of programs is extensionally universal.*

That is, any decidable over-approximation of an extensional set of programs contains at least one program for any computable partial function. Among other, it must contain at least one program that always loops. Thus, for example, a decidable set of programs capturing all the programs that compute the constant function $x \mapsto 0$ must also capture (at least) one program that always loops, one that computes the identity, one for sorting, for Ackermann function, for solving SAT, ... There is no hope of reaching intensional completeness and keeping extensional soundness.

Note that, contrary to Rice's Theorem, we only require the set of program to be non-empty rather than non-trivial. Indeed, the trivial set of all programs is certainly decidable, partially extensional and extensionally universal...

The proof will go by *reductio ab absurdum*, that is by supposing that $\mathtt{P}$ is **not** extensionally universal which is equivalent to saying that $\overline{\mathtt{P}}$ is also partially extensional for some (unknown) function. As mentioned in the discussion at the beginning of the Section, we cannot assume that this function is the never defined one (like in the proof of Rice's Theorem) and hence need to find another undecidable problem, which will be linked to the recursively inseparable sets we've introduced.

Proof. Let f be a function and let $\mathtt{P}$ be a decidable set of programs that is partially extensional for it: it contains all the programs computing f: $\mathcal{P}_f \subset \mathtt{P}$, that is $[\![\mathtt{s}]\!] = f \Rightarrow \mathtt{s} \in \mathtt{P}$. Let $\mathtt{p}$ be a program computing f: $[\![\mathtt{p}]\!] = f$, thus $\mathtt{p} \in \mathtt{P}$.

Assume, for contradiction, that $\mathtt{P}$ is not extensionally universal, that is there is a program $\mathtt{q}$ such that no program computing the same function is in $\mathtt{P}$: $[\![\mathtt{s}]\!] = [\![\mathtt{q}]\!] \Rightarrow \mathtt{s} \notin \mathtt{P}$.

Now, for every program $\mathtt{r}$, construct program $\mathtt{r}'$ which, on input x, works as follows:

- simulate program $\mathtt{r}$ on input 0; if it terminates and
  - if the result is 0, simulate program $\mathtt{p}$ on input x;
  - if the result is not 0, simulate program $\mathtt{q}$ on input x;

Note that given $\mathtt{r}$, $\mathtt{r}'$ is effectively computable (in a loose syntax, $\mathtt{r}'$ is essentially `if r(0)=0 then p(x) else q(x)`, details will obviously depend on the actual choice of $\mathcal{P}$). Now, depending on $[\![\mathtt{r}]\!](0)$, we have three cases.

- If $[\![\mathtt{r}]\!](0) = 0$, then $[\![\mathtt{r}']\!] = [\![\mathtt{p}]\!]$, hence because $\mathtt{P}$ is partially extensional for $f = [\![\mathtt{p}]\!]$, $\mathtt{r}' \in \mathtt{P}$.
- If $[\![\mathtt{r}]\!](0) \neq 0$ but it does terminates, then $[\![\mathtt{r}']\!] = [\![\mathtt{q}]\!]$, and hence $\mathtt{r}' \notin \mathtt{P}$ by hypothesis.
- If $[\![\mathtt{r}]\!](0)$ does not terminate, then we don't know whether $\mathtt{r}' \in \mathtt{P}$ or not.

Now, consider the set $C = \{\,\mathtt{r} \mid \mathtt{r}' \in \mathtt{P}\,\}$. As $\mathtt{P}$ is decidable (by hypothesis) and $\mathtt{r}'$ is effectively computable from $\mathtt{r}$, then C is decidable. But by the above observations $\{\,\mathtt{r} \mid [\![\mathtt{r}]\!](0) = 0\,\} \subseteq C$, and $\{\,\mathtt{r} \mid [\![\mathtt{r}]\!](0) \notin \{0, \bot\}\,\} \bigcap C = \emptyset$. However, these sets cannot be recursively separated. Hence, C cannot be decidable and the assumption that $\mathtt{P}$ is not extensionally universal is wrong. □

The deadlock created by this situation is depicted in Figure 6. We know for sure that our over-approximation will catch programs computing the Ackermann function or solving the Hydra game. What is worse, is that we also know for sure that it will catch programs solving SAT and we cannot know whether they are false positive (red question mark) or if SAT is indeed PTIME (blue question mark).

Moreover, we don't know anything about what's **not** in the approximation. Finding a SAT solver out of it (green question mark) would provide a lower bound, but even for functions that we know are not in PTIME, it is well possible that **all** their programs are actually in the over-approximation (that is, there may be no red question marks for Ackermann or the Hydra).

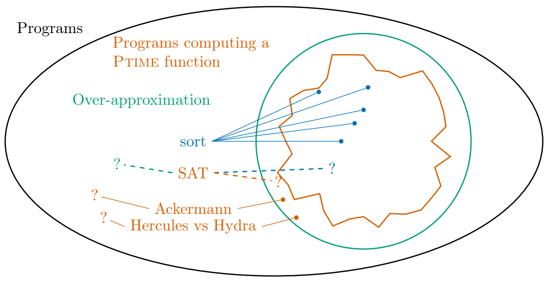


Figure 6: Extensionally universal over-approximation.

This Theorem is more general than Rice's Theorem:

Corollary 297 (Rice's Theorem). *Any (non-trivial) extensional set of programs is undecidable.*

Proof. Let $\mathtt{P}$ be an extensional decidable set of programs. If $\mathtt{P} = \emptyset$, then it is trivial and everything is alright. Otherwise, by definition, any extensional set is partially extensional. Hence, by the previous Theorem, $\mathtt{P}$ is extensionally universal.

Let $\mathtt{p} \in \mathtt{P}$ and $\mathtt{q}$ be any program. Because $\mathtt{P}$ is extensionally universal, there exists $\mathtt{q}'$ with $[\![\mathtt{q}]\!] = [\![\mathtt{q}']\!]$ and $\mathtt{q}' \in \mathtt{P}$. Because $\mathtt{P}$ is extensional, any program that computes the same function as $\mathtt{q}'$ is also in it. Especially $\mathtt{q} \in \mathtt{P}$.

Hence, $\mathtt{P}$ must be the trivial set containing all the programs. The only decidable extensional sets of programs are the trivial ones. □

The Theorem also has several more or less immediate corollaries which are well-known "folk knowledge" results in computer science usually explained by an argument of "adding arbitrarily many dummy `x:=x` instructions" and to which the Theorem gives a bit more punch.

Corollary 298 (Many programs). *For any computable function f, there are infinitely many programs computing it.*

Proof. If $\mathcal{P}_f$ is finite, it is decidable. By construction, it is partially extensional for f, hence by the Theorem it must be extensionally universal, but because there are infinitely many computable functions, any extensionally universal set must be infinite, contradicting the supposition that $\mathcal{P}_f$ is finite. □

Note that this is a bit less trivial than what it appears. This says that it is not possible to design a programming language with only finitely many possible implementations of a given function, without loosing some important ingredient for computability (universal machine, programs as data object, composition of programs, ...)

In turn, this leads to the question of *which* ingredients of computability must we loose in order to have only finitely many programs for a given function. Typically, even if Primitive Recursive *programs* are far from Turing-completeness, there still exists infinitely many PR program for each PR function (because, for example, the construction $\pi_0(\mathtt{p}, 0)$ is PR and creates infinitely many programs computing the same function).

We assume given a reasonable notion of size of both programs and their inputs (number of symbols of the source code, number of states of the TM, ...) noted $|\mathtt{p}|$ or $|x|$.

Corollary 299 (Arbitrary size)**.** *For any N, every computable function is computable by a program of size larger than N.*

Proof. Let $\mathtt{P} = \{\, \mathtt{p} \mid |\mathtt{p}| \leq N \,\}$. It is finite, hence by previous Corollary cannot contain all the programs computing a given function. □

4.3 Asperti-Rice

By replacing $\mathfrak{R}$ by $\mathfrak{A}$ in the previous Theorem, we can easily obtain an "intensional Asperti-Rice" Theorem stating that any decidable non-empty set containing at least one class of $\mathfrak{A}$ intersect all the classes of $\mathfrak{A}$. Changing the base equivalence from $\mathfrak{R}$ to another will be the subject of the second generalisation, hence we do not need to develop this further for now.

5 Intensionality: The Rice-Shapiro Theorem

Rice's Theorem is about extensional and decidable sets and basically states that they don't exist (except the trivial ones). However, it is obvious that extensional *semi*-decidable sets do exists, *e.g.* the set of programs that terminate on input 0 is extensional and RE (simply run $\mathtt{p}$ on input 0 and if it terminates, answer "Yes").

Rice-Shapiro Theorem [MS55, Sha56] investigates the RE extensional sets and gives a complete characterisation of them: they must be *compact* and *upward closed*. We will, similarly, look at RE partially extensional sets.

5.1 Rice-Shapiro Theorem

Classical presentations of Rice-Shapiro Theorem are done with RE sets of *functions* which are, in turn, defined using index sets. Sets of functions are exactly extensional sets of programs. Because we want to look at partially extensional sets, we will already express everything directly in term of *programs* and not *functions*.

We partially order functions and programs by agreeing on the smaller domain of definition:

Definition 300 (Domain ordering)**.** Let f_1, f_2 be two partial functions defined on D_1, D_2 we say that $f_1 \preceq f_2$ iff $D_1 \subseteq D_2$ and $\forall x \in D_1, f_1(x) = f_2(x)$. That is, f_1 is less defined than f_2 and both agree on the shared domain. Alternatively, we have $f_1 \preceq f_2$ iff $\forall x, f_1(x) = \perp$ or $f_1(x) = f_2(x)$.

Let f be a function. We note $\uparrow f$ the set of functions larger than it: $\uparrow f = \{\, g \mid f \preceq g \,\}$.

Similarly for programs, we say that $\mathtt{p} \precsim \mathtt{q}$ iff $\forall x, \mathtt{p}(x) = \perp$ or $\mathtt{p}(x) = \mathtt{q}(x)$ and we note $\uparrow \mathtt{p} = \{\, \mathtt{q} \mid \mathtt{p} \precsim \mathtt{q} \,\}$.

Note that this is really an ordering on functions while it is only a preorder on programs (because the definition is extensional), hence the different symbols.

Theorem 301 (Rice-Shapiro)**.** *Let $\mathtt{P}$ be an extensional set of programs. It is recursively enumerable if and only if it is compact and upward-closed:*

$$\mathtt{p} \in \mathtt{P} \Leftrightarrow \exists \mathtt{q} \in \mathtt{P}, \mathtt{q} \text{ is defined on finitely many input and } \mathtt{q} \precsim \mathtt{p}$$

Compactness ("if" direction) here means that any element of $\mathtt{P}$ has a "finite subset" in $\mathtt{P}$ (or rather a smaller element with finite domain which, when function are seen as set of couples, is indeed a finite subset). Upward closure ("only if" direction) comes from the following construction:

If $\mathtt{p} \precsim \mathtt{p}'$ and $\mathtt{p} \in \mathtt{P}$, then by the Theorem there exists $\mathtt{q} \in \mathtt{P}$ with finite domain and $\mathtt{q} \precsim \mathtt{p}$. However, by transitivity we have $\mathtt{q} \precsim \mathtt{p}'$ and thus using the "only if" direction of the Theorem, we can conclude that $\mathtt{p}' \in \mathtt{P}$. This implies that $\uparrow \mathtt{p} \subset \mathtt{P}$, *i.e.* $\mathtt{P}$ is upward closed.

Rice-Shapiro Theorem allows to show:

- the set of programs which, on input 0, return 0 is RE (it is exactly $\uparrow \mathtt{p}$ with $\mathtt{p}(\mathtt{x})$ = `if (x=0) then return 0 else loop`);
- the set of programs which terminate on input 14 is RE (by choosing $\mathtt{q}$ defined only on input 14 and returning the same result as $\mathtt{p}$);
- the set of programs which on inputs less than 100 return a result more than 1000 is RE;

- the set of programs which terminate on all inputs is not RE (because it is not compact: it does not contain programs defined on a finite domain);
- the set of programs which never terminate is not RE (it is not upward closed);
- the set of programs computing a monotonic (resp. injective) partial function is not RE (it is not upward closed as a monotonic partial function can be extended in a non-monotonic way);
- ...

Roughly speaking, Rice's Theorem says that inside decidability we cannot say anything on the input-output relation of programs. Rice-Shapiro Theorem states that moving to RE allows us to say something about a *finite* set of input-output constraints but as soon as we want to look at infinitely many inputs, the set is not RE.

In this sense, Rice's Theorem tells that we cannot ask "what is the result on input 0" because we can't even know if there will eventually be a result. It is about the need of the infinite *time* needed to run a program before finding out it will never ends. Rice-Shapiro Theorem tells that even if we could ask questions on specific inputs, there is another infinity hidden in functions which is the infinite *space* needed to run a program on infinitely many inputs simultaneously.

Proof of Rice-Shapiro Theorem. Let $\mathtt{P}$ be an extensional RE set of programs and $\mathtt{p} \in \mathtt{P}$.

Upward closure ($\Leftarrow$): Suppose that there exist $\mathtt{q} \gtrsim \mathtt{p}$ and $\mathtt{q} \notin \mathtt{P}$. Let K be any RE, not decidable set (of integers). By definition, there exist a program $\mathtt{p}_K$ such that $\mathtt{p}_K(x) \neq \bot$ iff $x \in K$. For any integer k, we build the program $\mathtt{q}_k$ as follows:

On input x, run in parallel $\mathtt{p}(x)$ and $\mathtt{p}_K(k)\mathtt{;q}(x)$ and return the first result (if any). Running in parallel can mean interleaving, parallel composition, or whatever construction the programming language has, thus in a loose syntax, we have $\mathtt{q}_k(x) = \mathtt{p}(x)||(\mathtt{p}_K(k)\mathtt{;q}(x))$.

Because $\mathtt{p} \lesssim \mathtt{q}$, we know that whenever $\mathtt{p}(x)$ returns a result, $\mathtt{q}(x)$ will also return the same result. Thus, when $k \in K$ and $\mathtt{p}(x)$ terminates, the second branch of the composition will necessarily return the same result and $\mathtt{q}_k$ is well defined (especially, its result does not depend on race condition between the parallel branches).

Now, if $k \in K$, the test in the second branch will always be true and $\mathtt{q}_k(x)$ will always have the same result as $\mathtt{q}(x)$. That is, $[\![\mathtt{q}_k]\!] = [\![\mathtt{q}]\!]$ and by extensionality, $\mathtt{q}_k \notin \mathtt{P}$.

However, if $k \notin K$, the second branch will never gives a result and $\mathtt{q}_k(x)$ will always have the same result as $\mathtt{p}(x)$. That is, $[\![\mathtt{q}_k]\!] = [\![\mathtt{p}]\!]$ and by extensionality, $\mathtt{q}_k \in \mathtt{P}$.

Thus, we have $\overline{K} = \{ k \mid \mathtt{q}_k \in \mathtt{P} \}$ but by hypothesis the former is not RE while the latter is, an absurdity. Thus we must conclude that no such $\mathtt{q}$ existed in the first place. So, $\mathtt{P}$ must be upward closed.

Now, if there is a finitely defined program $\mathtt{q} \in \mathtt{P}$ and $\mathtt{q} \lesssim \mathtt{p}$, we can conclude that $\mathtt{p} \in \mathtt{P}$, thus proving the "only if" part of the Theorem.

Compactness ($\Rightarrow$): Suppose that there is no finitely defined $\mathtt{q} \lesssim \mathtt{p}$ in $\mathtt{P}$. For any program $\mathtt{q}$ and integer n build the program $\mathtt{q}_n$ as follows:

$\mathtt{q}_n(t) =$ `if` $\mathtt{q}(n)$ terminates in t steps or less `then loop else` $\mathtt{p}(t)$. Again, the "clocked simulation" of $\mathtt{q}(n)$ for t steps is highly dependent on the actual choice of programming language (and cost model) but is nonetheless always possible. Also, the "clocked simulation" always terminates as we just stop it after t steps, hence we will always execute one of the branch of the test.

Now, if $\mathtt{q}(n)$ never terminates, then we are always in the "else" branch of the test and $\mathtt{q}_n(t)$ always return the same thing as $\mathtt{p}(t)$ hence $[\![\mathtt{q}_n]\!] = [\![\mathtt{p}]\!]$ and by extensionality, $\mathtt{q}_n \in \mathtt{P}$.

However, if $\mathtt{q}(n)$ does terminates, it does so in T steps. Thus, if $t < T$ we are in the "else" branch of the test and $\mathtt{q}_n(t) = \mathtt{p}(t)$ but if $t \geq T$, we are in the "then" branch and $\mathtt{q}_n(t) = \bot$. That is, $\mathtt{q}_n \lesssim \mathtt{p}$ and because there are only finitely many $t < T$, $\mathtt{q}_n$ has finite domain. Hence, by hypothesis, $\mathtt{q}_n \notin \mathtt{P}$.

Thus we have $\{ n \mid \mathtt{q}_n \in \mathtt{P} \} = \{ n \mid \mathtt{q}(n) \text{ does not terminates} \}$ but the former is RE and the later is not, an absurdity. Thus there must exist in $\mathtt{P}$ a finitely defined program smaller than $\mathtt{p}$. □

5.2 Intensional version

Again, the proof actually only require extensionality of $\mathtt{P}$ for $\mathtt{p}$ and of $\overline{\mathtt{P}}$ for some $\mathtt{q} \gtrsim \mathtt{p}$. Thus, Rice-Shapiro Theorem can immediately be extended to partially extensional sets by splitting the compactness and upward closure parts:

Corollary 302. *Let $\mathtt{P}$ be a RE set of programs, partially extensional for $\mathtt{p}$ then:*

- *there exist a finitely defined $\mathtt{p}' \lesssim \mathtt{p}$ and $\mathtt{p}' \in \mathtt{P}$;*
- *for any function $f \succeq [\![\mathtt{p}]\!]$, there exists a program $\mathtt{q} \in \mathtt{P}$ computing f.*

The compactness is kept because it is existential and thus finding one program is enough. The upward closure is universal ("all larger programs") and thus need to be adapted to partial extensionality, we cannot guarantee that all the more defined programs are in the set, but at least one that compute the same thing. In this sense, it is an extensional upward closure rather than a full upward closure (similar to extensional universality in the generalisation of Rice's Theorem).

For the same reason, we cannot easily give an "if and only if" formulation that would characterise the partially extensional RE sets. Indeed, the existence of a finite $\mathtt{p} \precsim \mathtt{q}$ in $\mathtt{P}$ only ensure that there exists $\mathtt{q}' \in \mathtt{P}$ with the same semantics as $\mathtt{q}$ in it but nothing is known on $\mathtt{q}$ itself.

We can show a more powerful result in the line of "over-approximations are of little interest".

Definition 303 (Close enough)**.** Let f, g be two partial functions. We say that they are *close enough*, noted $f \risingdotseq g$, iff they differ only on a finite set of inputs:

$$f \risingdotseq g \Leftrightarrow \{\, x \mid f(x) \neq g(x) \,\} \text{ is finite.}$$

Theorem 304. *Let $\mathtt{P}$ be RE set of programs which is partially extensional for some function.*

Then, for any computable function f, there exists a program $\mathtt{p} \in \mathtt{P}$ computing a close enough function:

$$\forall f, \exists \mathtt{p} \in \mathtt{P} / [\![\mathtt{p}]\!] \risingdotseq f$$

In the decidable case, $\mathtt{P}$ was forced to contain a program for any single function (extensional universality). Here, it won't be extensionally universal anymore but still get close enough to any function.

Proof. Let $\mathtt{p}$ be a program and $\mathtt{P}$ be a recursively enumerable set of programs partially extensional for $[\![\mathtt{p}]\!]$: $[\![\mathtt{q}]\!] = [\![\mathtt{p}]\!] \Rightarrow \mathtt{q} \in \mathtt{P}$. Let f be a function and $\mathcal{P}_f = \{\, \mathtt{q} \mid [\![\mathtt{q}]\!] = f \,\}$ be the set of all programs computing it.

Let $\mathtt{r} \in \mathcal{P}_f$ and assume, for contradiction, that $\mathtt{P}$ contains no program that differs from $\mathtt{r}$ on at most finitely many inputs.

Let $\mathtt{q}$ be a program and n be an integer, and construct $\mathtt{q}_n$ which, on input t, simulates program $\mathtt{q}$ on input n for t steps and if it halted, returns $\mathtt{r}(t)$ otherwise ends the simulation and returns $\mathtt{p}(t)$. In loose syntax, we have $\mathtt{q}_n(t) =$ **`if`** $\mathtt{q}(n)$ terminates in t steps or less **`then`** $\mathtt{r}(t)$ **`else`** $\mathtt{p}(t)$.

We have two cases:

- If $\mathtt{q}(n)$ never halts, then $\mathtt{q}_n$ always computes the same thing as $\mathtt{p}$. Thus, because $\mathtt{P}$ is partially extensional for $[\![\mathtt{p}]\!]$, $\mathtt{q}_n \in \mathtt{P}$.
- If $\mathtt{q}(n)$ halts in T steps, then $\mathtt{q}_n$ agrees with $\mathtt{p}$ on finitely many inputs $0, \dots, T$ and agrees with $\mathtt{r}$ on infinitely many inputs $T+1, \dots$ Thus, $\mathtt{q}_n$ differs from $\mathtt{r}$ on finitely many inputs and by assumption $\mathtt{q}_n \notin \mathtt{P}$.

Hence, $\{\, n \mid \mathtt{q}(n) \text{ does not halt} \,\} = \{\, n \mid \mathtt{q}_n \in \mathtt{P} \,\}$ but the former is not RE (it is co-RE complete) while the later is, an absurdity.

Therefore, for any program $\mathtt{r}$, $\mathtt{P}$ must contain a program close enough to $\mathtt{r}$. ☐

Note that by construction, we can ensure that the program close enough to $\mathtt{r}$ returns, when they disagree the same result as $\mathtt{p}$ (for which $\mathtt{P}$ is partially extensional). In this way, we are very close to the compactness part of the original Rice-Shapiro Theorem which stated that an extensional RE $\mathtt{P}$ must contain something close enough to the infinite loop which, when it returns, has the same result as another program for which $\mathtt{P}$ is extensional.

And the adaptation of one proof to the other is done the same way as it was done for Rice's Theorem. We cannot suppose anymore that $\overline{\mathtt{P}}$ is partially extensional for the infinite loop to build a contradiction, so we suppose it is partially extensional for some function and adapt the reduction from non-termination problem.

Obviously, any extensional set being partially extensional, we can use the exact same proof to show that any extensional RE set contains a program computing something close enough to a given computable function f. This is the traditional "back-and-forth" construction of Rice-Shapiro Theorem:

Start with any program $\mathtt{p} \in \mathtt{P}$. By compactness, there exists a finite $\mathtt{q} \precsim \mathtt{p}$ which is also in $\mathtt{P}$. But by upward closure, the program that agrees with $\mathtt{q}$ when it is defined and computes f otherwise must also be in the set. It is, by construction, close enough to f.

For example, if $\mathtt{P}$ is a recursively enumerable set of programs partially extensional for a total function (*e.g.* the constant function $x \mapsto 0$), then by choosing f as the infinite loop, we can conclude that $\mathtt{P}$ must contain a program that loops on all but finitely many inputs; by choosing another constant function (*e.g.* $x \mapsto 1$), we can conclude that $\mathtt{P}$ contains a program whose result is almost always 1, …

Thus, any enumeration of all the programs that compute a total function will also enumerate programs that almost never terminate…

Again, over-approximations must contain a lot of unwanted stuff.

6 Intensionality: second generalisation

6.1 Equivalence

In order to replace $\mathfrak{R}$ by any equivalence, we need to adapt our vocabulary.

Let $\mathfrak{P} = \{\mathtt{P}_1, \mathtt{P}_2, \ldots\}$ be a partition of a set S and $\mathtt{P}$ be a subset of S, it is

- *compatible with* $\mathfrak{P}$ if it is the union of classes of $\mathfrak{P}$;
- *partially compatible* for a block $\mathtt{P}_k$ if it contains all the elements in that block: $\mathtt{P}_k \subset \mathtt{P}$. Note that it does not need to be compatible for other blocks;
- *complete* for a set of blocks $\mathtt{P}_{k_1}, \mathtt{P}_{k_2}, \ldots$ if it contains at least one element of each of these blocks: $\mathtt{P} \bigcap \mathtt{P}_{k_i} \neq \emptyset$;
- *sound* for a set of blocks $\mathtt{P}_{k_1}, \mathtt{P}_{k_2}, \ldots$ if it is contained in the union of these blocks: $\mathtt{P} \subset \bigcup \mathtt{P}_{k_i}$;
- an *ICC characterisation* of a set of blocks if it is both sound and complete for it;
- *universal* if it is complete for $\mathfrak{P}$ itself: it contains at least one element in each block of the partition.

6.2 Switching functions

Definition 305. Let S and T be two sets and $\approx$ be an equivalence on T. Let $\mathcal{S} \subset \mathscr{P}(S)$ and $\mathcal{T} \subset \mathscr{P}(T)$ be families of subsets of S and T.

A *switching function* is a total function $\pi : S \to T$. It is *$\mathcal{S}$-$\mathcal{T}$-continuous* if the reverse image of any set in $\mathcal{T}$ is in $\mathcal{S}$: $\forall T' \in \mathcal{T}, \pi^{-1}(T') \in \mathcal{S}$.

Let $a, b \in T$. A switching function is *a-b-$\mathcal{S}$-intricated* with $\approx$ if the sets $A_\pi = \{\, x \in S \mid \pi(x) \approx a \,\}$ and $B_\pi = \{\, x \in S \mid \pi(x) \approx b \,\}$ are $\mathcal{S}$-inseparable.

A *switching family* indexed by $T \times T$ is a family of switching functions, $I = (\pi_{a,b})_{a,b \in T}$.

A switching family is *$\mathcal{S}$-intricated* with $\approx$ if for each pair of elements $a, b \in T$, the switching function $\pi_{a,b}$ is a-b-$\mathcal{S}$-intricated:

$$\forall a, b \in T,\ A_{a,b} = \{\, x \in S \mid \pi_{a,b}(x) \approx a \,\} \text{ and } B_{a,b} = \{\, x \in S \mid \pi_{a,b}(x) \approx b \,\} \text{ are } \mathcal{S}\text{-inseparable.}$$

If $S = T$ and $\mathcal{S} = \mathcal{T}$, then we can simply speak of $\mathcal{S}$-continuity.

Note that if $a \approx b$ then A_π and B_π (or $A_{a,b}$ and $B_{a,b}$) are the same and hence cannot be $\mathcal{S}$-separated.

Note that there may be elements in S which are neither in A_π nor in B_π, that is such that $\pi(x)$ is not equivalent to a and not equivalent to b.

Note that the "constant projections" $\pi_{a,b}(x) = a$ and $\pi_{a,b}(x) = b$ can be switching functions but in general switching functions in a family do not need to ignore one of their index.

Example 306 (Standard Switching family)**.** Let $\mathcal{P}$ be the set of programs and set $S = T = \mathcal{P}$, we define the *standard switching family* on it as:

$$\pi_{\mathtt{p},\mathtt{q}}(\mathtt{r})(x) = \mathtt{r}'(x) = \texttt{if r(0)=0 then p(x) else q(x)}$$

- Each function in the family is REC-continuous (or REC-REC-continuous to use the correct notation). Indeed, let D be a decidable set, to decide if $x \in \pi^{-1}(D)$, it suffices to compute $\pi(x)$ and decide $\pi(x) \in D$. Because each function in the family is computable, this is doable.
- Note that each $\pi_{\mathtt{p},\mathtt{q}}$ is actually computable with a low complexity (notably, it is computable in polynomial time). Hence, it is also, for example, PTIME-continuous. On the other hand, it is not clear whether, say, any PTIME-continuous function must be PTIME.
- The switching family is REC-intricated with $\mathfrak{R}$. Indeed, if we fix $\mathtt{p}, \mathtt{q}$ with $[\![\mathtt{p}]\!] \neq [\![\mathtt{q}]\!]$ then we have

  $$A_{\mathtt{p},\mathtt{q}} = \{\, \mathtt{r} \mid [\![\pi_{\mathtt{p},\mathtt{q}}(\mathtt{r})]\!] = [\![\mathtt{p}]\!] \,\}$$

  We've seen in the proof of Theorem 296 that $A_{\mathtt{p},\mathtt{q}} = \{\, \mathtt{r} \mid \mathtt{r}(0) = 0 \,\}$ and $B_{\mathtt{p},\mathtt{q}} = \{\, \mathtt{r} \mid \mathtt{r}(0) \notin \{0, \perp\} \,\}$ and these sets are recursively inseparable by Corollary 295.
- Of course, REC-intrication also implies, for example, PTIME-intrication as the PTIME sets are all recursive.
- Similarly, the standard switching family is REC-intricated with $\mathfrak{B}_\Phi$ and $\mathfrak{A}$ as used in the various proofs of previous results.

Basically, this example both motivates the definition and shows that non-trivial continuous and intricated switching families do exist.

6.3 Main result

Theorem 307. *Let S, T be two sets, $\mathcal{S} \subset \mathscr{P}(S)$, $\mathcal{T} \subset \mathscr{P}(T)$ be two families of subsets such that $\mathcal{S}$ is closed under complements and $\mathfrak{P}$ be an equivalence on T.*

Let $I = (\pi_{a,b})_{a,b \in T}$ be a switching family of $\mathcal{S}$-$\mathcal{T}$-continuous functions which is $\mathcal{S}$-intricated with $\mathfrak{P}$.

Any non-empty $T' \in \mathcal{T}$ which is partially compatible with $\mathfrak{P}$ is also universal for it.

That is, the existence of an intricated switching family of continuous functions is enough to ensure that any partially compatible set in the family is also universal.

In the first generalisation, we had for $\mathcal{S}$ the family of all decidable sets and took $\mathfrak{P} = \mathfrak{R}$. The existence of an intricated switching family was not required in the statement of the Theorem but used in the proof (with the standard switching family). Which resulted in the final statement that any non-empty, decidable (*i.e.* in $\mathcal{T}$) partially extensional (*i.e.* partially compatible with $\mathfrak{R}$) set must be extensionally universal.

Proof. Let $a, b \in T$, we note $[a]$ and $[b]$ there respective classes in the equivalence $\mathfrak{P}$. Let $T' \in \mathcal{T}$ and suppose that $[a] \subset T'$ (T' is partially compatible) but $[b] \bigcap T' = \emptyset$ (T' is not universal).

Let $x \in S$ be any element and set $y = \pi_{a,b}(x)$. We have three cases:

- If $x \in A_{a,b}$ then by definition $y = \pi_{a,b}(x)\mathfrak{P}a$. Hence, $y \in [a]$ and by partial compatibility, $y \in T'$.
- If $x \in B_{a,b}$ then by definition $y = \pi_{a,b}(x)\mathfrak{P}b$. Hence, $y \in [b]$ and by hypothesis, $y \notin T'$.
- Otherwise, we don't know whether $y \in T'$ or not.

Now, let $C = \{ x \mid y = \pi_{a,b}(x) \in T' \}$. We have immediately $C = \pi_{a,b}^{-1}(T')$. Because $T' \in \mathcal{T}$ and $\pi_{a,b}$ is $\mathcal{S}$-$\mathcal{T}$-continuous, we have $C \in \mathcal{S}$. Because $\mathcal{S}$ is closed under complements, we have $\overline{C} \in \mathcal{S}$.

So $x \in A_{a,b}$ implies $y \in T'$ hence $A_{a,b} \subset C$. On the other hand, $x \in B_{a,b}$ implies $y \notin T'$ hence $B_{a,b} \subset \overline{C}$. And obviously, $C \bigcap \overline{C} = \emptyset$. In other words, $A_{a,b}$ and $B_{a,b}$ are $\mathcal{S}$-separated by C and $\overline{C}$, contradicting the $\mathcal{S}$-intrication of the switching family.

Thus, there cannot exist a class $[b]$ of $\mathfrak{P}$ that does not intersects T', and T' must be universal. □

Note that the existence of an intricated switching family is enough. Details about what this family actually is are not needed in any way.

This Theorem implies, among other, that no (non-trivial) union of classes of the equivalence is in $\mathcal{T}$.

Corollary 308. *Under the same hypothesis:*

- *any over-approximation in $\mathcal{T}$ of a non-trivial union of classes of $\mathfrak{P}$ must intersect each class;*
- *if $\mathcal{T}$ is also closed under complement, then any under-approximation in $\mathcal{T}$ of a non-trivial union of classes of $\mathfrak{P}$ must leave out one element of each class;*

Proof. By definition, an approximation of a union of classes is partially compatible. Hence it must be universal.

If T' is an under-approximation of a non-trivial union of classes, U, then its complement $\overline{T'}$ is an over-approximation of $\overline{U}$ which is non-trivial if U is non-trivial. Thus, if $\mathcal{T}$ is closed under complement, then $\overline{T'} \in \mathcal{T}$ is partially compatible for $\overline{U}$ and must intersect each class. Therefore, T' must leave out an element of each class it is trying to under-approximate. □

6.4 Examples

The examples here illustrate the power of this result in various situations. This is obviously not an exhaustive list of applications and it is especially easy to extend these to similar cases.

As stated, the second generalisation is more general than the first:

Example 309. Let $S = T = \mathcal{P}$ be the set of programs and $\mathcal{S} = \mathcal{T} = \text{REC}$ be the family of decidable sets. It is closed under complements.

As shown in Example 306, the standard switching family is REC-intricated with $\mathfrak{R}$ and contains only REC-continuous functions.

Hence, any decidable non-empty set containing an union of classes of $\mathfrak{R}$ must intersect all the classes of $\mathfrak{R}$ (Theorem 296).

Similarly, the standard switching family is REC-intricated with $\mathfrak{A}$ (or with $\mathfrak{B}_\Phi$). Hence, any decidable non-empty set containing an union of classes of $\mathfrak{A}$ must intersect all the classes of $\mathfrak{A}$.

Example 310 (Complexity)**.** As stated above, the standard switching family is REC-intricated with $\mathfrak{A}$. Hence, any decidable over-approximation of an union of classes of $\mathfrak{A}$ must intersect all of them, that is, include programs of arbitrarily high (or low) complexity.

For example, we can consider the set of all polytime **programs**. Note that this is not an extensional set as it only contains the "efficient" programs and does not contains the inefficient programs computing "good" functions (*e.g.*, it does not contains the programs that sort in non-polynomial time). Thus, Rice's Theorem cannot say anything about it. Using Asperti-Rice Theorem, we can prove that this set is undecidable.

But now, we can do even more. We can show that any decidable over-approximation of the set of polytime programs must contains elements of **any** class of $\mathfrak{A}$. In other words, it must contains programs computing any function with any (time) complexity, and especially of any arbitrarily high complexity: it must contain not only exponential sorting programs but also non primitive recursive searching programs and even non multiple recursive programs computing the identity!

Similarly, the set of linear space programs (or quadratic time, or . . .) is a union of classes of $\mathfrak{A}$. Contrarily to polytime programs, it is not closed under composition. However, we still have the same result: any decidable over-approximation of the linear space programs must contain programs of arbitrarily high space complexity.

Example 311 (ICC)**.** Conversely, we may want to look at a decidable under-approximation of the set of polytime programs, that is, more or less, an ICC criterion. However, this is the complement of a decidable over-approximation of the set of non-polytime programs, an union of classes of $\mathfrak{A}$. We can apply our Result to this over-approximation and show that it must contain programs of arbitrarily low complexity. Thus, any decidable under-approximation of the polytime programs must leave out programs of constant complexity, or quadratic time sorting programs!

Even before requiring extensional completeness, we know for sure that any polytime-sound decidable set is doomed to leave out many programs with low complexity (at least one per function per complexity). This also greatly block any attempt at getting close to intensional completeness. Any ICC criterion will necessarily have a somewhat low expressivity as it must leave out many "good" programs.

Trying to go from the polytime programs to the programs computing functions in Ptime (that is, also accepting inefficient programs for good functions) won't help. The set of programs computing a Ptime function is still a union of classes of $\mathfrak{A}$. Thus, any under-approximation of it must still leave out at least one program for each function and each complexity. . .

Therefore, if we want to approximate the set of polytime programs by decidable under- and over-approximations as close as possible one from the other, we will necessarily have in the "grey zone" between them programs of any complexity, low or high. That is, such a "double bound" will always be very far from the actual set we're trying to study.

The new deadlock created by this situation is depicted on Figure 7. This time, we do not consider all programs computing a Ptime function, but only the programs that are themselves polytime (irregular red shape). Thus, the over-approximation (green circle) is an attempt to differentiate, *e.g.*, the "good" (efficient) sorting programs from the "bad" ones (blue and green dots).

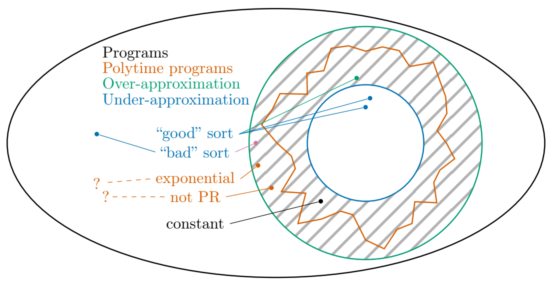


Figure 7: Complexity-universal over-approximation.

However, the over-approximation must contain programs of arbitrarily high-complexity. It must contain, among others, an exponential program or a non primitive recursive program (red dots). Thus it is very far from safe to try and consider that programs in the over-approximation have a "not too bad" complexity.

What is worse, we don't even know for sure that there actually are exponential programs **out** of the over-approximation (red question marks). It is well possible that the over-approximation actually contains non only all the polytime programs but also all the exponential time ones. . .

Even worse, if the goal is to discriminate good and bad sorting programs, we know for sure that the over-approximation must intersect all the classes of $\mathfrak{A}$ hence contain, *e.g.*, a sorting program whose complexity is not primitive recursive (pink dot)!

And if we also get a decidable under-approximation (blue circle) then we know for sure that it must leave out some constant time programs (black dot). And here also, we know for sure that at least one quadratic time sorting algorithm must be out of the under-approximation (green dot). . .

Thus, any attempt at approximating the set of polytime programs by a pair of decidable under- and over-approximations leaves out a grey area (hatched) that must necessarily contains basically examples of everything one may think about. . .

6.5 Input-output, viruses

We now build an example where having two sets S, T is useful.

Example 312 (Spambots)**.** Let S be the set of (one tape, 2 letters) Turing Machines and $T = \mathcal{P}$ be the set of all programs in a given language. Let $\mathcal{S}$ be the family of decidable sets of Turing Machines (it is closed under complements) and $\mathcal{T}$ be the family of decidable sets of programs.

We define the equivalence $\mathfrak{P}$ on T by $\mathtt{p}\mathfrak{P}\mathtt{q}$ iff for any input x, $\mathtt{p}(x)$ and $\mathtt{q}(x)$ send the same number of email. Note that this is not a complexity measure in the sense of Blum as the predicate "$\mathtt{p}(x)$ sends k emails" is not decidable ($\mathtt{p}$ may loop after sending $k-1$ emails).

Next, we consider given an evaluator $\mathtt{eval} \in \mathcal{P}$ which takes as inputs (the representation of) a Turing Machine, M, and an integer n and returns the result of M on input n: $[\![\mathtt{eval}]\!](M, n) = [\![M]\!](n)$. Moreover, we consider that `eval` never ever sends a single email. Such a `eval` do exist. Precise details on it obviously depends on the programming language and the representation of Turing Machines, but actually building such a program is not difficult and is a standard computability exercise.

Now, consider the switching family:

$$\pi_{\mathtt{p},\mathtt{q}}(M)(x) = \mathtt{r}(x) = \mathtt{if\ eval(M,0)=0\ then\ p(x)\ else\ q(x)}$$

This is a simple variation on the standard switching family. It is $\mathcal{S}$-$\mathcal{T}$-continuous because it is computable.

Because we've asked that `eval` never sends an email, we can easily see that $\pi_{\mathtt{p},\mathtt{q}}(M)\mathfrak{P}\mathtt{p}$ iff $[\![M]\!](0) = 0$ and $\pi_{\mathtt{p},\mathtt{q}}(M)\mathfrak{P}\mathtt{q}$ iff $[\![M]\!](0) \notin \{0, \perp\}$. And we know that these sets are REC-inseparable, hence the switching family is $\mathcal{S}$-intricated.

Note that, to show intrication, it is crucial that the evaluator do not send any email. Thus, we need the argument (here, the Turing Machine) to live in a world where email cannot be send and trying to do the same thing with $S = T = \mathcal{P}$ would be much more difficult.

Anyway, we have a $\mathcal{S}$-intricated switching family of $\mathcal{S}$-$\mathcal{T}$-continuous functions, thus we can apply our Theorem.

Any decidable over-approximation of the set of programs that send 0 email must contains programs that send an arbitrarily large number of emails. That is, if we try to accept all programs that won't send mail, we must also accept "Spambots" that will constantly send mails.

Similarly, trying to accept all programs that won't access Internet will result in also accepting botnets that often contact external servers and receive orders from them...

Computability in the Lattice of Equivalence Relations

Jean-Yves Moyen, Jakob Grue Simonsen

Work in progress

Abstract:

The set of equivalence relations on any set is equipped with a natural order that makes the set a complete (and atomistic, bounded and relatively complemented) lattice. The lattice structure only depends on the cardinality of the set, and thus the study of the lattice structure on any countably infinite set is (up to order-isomorphism) the same as studying the lattice of equivalence relations on the natural numbers.

We investigate computability in the lattice of equivalence relations on the natural numbers. This includes computability of the natural operations on the lattice, and investigating whether the subsets of appropriately defined subrecursive equivalence relations–for example the set of all polynomial-time decidable equivalence relations–form sublattices of the lattice.

While our work concern purely mathematical results, it is motivated by the possibility of using the lattice of equivalence relations to reason about computability and computational complexity.

1 Introduction

The purpose of this paper is to study sets of computable equivalence relations in the lattice $\mathrm{Equ}(S)$ of equivalence relations on any countably infinite set S. For example, the set of recursively enumerable equivalence relations on S form a proper sublattice, but the set of decidable equivalence relations is not even closed under finite joins, hence *a fortiori* not a sublattice.

The primary motivation is twofold: *Firstly*, from the point of view of pure mathematics, it is interesting to study those substructures of a very well-known order-theoretic structure such as $\mathrm{Equ}(S)$ that are induced by taking the collections of objects satisfying an interesting property, for example computability. *Secondly*, from the point of view of computer science, there is a strong connection between *program properties* and equivalence relations on certain classes of decidable sets, as explained below.

1.1 Motivation: complexity classes, implicit complexity and subrecursive equivalence relations

As there are uncountably many properties of programs (because each set of programs is a property), most properties must be undecidable. A celebrated negative result shows that there are even further barriers to decidability: Rice's Theorem [Ric53] states that every non-trivial, extensional property on programs is undecidable. An extensional property is one that depends solely on the (input-output) *function* computed by the program .

While Rice's theorem is six decades old, it still spurs new research trying to find the boundary of decidable progam properties, for example Asperti's work on complexity cliques [Asp08]. Such properties can be studied fruitfully by viewing results such as Rice's and Asperties as assertions about *equivalence relations*; indeed, any program property is an equivalence relation that has at most two classes: the class of programs having the property and the class of programs not having it (the equivalence relation has one class iff it is trivial, that is either no program has the property, or all programs do). Conversely, any equivalence relation on the set of all programs that has exactly two classes is a program property. In this view, Rice's Theorem studies the *extensional equivalence*, $\mathfrak{R}$, where two programs are equivalent if and only if they compute the same function. Rice's theorem thus states that *no equivalence class or union of equivalence classes in $\mathfrak{R}$ is decidable.*

Rather than studying individual equivalences such as $\mathfrak{R}$, it is interesting to look at the set of all equivalences between programs and at subsets of equivalences sharing certain properties. The set of all equivalence relations between programs has a very natural complete lattice structure. With this order, taking the union of some classes yields a larger equivalence relation and Rice's Theorem says that any class of any equivalence relation[1] in the principal filter at $\mathfrak{R}$ is undecidable.

Equivalence relations that are *not* extensional is a completely different kettle of fish. Some are easy to decide, such as "two programs are equivalent if they have the same length"; some others are decidable, but may require tremendous resources to decide, such as "two programs are equivalent if they have the same running time on all positive integer inputs below one million"; and some are harder than the halting problem, *e.g.* "two programs are equivalent if they both run in polynomial time" is Σ^0_2.

Our primary interest lies in investigating how properties of the lattice $\mathrm{Equ}(S)$ of equivalence relations on a set S interact with decidability or subrecursive computability of the equivalence relations in $\mathrm{Equ}(S)$. Typically, we wish to know whether, say, the set of equivalence relations decidable in exponential time is a sublattice or not. Such a characterisation could lead to the study of a subset of the equivalence relations that would approximate the whole set in a way similar than the rational numbers approximate the reals.

The lattice-theoretic properties of $\mathrm{Equ}(S)$ are well-understood; basic facts can be gleaned from the papers [Ore42, RS92] and the textbooks [Bir40, §8-9] [Grä03, Sec. IV.4]. As the lattice structure of $\mathrm{Equ}(S)$ is isomorphic to $\mathrm{Equ}(T)$ for any sets S and T of identical cardinality, we may without loss of generality consider the set of equivalences over the natural numbers, $\mathrm{Equ}(\mathbb{N})$. That is, we work modulo an unspecified encoding of programs. Observe that as long as the equivalence relations we consider do not use the fact that the elements are numbers rather than programs, the precise choice of the encoding (even an undecidable encoding) plays no role in our results. For example, if we prove a result about "the equivalence relations decidable in polynomial time", the result will hold regardless of whether we speak of equivalence relations of numbers or programs as long as there exists a polytime computable encoding between programs and numbers. On the other hand, if we prove a result about "the equivalence relations where all the even numbers are in the same class", then we do have an issue with the precise choice of the encoding. Note that *all results in the paper hold for both numbers and programs* via polytime encodings; if this is not obvious from the proof of a particular result, we describe the correspondence in the running text immediately following the proof.

In the following, we will consider three broad classes of equivalences: the *automatic* ones, decidable by an automaton; the *subrecursive* ones, decidable by a Turing Machine operating within some restricted time or space bound; and the *superrecursive* ones, corresponding to sets in the arithmetical hierarchy.

[1]Except the trivial one where all elements are equivalent.

2 Preliminaries on orders, lattices, equivalence relations, computability, and complexity

2.1 Equivalence relations

We consider the set Equ($\mathbb{N}$) of all equivalences over natural numbers. If $\mathcal{E}$ is an equivalence, we write as usual $m\mathcal{E}n$ to say that m and n are equivalent.

Equ($\mathbb{N}$) is ordered by $\mathcal{E} \leq \mathcal{E}'$ iff $m\mathcal{E}n \Rightarrow m\mathcal{E}'n$. Note that if an equivalence is seen as a subset of $\mathbb{N} \times \mathbb{N}$, this order is exactly the subset ordering $\subseteq$ on $\mathbb{N} \times \mathbb{N}$. By standard correspondence between equivalences and partitions, this is isomorphic with the *refinement ordering* on partitions.

We can easily see that $\mathcal{E} \leq \mathcal{E}'$ iff each class of $\mathcal{E}'$ is the union of one or more classes of $\mathcal{E}$. If $\mathcal{E} < \mathcal{F}$ and there is no $\mathcal{G}$ such that $\mathcal{E} < \mathcal{G} < \mathcal{F}$ then we say that $\mathcal{E}$ is a predecessor of $\mathcal{F}$, and $\mathcal{F}$ is a successor of $\mathcal{E}$, written $\mathcal{E} \prec \mathcal{F}$. This is the same as saying that $\mathcal{F}$ covers $\mathcal{E}$. In this case, $\mathcal{F}$ is obtained from $\mathcal{E}$ by merging exactly two of its classes.

We'll use cursive letters $\mathcal{E}$, $\mathcal{F}$, ... to design equivalences and bold capital roman letters $\mathbf{C}$, $\mathbf{D}$, ... to design sets of equivalences.

2.2 Lattices and lattice operations

(Equ($\mathbb{N}$), $\leq$) is a lattice with the following operations:

- The meet (greatest lower bound) of $\mathcal{E}$ and $\mathcal{F}$ is $\mathcal{G} = \mathcal{E} \wedge \mathcal{F}$ such that $m\mathcal{G}n$ iff $m\mathcal{E}n$ and $m\mathcal{F}n$. In this case, the classes of $\mathcal{G}$ are exactly the (non-empty) intersections of one class of $\mathcal{E}$ and one class of $\mathcal{F}$.
- The join (lowest upper bound) of $\mathcal{E}$ and $\mathcal{F}$ is $\mathcal{G} = \mathcal{E} \vee \mathcal{F}$ such that $m\mathcal{G}n$ iff there exists a finite sequence $a_1, \dots, a_k$ such that $m\mathcal{E}a_1\mathcal{F}a_2\mathcal{E}\dots\mathcal{F}n$.

Note that when expressing the join, the sequence of equivalences may end either with $\mathcal{E}$ or $\mathcal{F}$ depending on the parity of k (and, similarly, may start with either). However, it is always possible to choose $a_k = n$ and add $n\mathcal{F}n$ at the end (and similarly at the start). So, for the sake of simplicity, we'll assume wlog. that such sequences always start with $\mathcal{E}$ and end with $\mathcal{F}$.

If equivalence relations are seen as subsets of $\mathbb{N} \times \mathbb{N}$, then $\mathcal{E} \wedge \mathcal{F} = \mathcal{E} \bigcap \mathcal{F}$ while $\mathcal{E} \vee \mathcal{F}$ is the smallest equivalence that contains $\mathcal{E} \bigcup \mathcal{F}$. For the lattice of *relations* with the same order, $\mathcal{E} \vee \mathcal{F} = \mathcal{E} \bigcup \mathcal{F}$. However, not all the relations are equivalences and transitivity is closed under intersection but not under union. That is, if $x\mathcal{E}y$ and $y\mathcal{F}z$, then we have $x(\mathcal{E} \vee \mathcal{F})z$ but $(x, y) \in \mathcal{E}$ and $(y, z) \in \mathcal{F}$ do not imply $(x, z) \in \mathcal{E} \bigcup \mathcal{F}$. $\mathcal{E} \vee \mathcal{F}$ is exactly the transitive closure of $\mathcal{E} \bigcup \mathcal{F}$.

Note that the join is much harder to express (and compute) than the meet. This will be reflected in the results. Indeed closure for meet will usually boils down to closure under intersection which is a natural thing to require, but closure for join is far from closure under union.

Standard results for Equ($\mathbb{N}$) were laid out in the seminal paper by Ore [Ore42]. It is known that Equ($\mathbb{N}$) is, among others,

- bounded with minimal element $\perp$ being the equivalence where each class is a singleton (no two different elements are equivalent) and the maximal element $\top$ being the equivalence with a single class containing every elements (any two elements are equivalents);
- complete, that is closed under arbitrary join and meet (and not only under finite ones);
- atomistic, each element is the join of atoms, where atoms are successors of $\perp$, that is equivalences with on class containing two elements and the other are singletons;
- relatively complemented (hence complemented), for each $\mathcal{E}$, there exists $\mathcal{F}$ such that $\mathcal{E} \vee \mathcal{F} = \top$ and $\mathcal{E} \wedge \mathcal{F} = \perp$; non $\top$ or $\perp$ equivalences have infinitely many complements, most have uncountably many; we note compl($\mathcal{E}$) the set of complements to $\mathcal{E}$;

Observe that if $\mathbf{E} = \{\mathcal{E}_k\} \subseteq$ Equ($\mathbb{N}$), then $\mathcal{E} = \bigwedge \mathbf{E}$ is the equivalence relation where each class is obtained as $\bigcap_k C_k$ where each C_k is a class of $\mathbf{E}_k$. That is, $m\mathcal{E}n$ iff $m\mathcal{E}_kn$ for all k. Likewise, $\mathcal{E} = \bigvee \mathbf{E}$ is the equivalence defined by $m\mathcal{E}n$ iff $m(\bigcup_k \mathbf{E}_k)^*n$, that is, $m\mathcal{E}n$ iff there exist finite sequences $a_1, \dots, a_{p-1} \in \mathbb{N}$ and $\mathcal{E}_1, \dots, \mathcal{E}_p \in \mathbf{E}$ such that $m\mathcal{E}_1a_1, a_1\mathcal{E}_2a_2, \dots, a_{p-1}\mathcal{E}_pn$.

An equivalence relation $\mathcal{E}$ where exactly one class is not a singleton is called *singular*. When $\mathcal{E}$ is singular, there exists an element e such that for any element x we have either (i) $x\mathcal{E}e$ or (ii) for all $y \neq x, \neg(x\mathcal{E}y)$.

Lemma 313 (Complements to singular equivalence relations)**.** *Let $\mathcal{E}$ be a singular equivalence relation with non-singleton class E. $\mathcal{F}$ is one of its complement iff each class of $\mathcal{F}$ contains exactly one element of E.*

Proof. Suppose that $m, n \in E$ and $m\mathcal{F}n$, that is, there exist a class of $\mathcal{F}$ containing at least two elements m and n of E. In this case, since $m, n \in E$, we have $m\mathcal{E}n$ and, by hypothesis, $m\mathcal{F}n$. Thus, $m(\mathcal{E} \wedge \mathcal{F})n$. However, since $\mathcal{F}$ is a complement of $\mathcal{E}$, the meet is $\perp$ and no two different elements are equivalent. Hence, no class of $\mathcal{F}$ may contain several elements of E.

On the other hand, suppose that there is a class F of $\mathcal{F}$ that contains no element of E. Let f be an element of that class and $e \in E$ be an element of E. If we try to build a finite sequence $f\mathcal{E}a_1\mathcal{F}a_2 \dots a_k\mathcal{F}e$, it is easy to see that a_1 must be f as $f \notin E$, the only non-singleton class of $\mathcal{E}$. Thus, $f\mathcal{F}a_2$ implies that $a_2 \in F$, hence $a_2 \notin E$. It is thus possible to iterate the reasoning and see that such a finite sequence is not possible, that is $\neg(f(\mathcal{E} \vee \mathcal{F})e)$. However, the join must be $\top$ and every two elements are equivalent. Thus, each class of $\mathcal{F}$ must contain at least one element of E.

Conversely if each class of $\mathcal{F}$ contains exactly one element of E, consider two elements $m \neq n$. If $m\mathcal{F}n$, then one of them must be a singleton in $\mathcal{E}$ and $\neg(m\mathcal{E}n)$. Thus, it is not possible to have both $m\mathcal{E}n$ and $m\mathcal{F}n$, hence $\mathcal{E} \wedge \mathcal{F} = \perp$. On the other hand, there exists $m', n' \in E$ such that $m\mathcal{F}m'$ (because each class contains one element of E), $m'\mathcal{E}n'$ and $n'\mathcal{F}n$. Thus, $m(\mathcal{E} \vee \mathcal{F})n$ and $\mathcal{E} \vee \mathcal{F} = \top$. $\mathcal{F}$ is a complement to $\mathcal{E}$. □

2.3 Computability and complexity

We refer to standard textbooks convering computability and complexity theory (*e.g.*, [Jon97]), and only give a short overview of these as they relates to equivalences.

Unless otherwise stated, all Turing Machines are multi-tape machines with two designated read-only input tapes, one write-only output tape, and any number of work tapes. Machines are assumed to be deterministic unless otherwise stated. The reason for having two input tapes is that decidability questions for equivalence relations naturally require two inputs (*i.e.*, to decide whether the inputs are in the same equivalence class or not).

An equivalence relation $\mathcal{E} \in \mathrm{Equ}(\mathbb{N})$ is *decidable* if there exists a Turing Machine M with two input tapes such that for every $(m, n) \in \mathbb{N}$, M halts with output 1 if $m\mathcal{E}n$ and halts with output 0 otherwise, when M is started with the binary representations $\mathtt{m}$ and $\mathtt{n}$ on the first and second input tapes, respectively. It is immediate that $\mathcal{E}$ is decidable iff a Turing machine with a single input tape over alphabet $\{0, 1, \|\}$ decides the language

$$\{\mathtt{m}\|\mathtt{n} : m, n \in \mathbb{N} \text{ with } m\mathcal{E}n\}$$

Similarly, $\mathcal{E}$ is *recursively enumerable* (abbreviated *r.e.*) if there is a two tapes TM that halts on the corresponding input (or equivalently if there is a single-tape Turing machine that halts on input $\mathtt{m}\|\mathtt{n}$ iff $m\mathcal{E}n$).

Note that if $\mathcal{E}$ is recursively enumerable, and we have a specific element n, then we can enumerate all the elements equivalent to it by standard dovetailing techniques.

We note $\mathrm{Equ}(\mathbb{N})_{\text{r.e.}}$ (resp. $\mathrm{Equ}(\mathbb{N})_{\text{dec}}$) the sets of all recursively enumerable equivalences (resp. decidable equivalences).

If the Turing Machine deciding $\mathcal{E}$ does so within some time or space bound (*e.g.* polynomial time), we'll say that $\mathcal{E}$ is *subrecursive*. If $\mathcal{E}$ can be decided by an automaton, we'll say that it is *automatic*. If deciding $\mathcal{E}$ corresponds to a problem in the arithmetical hierarchy, we'll say that it is *superrecursive*. See details about these definitions in the corresponding Sections.

2.4 Results

The paper concerns four broad categories: automatic, subrecursive, recursive, and superrecursive equivalence relations and how they interact with the basic properties of the entire lattice of equivalence relations, namely meet, join and complements. Each of these categories is further subdivided into smaller categories, for instance "subrecursive" is divided into categories correponding to well-known subrecursive classes, such as the set of polynomial-time decidable equivalence relations, or the set of primitive recursive equivalence relations.

We summarise our results informally in the below table. Each row in the table denotes a class of equivalence relations, and each column denotes a closure property that this class may have. Each entry indicates whether the class enjoys the property or not. For example, "finite $\wedge$" means that if $A = \{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ is a finite set of (representations of) equivalence relations in the class, then $\mathcal{E}_1 \wedge \dots \wedge \mathcal{E}_n$ is also an element of the class. The columns under the heading "Arithmetical" denote whether the particular class is closed under taking *meet*/*join* of Σ^0_k sets of equivalence relations from the class. The column "Arbitrary" denotes whether the class is closed under taking *meet*/*join* of equivalence relations from the class (note that as the answer is "No" in each entry in the column, it immediately follows that none of the classes are *complete* lattices).

The entry "N/A" means that the property in question is meaningless or ill-defined (e.g., having complements is meaningless for a class that does not include both $\top$ and $\perp$). The entry ? means that it is an open question whether the class enjoys the property.

	Finite		Arithmetical		Arbitrary	
	$\wedge$	$\vee$	$\wedge$	$\vee$	$\wedge/\vee$	complements
Automatic	Yes	Yes	No	Yes	No/Yes	N/A
Subrecursive	Yes	No[†]	?	No[†]	No	$\geq$ PSPACE
Σ^0_k	Yes	Yes	No	Yes	No	No
Π^0_k	Yes	No	Yes	No	No	?
Δ^0_k	Yes	No	No	No	No	Yes

†: for LOGSPACE or larger classes.

Note that a "Yes" entry in row i and colunmn j does not imply any results concerning actual computation of meet or join–it merely implies that the subset of equivalence relations in row i is closed under the operation of column j–but the actual computation involved in the operation may require resources beyond those implied in row i. If we consider a subset S of all equivalences and an operation on the equivalences (*e.g.* the meet), there are four possibilities:

1. S is not closed under this operation, that is there exist elements in S such that, when applying the operation to them we can obtain an element not in S. *This is the canonical meaning of "No" in the above table.*
2. S is closed under the operation (i.e., there is a "Yes" in the above table), and either
   (a) the operation is computable within the resource requirements defining the class, e.g. polytime computable for the set of polytime decidable equivalence relations.
   (b) the operation is computable, but not within the resource requirements defining the class.
   (c) the operation is not computable.

While our primary concern is merely answering "yes" or "no" to item 2 in the list above, we have endeavoured to answer which of the mutually exclusive subcases 2a–2c holds, wherever possible.

3 Automatic equivalence relations

There are several "natural" ways to define equivalence relations decidable by finite automata. The one we consider here uses a single, two-way input tape with two inputs separated by a special symbol; another natural variation would have the two inputs on two separate tapes, or a single tape but two heads. Unlike Turing machines, the class of sets decidable by finite automata properly increases with the number of input tapes and tape heads; it is therefore quite conceivable that simple variations of the class we consider would have different properties.

We consider two-way finite automata that have only a single tape where both inputs are encoded. We believe the most natural form to be $a\Box b$ where $a, b \in \{0,1\}^+$ and $\Box$ is a separator symbol ("blank").

Definition 314. An equivalence relation $\mathcal{E}$ on $\mathbb{N}$ is said to be *single-tape automatic* if there is a (single-tape) two-way non-deterministic finite automaton that accepts the language $\{\mathtt{m}\Box\mathtt{n} : m\mathcal{E}n\}$. *Par abus de langage* we say that the automaton *accepts* $\mathcal{E}$.

The set of single-tape automatic equivalence relations is denoted by $\mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$.

By standard results, for any two-way NFA, there exists a one-way DFA that accepts the same language. Hence, for any $\mathcal{E} \in \mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$ the language $\{\mathtt{m}\Box\mathtt{n} : m\mathcal{E}n\}$ is regular.

Proposition 315. *Let $\mathcal{E}$ be a single-tape automatic equivalence relation. Then $\mathcal{E}$ has a finite number of equivalence classes.*

The proof uses an argument taken from the Myhill-Nerode Theorem [Ner58].

Proof. Let A be a DFA accepting $\{\mathtt{m}\Box\mathtt{n} : m\mathcal{E}n\}$.

Let m be an integer, we note q_m the state of A reached after reading $\mathtt{m}\Box$.

Let m and n be two integers. If $q_m = q_n$, then because $\mathcal{E}$ is reflexive, $\mathtt{m}\Box\mathtt{m}$ is accepted but because A is deterministic and $q_m = q_n$ then $\mathtt{n}\Box\mathtt{m}$ must lead in the same state and is also accepted. Hence $m\mathcal{E}n$.

Thus, $q_m = q_n \Rightarrow m\mathcal{E}n$ and there cannot be more classes than the number of states of A. $\Box$

Note that Proposition 315 implies that some very simple equivalence relations are not elements of $\mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$; for example, $\bot \notin \mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$ (that is, $\{\mathtt{n}\Box\mathtt{n}\}$ is not a regular language, a well-known fact).

Since the number of classes in any equivalence of $\mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$ is finite but unbounded, given such an equivalence, it is always possible to find a class with several elements and create a new equivalence by taking one element out of this class and making a new singleton class. This new equivalence is smaller than the initial one and is still single-tape automatic.

Hence, $\mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$ is neither bounded, nor complete.

Moreover, let $\mathcal{E}$ and $\mathcal{F}$ be two equivalences, each with finitely many classes $E_1, \dots, E_m$ and $F_1, \dots, F_n$. Since the classes of $\mathcal{E} \wedge \mathcal{F}$ are exactly the non-empty $E_i \bigcap F_j$, there are at most $n \times m$ such classes, that is, a finite number. Thus, a complement to an equivalence with finitely many classes must have infinitely many classes (because their meet is $\bot$, which has infinitely many classes).

Hence, no complement of a single-tape automatic equivalence is single-tape automatic.

Theorem 316. *$Equ(\mathbb{N})_{AUT}$ is a lattice.*

Proof. Let $\mathcal{E}_1, \mathcal{E}_2 \in \mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$ and let A_1 and A_2 be two minimal DFA accepting $\{\mathtt{m}\Box\mathtt{n} : m\mathcal{E}_1 n\}$ and $\{\mathtt{m}\Box\mathtt{n} : m\mathcal{E}_2 n\}$, respectively.

Meet. As $\mathcal{E}_1 \wedge \mathcal{E}_2 = \mathcal{E}_1 \cap \mathcal{E}_{\in}$ we have, for arbitrary $m, n \in \mathbb{N}$, that $m(\mathcal{E}_1 \wedge \mathcal{E}_2)n$ iff $\mathtt{m}\Box\mathtt{n} \in L(A) \cap L(B)$. And as the set of regular languages is closed under intersection, there is an automaton C that accepts $L(A) \cap L(B)$, whence $\mathcal{E}_1 \wedge \mathcal{E}_2 \in \mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$.

Join. First, note that for each m, there is an automaton accepting exactly the n with $m\mathcal{E}_1 n$. Indeed, it is A_1 with input state moved to q_m (with the notation of the previous proof).

By Proposition 315, both $\mathcal{E}_1$ and $\mathcal{E}_2$ have a finite number of equivalence classes. Thus, Let j_1 and j_2 be the number of equivalence classes in $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively, and let $m_1, \dots, m_{j_1}$ and $n_1, \dots, n_{j_2}$ be representatives of the equivalence classes.

For each of these representatives, we can precompute whether $m_k(\mathcal{E}_1 \vee \mathcal{E}_2)n_i$ or not.

Finally, we construct the NFA that follows:

- Start by j_1 ϵ-transitions leading to copies of A_1 with initial state (that is, the end of the ϵ-transition) in q_{m_k}.
- At the accepting states of these copies, put $\Box$-transitions towards sufficiently many copies of A_2 with initial state q_{n_i} for those $n_i(\mathcal{E}_1 \vee \mathcal{E}_2)m_k$.
- If we reach an accepting state of the copies of A_2, accept, otherwise reject.

Thus, the copies of A_1 find the unique m_k such that $m_k\mathcal{E}_1 m$, then from here the $\Box$ transitions start to look into $\mathtt{n}$ and the copies of A_2 find whether there exists n_i with $m_k(\mathcal{E}_1 \vee \mathcal{E}_2)n_i$ (we've only put copies for these n_i) and $n_i\mathcal{E}_2 n$ (that's what the copy is testing).

Finally, we just need to make a DFA equivalent to the NFA we just built. $\Box$

Note that the construction in the proof is actually constructive. Indeed, because the automata are minimal, there exists a word leading to each state and it is easy to find it by standard search in a graph with loops. Thus we can actually find a representative of each class of the equivalence. Next, the precomputation of the links between the m_k and n_i is simply a matter of $2 \times j_1 \times j_2$ initial tests plus some closure algorithm on the result, that is finding connected components in the graph of $j_1 + j_2$ vertices with edge if either $m\mathcal{E}_1 n$ or $m\mathcal{E}_2 n$.

Thus, not only is $\mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$ a lattice, but both the meets and the joins are effectively computable. Computing these, however, cannot be performed by a DFA (this is case 2b of the discussion after the results Table).

While $\mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$ is closed under finite meet and join, it is not closed under infinite meet. This follows *a fortiori* from the following result.

Proposition 317. *There is an r.e. set $A = \{ A_k \mid k \in \mathbb{N} \}$ of deterministic, single-tape finite automata, each of which accepts some $\mathcal{E}_k \in Equ(\mathbb{N})_{AUT}$, but such that $\mathcal{E} = \bigwedge_{k\in\mathbb{N}} \mathcal{E}_k \notin Equ(\mathbb{N})_{AUT}$.*

Proof. Let $B = 0|(1\{0,1\}^*)$ be the set of binary representations of natural numbers; it is a regular language. Let $L_k = \{ x \in B \mid |x| \leq k \}$ be the set of binary representations of length k or less; it is a regular language. Let $F_k = \{ a\Box b \mid a, b \in L_k \}$ and $F'_k = \{ a\Box b \mid a, b \notin L_k \}$; they are both regular languages. Lastly, let $E_k = F_k \bigcup F'_k$, and let $\mathcal{E}_k$ be the following relation $m\mathcal{E}_k n \Leftrightarrow \mathtt{m}\Box\mathtt{n} \in E_k$. By construction, $\mathcal{E}_k$ is an automatic relation. Moreover, given k, it is possible to build an automaton A_k deciding $\mathcal{E}_k$, hence the set $\{ A_k \mid k \in \mathbb{N} \}$ is recursively enumerable.

$\mathcal{E}_k$ is an equivalence relation. Indeed, it is the equivalence relation having two classes, one with all numbers whose binary representation has length k or less (is in L_k) and one with all numbers whose binary representation has length strictly more than k (is not in L_k).

Let $m, n \in \mathbb{N}$ with $|\mathtt{m}| = i$ and $|\mathtt{n}| = j$. If $i < j$ then we have $\neg(m\mathcal{E}_i n)$. Hence, if $\mathtt{m}$ and $\mathtt{n}$ have different lengths, there exists k such that m and n are not equivalent in $\mathcal{E}_k$.

Now, let $\mathcal{E} = \bigwedge \mathcal{E}_k$. We have $m\mathcal{E}n$ iff for all k, $m\mathcal{E}_k n$, that is iff $\mathtt{m}$ and $\mathtt{n}$ have the same length. Hence, $\mathcal{E}$ has infinitely many classes and cannot be automatic. $\Box$

However, $\mathrm{Equ}(\mathbb{N})_{\mathrm{AUT}}$ is closed under arbitrary infinite join.

Proposition 318. *Let $A \subseteq Equ(\mathbb{N})_{AUT}$ be non-empty. Then, $\bigvee A \in Equ(\mathbb{N})_{AUT}$.*

Proof. Every equivalence class of $\bigvee A$ can be obtained as a union of equivalence classes of any element $\mathcal{E} \in A$. By Proposition 315, every such $\mathcal{E}$ has a finite number of classes, hence so does $\bigvee A$, and any class in $\bigvee A$ can be obtained as a *finite* union of classes of $\mathcal{E}$, hence can be recognized by a deterministic, one-tape finite automaton by standard constructions; as $\bigvee A$ has a finite number of classes, the result follows. $\square$

As mentioned at the beginning of the Section, the expressive power of automata varies with the number of tapes or heads. Hence, small variations of the model can drastically change the property of the corresponding subset of equivalences. For example, two-tape automata can test whether both tape contain the same word and thus decide $\perp$.

Even the choice of representation of inputs may affect the expressive power. For example, instead of sequencing the inputs ($\mathtt{m}\square\mathtt{n}$), it is possible to interleave them ($m_1n_1m_2n_2\dots m_in_i\dots\square n_k$ if n is longer than m). This representation allows an automaton to decide $\perp$ (note that the Myhill-Nerode argument does not work in this case).

Moreover, when working with multi-tape automata, the question of synchronicity arises: should the tapes be "at the same position" on each tape, or can the heads move independently?

Thus, we have only studied one particular class of automata with one particular way of representing equivalences. $\mathrm{Equ}(\mathbb{N})_{\textsc{Aut}}$ hence enjoys properties that other classes of "automatic" relation may or may not have. While the class of single-tape, single-head DFAs studied above is the simplest kind of finite automaton–hence most apt to study first–it remains to perform a systematic study of equivalence relations decidable by other, more particular, kinds of automata. We leave this for future work.

4 Subrecursive equivalence relations

We now treat classes of equivalence relations decidable within bounds on their resources. The definition of such classes are simply the standard definitions of computational complexity theory using Turing machines with two input tapes. The following section establishes terminology.

4.1 Subrecursive classes

A *complexity bound* is a pair (S, res) where S is a set of functions $f : \mathbb{N} \longrightarrow \mathbb{N}$ and res is some computational resource, for example space or time. If (S, res) is a complexity bound, a subset $A \subseteq \{0,1\}^*$ is said to be decidable within (S, res) if there is a (multitape) Turing machine M deciding A such that there is a function $f \in S$ where M uses resources res bounded above by f. We denote by S-RES the set of subsets of $\{0,1\}^*$ decidable within the bounds specified by (S, res). If $\mathcal{E}$ is an equivalence relation on $\mathbb{N}$, we denote by $\mathrm{Equ}(\mathbb{N})_{\text{S-RES}}$ the set of equivalence relations $\mathcal{E}$ decidable by (two-input-tape, multiple work-tape) Turing machines deciding $\mathcal{E}$ within the bounds specified by (S, res).

The same concept is defined for non-deterministic machines *mutatis mutandis*, *i.e.* a set is decidable within resources in (S, nres) there is a non-deterministic Turing machine that accepts $\mathcal{E}$ such that M operates with the bounds specified by (S, nres). The sets S-NRES and $\mathrm{Equ}(\mathbb{N})_{\text{S-NRES}}$ are defined as expected.

Example 319. For example, if $S = \mathrm{poly}$ is the set of polynomials over the naturals, S-RES = POLY-TIME is the set polynomial-time decidable sets, *i.e.* S-RES = POLY-TIME = PTIME. Similarly, $\mathrm{Equ}(\mathbb{N})_{\text{S-RES}}$ is the set of equivalence relations decided by a polynomial-time TM. That is there exists a two-tape TM that on inputs $\mathtt{m}$ and $\mathtt{n}$ halts in time bounded by a polynomial in the size of the inputs and accepts if and only if m and n are equivalent.

When o is an operation on functions from natural numbers to natural numbers, we say that S is *closed under o* if $\forall f_1, \dots, f_n \in S. o(f_1, \dots, f_n) \in S$. For example, the set of polynomials with positive integer coefficients is closed under addition and multiplication.

The sets of equivalence relations on $\mathbb{N}$ consisting of primitive recursive, EXPTIME-, PSPACE-, PTIME-, and LOGSPACE-decidable equivalence relations are denoted by $\mathrm{Equ}(\mathbb{N})_{\mathrm{p.r.}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{ExpTime}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{Pspace}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{Ptime}}$, and $\mathrm{Equ}(\mathbb{N})_{\textsc{LogSpace}}$, respectively. The notation is extended to other sets in straightforward ways later in the text.

4.2 Finite meet and join

Lemma 320. *Let $\mathcal{E}$ be decidable in time $T(\mathcal{E})$ (resp. space $S(\mathcal{E})$) and $\mathcal{F}$ be decidable in time $T(\mathcal{F})$ (resp. space $S(\mathcal{F})$). Then $\mathcal{E} \wedge \mathcal{F}$ is decidable in time $T(\mathcal{E}) + T(\mathcal{F})$ (resp. space $\max(S(\mathcal{E}), S(\mathcal{F}))$).*

Proof. $m(\mathcal{E} \wedge \mathcal{F})n$ iff $m\mathcal{E}n$ and $m\mathcal{F}n$.

Hence to check whether $m(\mathcal{E} \wedge \mathcal{F})n$, it is sufficient to first check whether $m\mathcal{E}n$ and then check whether $m\mathcal{F}n$. Space can be reused, hence only the maximum counts, but time adds. $\square$

Corollary 321. *Let S be a time bound closed under addition, or a space bound. The set of S-decidable equivalence relations is closed under finite meet.*

Especially, $\mathrm{Equ}(\mathbb{N})_{\mathrm{r.e.}}$, $\mathrm{Equ}(\mathbb{N})_{\mathrm{dec}}$, $\mathrm{Equ}(\mathbb{N})_{\mathrm{p.r.}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{ExpTime}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{Pspace}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{Ptime}}$, and $\mathrm{Equ}(\mathbb{N})_{\textsc{LogSpace}}$ are all closed under finite meet (they form lower subsemilattices).

Proposition 322. *There exist two equivalence relations, both of which are decidable in logarithmic space, but whose join is undecidable.*

Proof. Let M be a deterministic Turing machine. We may assume wlog. that the machine is not stuck on any of its configurations unless the state in the configuration is either "accept" or "reject".

let c, c' be any two of the configurations of M, and let $\mathtt{c}, \mathtt{c}'$ be the binary representations of these under some standard one-to-one encoding. We assume wlog. that each representation starts with 1. Note in particular that no configuration is represented by the string "0" and that the representation of any configuration is also a representation of some positive integer.

We now define a partial binary relation $\rightarrow \subseteq \{0,1\}^{+} \times \{0,1\}^{+}$ as follows: Write $\mathtt{c} \rightarrow \mathtt{c}'$ if c and c' are configurations of M and c' is the result of executing one step of M from configuration c. Moreover, we *define* $\mathtt{c} \rightarrow 0$ whenever c is a final configuration of M (*i.e.*, c is in an accept or reject state).

The relation $\rightarrow$ can be decided in logarithmic space in the size of $\mathtt{c}$ and $\mathtt{c}'$ by standard techniques: If M is a Turing machine and configurations d are on the form $\mathtt{d} = (\text{state}, \text{tape content}, \text{tape head position})$, there is a Turing machine R_M that on input $(\mathtt{c}, \mathtt{c}')$ decides whether $\mathtt{c} \rightarrow \mathtt{c}'$ using a fixed number of counters to (a) keep track of where in each of $\mathtt{c}$ and $\mathtt{c}'$ the tape heads of R_M are currently reading[2] and (b) to check whether the tape contents of $\mathtt{c}$ and $\mathtt{c}'$ are identical except for, possibly, at the tape head positions in $\mathtt{c}$ and $\mathtt{c}'$. R_M carries a copy of M in its internal logic as a lookup table and simply performs a lookup to see whether there is a transition in M that would allow the change observed at the tape head positions in $\mathtt{c}$ and $\mathtt{c}'$ (and checks whether the tape positions differ by at most one).

We denote by $\mathtt{c} \approx \mathtt{c}'$ the reflexive transitive symmetric closure of $\rightarrow$. Clearly, $\approx$ is undecidable as $\mathtt{c} \approx 0$ iff the execution starting in configuration c terminates.

Now, given a non-negative integer n whose binary representation is $\mathtt{n}$ and a configuration c, we build the couple (n, c) as the base-3 number $(\mathtt{n}, \mathtt{c}) = \mathtt{n2c}$. We define the *clocked one-step relation* $\rightarrow'$ on $\{0,1\}^{+}$ by $(\mathtt{n}, \mathtt{c}) \rightarrow' (\mathtt{m}, \mathtt{c}')$ iff either $m = n + 1$ and $\mathtt{c} \rightarrow \mathtt{c}'$ or if c is final and $(\mathtt{m}, \mathtt{c}') = 020$ ($= 6$, in base-3). Note that as $\rightarrow$ is decidable in logarithmic space in the size of $\mathtt{c}$ and $\mathtt{c}'$, and as it can be checked in logarithmic space whether $m = n + 1$ and in constant space whether $(\mathtt{m}, \mathtt{c}') = 020$, then $\rightarrow'$ is decidable in logarithmic space in $(\mathtt{n}, \mathtt{c})$ and $(\mathtt{m}, \mathtt{c}')$. We define $\approx'$ as the reflexive transitive symmetric closure of $\rightarrow'$. Now, $\approx'$ is an *un*decidable equivalence relation as $(\mathtt{n}, \mathtt{c}) \approx' 020$ iff the execution starting at c terminates.

We now define two further equivalence relations $\rightarrow_{\mathrm{even}}$ and $\rightarrow_{\mathrm{odd}}$ as follows: $(\mathtt{n}, \mathtt{c}) \rightarrow_{\mathrm{even}} (\mathtt{m}, \mathtt{c}')$ (resp. $(\mathtt{n}, \mathtt{c}) \rightarrow_{\mathrm{odd}} (\mathtt{m}, \mathtt{c}')$) if either (i) $(\mathtt{n}, \mathtt{c}) \rightarrow' (\mathtt{m}, \mathtt{c}')$ and n is even (resp. odd) or (ii) c is final, n is even (resp. odd) and $(\mathtt{m}, \mathtt{c}') = 020$. It is obvious that $\rightarrow_{\mathrm{even}}$ and $\rightarrow_{\mathrm{odd}}$ are still decidable in logarithmic space.

Note that by construction, if $(\mathtt{n}, \mathtt{c}) \rightarrow_{\mathrm{even}} (\mathtt{m}, \mathtt{c}')$ then n must be even, hence $m = n + 1$ must be odd (except for the special case of final configuration). Hence, there is no $(\mathtt{p}, \mathtt{c}'')$ such that $(\mathtt{m}, \mathtt{c}') \rightarrow_{\mathrm{even}} (\mathtt{p}, \mathtt{c}'')$.

Now, consider $\approx_{\mathrm{even}}$, the reflexive transitive symmetric closure of $\rightarrow_{\mathrm{even}}$. We claim that $\approx_{\mathrm{even}}$ is decidable in logarithmic space. Indeed, the only cases where we can have $x \approx_{\mathrm{even}} y$ are:

- $x = y$.
- $x \rightarrow_{\mathrm{even}} y$, this is decidable in logarithmic space.
- $y \rightarrow_{\mathrm{even}} x$, this is decidable in logarithmic space.
- There exists z such that $x \rightarrow_{\mathrm{even}} z$ and $y \rightarrow_{\mathrm{even}} z$, this is decidable in logarithmic space by simulating one step of the execution on the configurations in x and y and checking whether the results are the same. As the Turing macine M is deterministic, at most one step per configuration must be simulated.

In the last case, we cannot directly simulate one step of the machine from each of the configurations x and y and compare the results in logarithmic space (because we cannot store the results). However, we can check that x and y only differ by the state of the machine and the symbol read by the tape head; then, the step can be performed by changing only local information, i.e., compute the new state, tape symbol, and move. All of these can clearly be done in logarithmic space.

As it is not possible to have $x \rightarrow_{\mathrm{even}} z \rightarrow_{\mathrm{even}} y$, it is never necessary to simulate several steps in a row in order to check whether $x \approx_{\mathrm{even}} y$. Thus, *at most one* step from each of x and y must be simulated. Hence, $\approx_{\mathrm{even}}$ is decidable in logarithmic space. Similarly, $\approx_{\mathrm{odd}}$ is decidable in logarithmic space. However, $\approx_{\mathrm{even}} \vee \approx_{\mathrm{odd}} = \approx'$ is not decidable. $\square$

Corollary 323. *None of $Equ(\mathbb{N})_{\mathrm{dec}}$, $Equ(\mathbb{N})_{\mathrm{p.r.}}$, $Equ(\mathbb{N})_{\textsc{ExpTime}}$, $Equ(\mathbb{N})_{\textsc{Pspace}}$, $Equ(\mathbb{N})_{\textsc{Ptime}}$, or $Equ(\mathbb{N})_{\textsc{LogSpace}}$ are closed under finite join. Hence none of these sets is a sublattice of $Equ(\mathbb{N})$.*

[2]Note that $\mathtt{c}$ does contain the whole tape, of size N and the head position in it. Because the head must point into the tape, the head position is at most N and thus can be stored in binary using only $\log N$ space. Therefore, counting up to N to actually find the position in the tape requires space $\log N$ (which is indeed logarithmic in the size of $\mathtt{c}$).

4.3 Complements

Even if $\mathrm{Equ}(\mathbb{N})_{\textsc{Ptime}}$ and such are not sublattices, as they are not closed under join, we may consider complements: each element of $\mathrm{Equ}(\mathbb{N})$ (except $\top$ and $\bot$) has infinitely many complements, and in some cases may have uncountably many [AMRS17]. On the other hand, $\mathrm{Equ}(\mathbb{N})_{\textsc{Ptime}}$ and any other set of equivalence relations where every relation must be decided by a Turing machine, must necessarily be countable, as there are only countably many Turing machines. Hence, we can in general not hope to encompass all complements of an equivalence relation in a single set among those we consider.

However, we can show that any equivalence relation in, say, $\mathrm{Equ}(\mathbb{N})_{\textsc{ExpTime}}$ has *some* of its complements in $\mathrm{Equ}(\mathbb{N})_{\textsc{ExpTime}}$.

A note on nomenclature: The ordered sets we consider are not necessarily sublattices (*e.g.*, $\mathrm{Equ}(\mathbb{N})_{\textsc{ExpTime}}$ is demonstrably not a sublattice), whence we cannot use the term "complemented". This motivates the following definition

Definition 324. Let $\mathcal{A} \subset \mathrm{Equ}(\mathbb{N})$ be a set of equivalence relations, not necessarily a sublattice. We say that $\mathcal{A}$ *always admits complements* if for any element $s \in \mathcal{A}$, $\mathcal{A}$ contains at least one complement to s (in $\mathrm{Equ}(\mathbb{N})$), that is $\forall s \in \mathcal{A}, \mathcal{A} \bigcap \mathrm{compl}(s) \neq \emptyset$.

Theorem 325. *Let (S, res) be a complexity bound where* res *is either (deterministic or non-deterministic) space or time. Assume that (i) S is closed under taking integer polynomials (i.e., if $P(x) \in \mathbb{N}[X]$ and $f \in S$, then $P(f) \in S$), and (ii) S contains a function f such that $n < f(n)$ for all n, (iii) S-*RES *is closed under complement, and (iv) if res is time, then S contains $n \mapsto 2^n$. Then, $Equ(\mathbb{N})_{S\text{-}\textsc{res}}$ always admit complements.*

Proof. Let $\mathcal{E} \in \mathrm{Equ}(\mathbb{N})_{\mathrm{S}\text{-}\textsc{res}}$ be non-trivial. Let $\mathcal{F}$ be the singular equivalence relation whose unique non-singleton class, F, contains exactly the least element of each class of $\mathcal{E}$.

Complement. $\mathcal{F}$ is a singular partition and, by Lemma 313, $\mathcal{E}$ is one of its complements.

Least element. Consider two natural numbers $m \neq n$. If there exists $k < m$ such that $k\mathcal{E}m$, then by construction $m \notin F$ which implies $\neg(m\mathcal{F}n)$. Similarly, if there exists $k' < n$ such that $k'\mathcal{E}n$ then $\neg(m\mathcal{F}n)$. Conversely, if there exists no such k (resp. k') then m (resp. n) is the least element of its class of $\mathcal{E}$ and is in F. Thus $m\mathcal{F}n$ iff no such k and k' exist.

Bound. We may transform any Turing machine M deciding an equivalence relation within some bound $f \in S$ into a one-input tape Turing machine that decides the set $\{\mathtt{m}\Box\mathtt{n} : m\mathcal{E}n\}$ within bound $f + O(m+n)$ (both if res is time or space): The machine merely copies $\mathtt{m}$ and $\mathtt{n}$ to two new auxiliary worktapes and executes M as a subroutine where the two new worktapes are regarded as input tapes. It takes linear time to perform the copying, and linear space to store the extra inputs. Hence, as S contains a function f such that $n < f(n)$ for all n and is S closed under taking polynomials, we obtain $\{\mathtt{m}\Box\mathtt{n} : m\mathcal{E}n\} \in \mathrm{S}\text{-}\textsc{res}$. As S-RES is closed under complement, we have $\{\mathtt{m}\Box\mathtt{n} : \neg(m\mathcal{E}n)\} \in \mathrm{S}\text{-}\textsc{res}$, and we may thus let $\overline{M_{\mathcal{E}}}$ be a Turing machine witnessing that fact.

Consider the TM $M_{\mathcal{F}}$ which, on inputs $\mathtt{m}$ and $\mathtt{n}$ proceeds as follow:

1. If $\mathtt{m} = \mathtt{n}$, accept. This takes zero space and linear time.

2. If $\overline{M_{\mathcal{E}}}$ accepts all inputs $\mathtt{k}$ and $\mathtt{m}$ for $k < m$ go on, otherwise reject. This requires m calls to $\overline{M_{\mathcal{E}}}$ and as S is closed under polynomials, the total resource use needed to perform all calls is majorized by a function in S. The bookkeeping of the iteration requires storing the values of $\mathtt{k}$ which have size at most $|\mathtt{m}|$, hence linear space, but requires linear time in m to process in each iteration. As there are at most m iterations and as S is closed under polynomials, it follows that if res is space, the resource usage is majorized by a function in S. If res is time, the fact that S contains $n \mapsto 2^n$ (and $m = 2^{|\mathtt{m}|}$), and is closed under polynomials, implies that the resource usage is majorized by a function in S.

3. Similarly, check whether $\overline{M_{\mathcal{E}}}$ accepts all inputs $\mathtt{k}'$ and $\mathtt{n}$ for $k' < n$.

4. If the input was not rejected yet, accept it.

This machine decides $\mathcal{F}$ and operates within bound S. □

Observe that the methods employed in the proof of Theorem 325 do not work for S-RES = LOGSPACE because the counter k must index elements up to m, requiring space $O(\log m) = O(|\mathtt{m}|)$, *i.e.*, memory linear in the input size; nor for S-RES = PTIME because there are $m + n$ calls to $\overline{M_{\mathcal{E}}}$ (or $\max(m, n)$ with a more clever implementation), thus the time needed is $O(\max(m, n))$ which is *exponential* in $|\mathtt{m}| = O(\log m)$.

It is tempting to conjecture that $\mathrm{Equ}(\mathbb{N})_{\textsc{Ptime}}$ does not always admit complements. However if that were the case, we would have $\mathrm{Equ}(\mathbb{N})_{\textsc{Ptime}} \neq \mathrm{Equ}(\mathbb{N})_{\textsc{Pspace}}$, and by using a polynomial-time pairing function $\mathbb{N}^2 \longrightarrow \mathbb{N}$, existence of $\mathcal{P} \in \mathrm{Equ}(\mathbb{N})_{\textsc{Pspace}} \setminus \mathrm{Equ}(\mathbb{N})_{\textsc{Ptime}}$ entails existence of a set in PSPACE \ P, and hence PSPACE ≠ P; hence, the conjecture will be exceedingly hard to prove.

Corollary 326. *$Equ(\mathbb{N})_{\textsc{Pspace}}$, $Equ(\mathbb{N})_{\textsc{ExpTime}}$, $Equ(\mathbb{N})_{\mathrm{p.r.}}$, and $Equ(\mathbb{N})_{\mathrm{dec}}$ all always admit complements.*

5 Superrecursive equivalence relations

We now investigate closure properties under *meet* and *join* of sets A of equivalence relations where A is an arithmetical sets of relations all of which are also arithmetical sets.

5.1 The arithmetical hierarchy

We briefly recall basic definitions and results for the (Kleene-Mostowski) Arithmetical Hierarchy.

Definition 327. A n-ary relation R is *decidable* if there exists a n-tape TM that always halts and which, on inputs $\mathtt{a}_1, \dots, \mathtt{a}_\mathtt{n}$ accepts if and only if $R(a_1, \dots, a_n)$.

We note $\Sigma^0_0 = \Pi^0_0$ the set of decidable relations.

For $k \in \mathbb{N}$, we define:

- $R \in \Sigma^0_{k+1}$ iff there exists $R' \in \Pi^0_k$ such that $R(a_1, \dots, a_n) \Leftrightarrow \exists r.R'(a_1, \dots, a_n, r)$. r is called a witness.
- $R \in \Pi^0_{k+1}$ iff there exists $R' \in \Sigma^0_k$ such that $R(a_1, \dots, a_n) \Leftrightarrow \forall r.R'(a_1, \dots, a_n, r)$.
- $\Delta^0_k = \Sigma^0_k \bigcap \Pi^0_k$.

Note that Σ^0_1 is the set of r.e. relations, Π^0_1 is the co-r.e. relations and thus $\Delta^0_0 = \Delta^0_1$ is the decidable relations.

Moreover, $\Pi^0_k = \text{co-}\Sigma^0_k$. Hence, Δ^0_k is closed under complement for each k.

Alternating quantifiers A standard expansion of the definitions shows that the Σ^0_k relations have the shape $\exists r_1 \forall r_2 \dots Q_k r_k R(a_1, \dots, a_n, r_1, \dots, r_k)$ where R is decidable (and Q_k is either $\exists$ or $\forall$ depending on the parity of k). Similarly, Π^0_k is expressed by an alternating chain of quantifiers starting with $\forall$.

Tuples By standard encoding of tuples, there may be several numbers (witnesses) in each quantifier, that is R is Σ^0_{k+1} iff there exists a Π^0_k relation R' such that $R(a_1, \dots, a_n) \Leftrightarrow \exists r_1, \dots, r_m.R'(a_1, \dots, a_n, r_1, \dots, r_m)$ (and similarly for $R \in \Pi^0_{k+1}$). As an immediate consequence, if R' is Σ^0_k and $R(a_1, \dots, a_n) \Leftrightarrow \exists r.R'(a_1, \dots, a_n, r)$, then R is also Σ^0_k (and similarly for Π^0_k).

Dependant tuples Because encoding and decoding are decidable (or rather because we can choose a decidable encoding), the number of elements in such tuples can actually depends on a variable quantified at the same time. That is, for example, if $(R_k)_{k\in\mathbb{N}}$ is a family of decidable relations of arity k, then the relation "$\exists n, A.A$ is the encoding of $(a_1, \dots, a_n)$ and $R_n(a_1, \dots, a_n)$" is Σ^0_1. We will abusively write such quantifications as $\exists n, a_1, \dots, a_n.R_n(a_1, \dots, a_n)$ for the sake of clarity.

Conjunction, disjunction To avoid confusion with the join and meet, we denote the logical conjunction and disjunction by && and ||. All these classes are closed under (finite) conjunction and disjunction. That is, for example, if both $R_1(a_1, \dots, a_n)$ and $R_2(b_1, \dots, b_m)$ are Π^0_k then so is

$$R(a_1, \dots, a_n, b_1, \dots, b_m) \Leftrightarrow R_1(a_1, \dots, a_n) \mathbin{\&\&} R_2(b_1, \dots, b_m)$$

As an immediate consequence, we have that the classes of Σ^0_k (resp. Π^0_k, Δ^0_k) equivalences are closed under finite meet.

Separation A straightforward consequence of Post's Theorem (see [Rog67, §14.5]) is that, for each $k \geq 1$, $\Sigma^0_k \backslash \Pi^0_k \neq \emptyset$, $\Pi^0_k \backslash \Sigma^0_k \neq \emptyset$ (the existential and universal sets are separated), and for $k \in \mathbb{N}$ that $\Sigma^0_k \subsetneq \Sigma^0_{k+1}$, respectively $\Pi^0_k \subsetneq \Pi^0_{k+1}$ (the hierarchy is separated).

5.2 Joins

Lemma 328. *For every $k \geq 1$, the set of equivalence relations in Σ^0_k is closed under finite join.*

Proof. Let x and y be two integers. By definition, $x(\mathcal{E} \vee \mathcal{F})y$ iff there exists $a_1, \dots, a_n$ such that $x\mathcal{E}a_1\mathcal{F}\dots\mathcal{E}a_n\mathcal{F}y$. That is

$$\exists n, a_1, \dots, a_n.x\mathcal{E}a_1 \mathbin{\&\&} a_1\mathcal{F}a_2 \mathbin{\&\&} \dots \mathbin{\&\&} a_n\mathcal{F}y$$

Since both $\mathcal{E}$ and $\mathcal{F}$ are Σ^0_k, then so is the conjunction. Since adding existential quantifiers in front of a Σ^0_k relation does not change its level in the hierarchy, then $\mathcal{E} \vee \mathcal{F}$ is Σ^0_k. □

Note that k, as well as any of the a_i, are not bounded and may be as large as wanted. Thus, the construction does not work for any time or space bound.

Even with decidable equivalence relations (but no bound), the construction does not work as it is not possible to guess a number of unbounded size and still guarantee termination. A non-deterministic r.e. machine may guess an unbounded number by alternating choices between "0 or 1?" and "stop or continue?" but this procedure may potentially loop infinitely. A machine that always terminate may only guess a number of bounded size.

This is exactly the same reason why one can semi-decide the halting problem by first guessing the number of steps, then guessing the intermediate configurations and lastly check the individual transitions, but this procedure is not doable with an always terminating machine.

Lemma 329. *The join of any two Π^0_k equivalence relations is in Σ^0_{k+1}.*

Proof. Immediate by unfolding the definition of $\vee$ as above. □

Similar, we have the following generalization of Proposition 322.

Proposition 330. *For any $k \geq 1$, the set of equivalence relations in Δ^0_k is not closed under finite join.*

The proof is essentially the same as for Proposition 322 but much simpler as the entirety of the discussion concerning logarithmic space can be omitted. However, a different upper bound is needed, as the halting problem no longer suffices.

Definition 331. Let A be a Δ^0_k -complete language. An *oracle machine* over A is a multi-tape Turing Machine with a special *oracle* tape and *query* state $q_?$ and *answer* states q_y and q_n such that whenever the machine is in $q_?$ it goes to q_y if the content of the oracle tape is in A and to q_n otherwise.

Fact 4. There is no oracle machine over A that can solve the halting problem for oracle machines over A.

This is a well-known fact whose proof is an adaptation of Turing's proof for standard TMs.

First, note that universal oracle machines do exist because the universal machine may access its own oracle whenever it needs to simulate an oracle query.

Proof. Suppose that H is an oracle machine over A able to solve the halting problem for oracle machines over A. That is, H always halts and it accepts `M#n` iff `M` is the encoding of an oracle machine that terminates on input `n`.

Now, using an universal machine, build the *anti-diagonal* machine D such that D terminates on input `X` iff H rejects `X#X`.

It is immediate that D terminates on input `D` iff H rejects `D#D`, that is iff `D` does not terminates on inupt `D`, an absurdity. □

The rest of the proof here is a shorter version of the proof of Proposition 322 and the reader may want to refer to details there that are used similarly (replacing "decidable in logarithmic space" by "decidable by an oracle machine").

Proof of Proposition 330. Given an oracle machine M and the binary representations of two configurations, the $\rightarrow$ relation, corresponding to simulating one step of M, is decidable by an oracle machine (using its own oracle if it needs to simulate a step from $q_?$). However, the reflexive, transitive, symmetric closure of $\rightarrow$, $\approx$, is not decidable by an oracle machine as $\texttt{c} \approx 0$ is the halting problem for oracle machines.

We define the *clocked one-step relation* $\rightarrow'$ as previously. Again, $\rightarrow'$ is decidable by an oracle machine but its reflexive, transitive, symmetric closure $\approx'$ is not.

Depending on the parity of the clock, we define the two relations $\rightarrow_{\text{even}}$ and $\rightarrow_{\text{odd}}$. They are both decidable by oracle machines, and their reflexive, transitive, symmetric closures $\approx_{\text{even}}$ and $\approx_{\text{odd}}$ also are because it is not possible to have several steps with the same parity in a row.

However $\approx_{\text{even}} \vee \approx_{\text{odd}} = \approx'$ is not decidable by an oracle machine. □

Proposition 332. *For every $n \geq 1$, the set of equivalence relations in Π^0_n is not closed under finite join.*

Proof. Assume, for contradiction, that the join of every pair of Π^0_k equivalence relations were Π^0_k. Consider any two Δ^0_k equivalence relations. A fortiori, both of these relations are both Σ^0_k and Π^0_k, and thus by Lemma 328, the join of the two relations is Σ^0_k, and by the assumption above also Π^0_k, hence Δ^0_k.

But by Proposition 330, Δ^0_k is not closed by join. □

5.3 Complements

Theorem 333. *For every $k \geq 1$, there are equivalence relations in Σ^0_k none of whose complements are in Σ^0_k.*

Proof. Let, by Post's theorem, P be a unary relation in Σ^0_k whose complement is not in Σ^0_k. Thus, there exists a Π^0_{k-1} binary relation P' such that $P(x) \Leftrightarrow \exists r.P'(x,r)$.

Define $E = \{x \mid P(x)\}$ and let $\mathcal{E}$ be the singular equivalence with non-singleton class E. It is in Σ^0_k because $x\mathcal{E}y \Leftrightarrow x = y \,||\, (P(x) \,\&\&\, P(y)) \Leftrightarrow \exists r_1, r_2.x = y \,||\, (P'(x, r_1) \,\&\&\, P'(y, r_2))$.

Let $\mathcal{F}$ be a complement to $\mathcal{E}$. We have $x \in \overline{E}$ iff $\exists e.e \neq x \,\&\&\, e \in E \,\&\&\, e\mathcal{F}x$. Indeed such an e exists because each class of $\mathcal{F}$ must intersect E and because e is the only element in its class (of $\mathcal{F}$) which is also in E, then $x \notin E$. This relation is equivalent to $\exists e, r.e \neq x \,\&\&\, P'(e, r) \,\&\&\, e\mathcal{F}x$.

Suppose that $\mathcal{F}$ is Σ^0_k, then there exists a Π^0_{k-1} relation $\mathcal{F}'$ such that $x\mathcal{F}y \Leftrightarrow \exists r.\mathcal{F}'(x, y, r)$. But now we have $x \in \overline{E} \Leftrightarrow \exists e, r_1, r_2.e \neq x \,\&\&\, P'(e, r_1) \,\&\&\, \mathcal{F}'(e, x, r_2)$ is also Σ^0_k which contradicts the hypothesis.

Hence $\mathcal{F}$ is not Σ^0_k. □

Note that here we need to "go down a level" in the hierarchy to have the proof working.

We next prove a generalization of Theorem 325.

Theorem 334. *Let $k \geq 0$. Every equivalence relation $\mathcal{E}$ in Δ^0_k has at least one complement in Δ^0_k.*

Proof. If $\mathcal{E} \in \{\bot, \top\}$, we have that both $\mathcal{E}$ and its complement are in Δ^0_0, and as $\Delta^0_0 \subseteq \Delta^0_k$, the theorem holds. Further, as $\Delta^0_0 = \Delta^0_1$, we wlog. assume in the remainder of the proof that $k \geq 1$.

Accordingly, let $\mathcal{E}$ be an equivalence relation that is neither $\bot$ nor $\top$. Let $\mathcal{F}$ be the singular equivalence relation whose unique non-singleton class, F, contains exactly the least element of each class of $\mathcal{E}$.

Complement. $\mathcal{F}$ is a singular partition and, by Lemma 313, $\mathcal{E}$ is one of its complements.

Least element. Consider two elements $m \neq n$. If there exist $i < m$ such that $i\mathcal{E}m$, then by construction $m \notin F$ which implies $\neg(m\mathcal{F}n)$. Similarly, if there exists $i' < n$ such that $i'\mathcal{E}n$ then $\neg(m\mathcal{F}n)$. Conversely, if there exists no such i (resp. i') then m (resp. n) is the least element of its class of $\mathcal{E}$ and is in F. Thus $m\mathcal{F}n$ iff no such i and i' exist.

Bound. Since $\mathcal{E}$ is in Δ^0_k, then so is its complement $\overline{\mathcal{E}}$: $m\overline{\mathcal{E}}n \Leftrightarrow \neg m\mathcal{E}n$. Note that $\overline{\mathcal{E}}$ is not an equivalence.

Now, we have:

$$m\mathcal{F}n \Leftrightarrow m = n \mid\mid \left(0\overline{\mathcal{E}}m \;\&\&\ldots\&\&\; (m-1)\overline{\mathcal{E}}m \;\&\&\; 0\overline{\mathcal{E}}n \;\&\&\ldots\&\&\; (n-1)\overline{\mathcal{E}}n\right)$$

Since all the relations involved are Δ^0_k (and the combination of them on the right hand side is primitive recursive), then so is $\mathcal{F}$. □

6 Arbitrary meet and join

First, note that relations in the arithmetical hierarchy can be *specialised*[3] by fixing some of their free variables. That is, if $R_1(a_1, \ldots, a_n)$ is, say, Π^0_k then so is $R_2(a_2, \ldots, a_n) \Leftrightarrow R_1(42, a_2, \ldots, a_n)$ (specialised with $a_1 = 42$). Indeed, if $k = 0$, then this is immediate (R_1 and R_2 are both decidable) and otherwise there exists, by definition, a Σ^0_{k-1} relation R'_1 such that $R_1(a_1, \ldots, a_n) \Leftrightarrow \forall r, R'_1(a_1, \ldots, a_n, r)$. But then we have $R_2(a_2, \ldots, a_n) \Leftrightarrow \forall r, R'_2(a_2, \ldots, a_n, r)$ with $R'_2(a_2, \ldots, a_n, r) \Leftrightarrow R'_1(42, a_2, \ldots, a_n, r)$ which is Σ^0_{k-1} by induction hypothesis.

6.1 Arbitrary join

The atoms of Equ($\mathbb{N}$) are equivalence relations such that $\bot \prec \mathcal{E}$, and it is easily seen that an equivalence relation is an atom iff it is singular and the unique non-singleton class consists of two elements, that is there are distinct a, b such that $a\mathcal{E}b$ and for all $x \neq a, b$ and all $y \neq x$, we have $\neg(x\mathcal{E}y)$.

Lemma 335. *Every atom in Equ($\mathbb{N}$) is decidable in zero space and linear time by a two-tape Turing machine (hence, in particular, each atom can be decided by a deterministic, one-way, two-tape automaton).*

Proof. Let $\mathcal{E}$ be an atom and let a, b be distinct elements such that $a\mathcal{E}b$ and for all $x \neq a, b$ and all $y \neq x$, we have $\neg(x\mathcal{E}y)$. Consider a Turing machine having both a and b encoded in its states: on input (m, n), the machine first checks if $m = a$ or $m = b$ (no space needed, constant time $|a| + |b|$), then it checks if $n = a$ or $n = b$ (no space needed, constant time $|a| + |b|$). If both tests are true, the machine accepts, otherwise it checks whether $m = n$ (no space, linear time $\min(|m|, |n|)$) and accepts or rejects accordingly. □

Each of the tests done by this machine only require to read the corresponding word once and can easily be performed by a (two-tape) DFA. Performing all the tests at the same time (rather than sequentially as described here) is then a standard product of DFA, thus all this can be done by reading m and n only once (one-way).

Note that Lemma 335 does *not* entail that any atom automatic as the very restricted notion of automatic equivalence relation we have defined requires automata to have a single tape.

Proposition 336. *Let $I \subseteq \mathbb{N}$. There exists a set of atoms whose join is the singular equivalence with non-singleton class I.*

Proof. Let $x \in I$ be an element of I. For each other $i \in I$, let $\mathcal{A}_i$ be the atom with $x\mathcal{A}_i i$. Let $\mathcal{E} = \bigvee \mathcal{A}_i$, it is the singular equivalence whose non-singleton class is I. □

Corollary 337. *None of Σ^0_k, Π^0_k, Δ^0_k are closed under arbitrary join.*

Proof. Let I be, *e.g.* a non-Σ^0_k set containing 0. By the construction above, there exists a set of atoms whose join is the singular equivalence relation, $\mathcal{E}$, with non-singleton class I.

Each atom is decidable by an automaton, hence Σ^0_k. However, we have $x \in I \Leftrightarrow 0\mathcal{E}x$ hence $\mathcal{E}$ is not Σ^0_k (that is, the membership in I is the specialisation of $\mathcal{E}$ with the first argument equal to 0).

The reasoning is the same for the other cases. □

[3]In the sense of *program specialisation*, which is the modern equivalent of the *snm*-theorem.

6.2 Arbitrary meet

Recall that the upper set $\uparrow\{n\}$ contains n and all the elements greater than n.

We say that an equivalence is *small* if it has finitely many classes. Note that small singular equivalence relations have finitely many singleton classes, whose largest element is N and the non-singleton class is thus the union of a finite set and the upper set $\uparrow\{N+1\}$.

Note that small singular equivalence relations are decidable by single-tape automata by encoding the finitely many singletons in the states and checking whether one of the inputs is among these numbers. Actually, since the inputs need only to be compared against finitely many fixed numbers, this only required a fixed amount of time as the input don't need to be completely read if they differ from these numbers.

Proposition 338. *For every $I \subseteq \mathbb{N}$, there is a set of small singular equivalence relations whose meet is the singular equivalence with non-singleton class I.*

Proof. Let $I \subseteq \mathbb{N}$. Let $(f_i)_{i\in\mathbb{N}}$ be a strictly increasing sequence and let $F_i = I \bigcap [0; f_i[$. Note that $I = \bigcup_i F_i$.

Let $\mathcal{F}_i$ be the small singular equivalence with non-singleton class $F_i^+ = (F_i \bigcup \uparrow\{f_i\})$, note that we may have several copies of the same equivalence in the family if, typically, $f_{i+1} = f_i + 1$ and $f_i \in I$, but this is not a problem. Finally, let $\mathcal{E} = \bigwedge \mathcal{F}_i$. By definition, we have $m\mathcal{E}n$ iff $m\mathcal{F}_i n$ for all i.

Let $m \neq n$ with $m\mathcal{E}n$. As $(f_i)_{i\in\mathbb{N}}$ is strictly increasing, there exists a j such that $f_j \geq \max(m,n)$ thus $m\mathcal{F}_j n$ implies $m, n \in F_j$ hence $m, n \in I$.

Conversely, if $m, n \in I$ then let i be the smallest number such that $m \in F_i$. Let $k \in \mathbb{N}$ be arbitrary. Now,

- If $i \leq k$ then $F_i \subseteq F_k$ hence $m \in F_k \subseteq F_k^+$.
- If $k < i$ then we have both $m \in I$ and $m \notin F_k$, and thus $m \geq f_k$. Hence $m \in \uparrow\{f_k\} \subseteq F_k^+$.

Thus, for all k we have $m \in F_k^+$. Similarly, $n \in F_k^+$. Hence, for all k, $m\mathcal{F}_k n$ which implies $m\mathcal{E}n$.

Thus, $\mathcal{E}$ is the singular equivalence with non-singleton class I. □

Note that the ordering (and upper sets) plays a big role in the proof, hence adapting it to equivalences on programs require some care.

Corollary 339. *None of Σ^0_k, Π^0_k, Δ^0_k are closed under arbitrary meet.*

Proof. Let I be, *e.g.* a non-Σ^0_k set containing 0. By the construction above, there exists a set of small singular equivalences whose meet is the singular equivalence relation, $\mathcal{E}$, with non-singleton class I.

Each small singular equivalence is decidable by an automaton, hence Σ^0_k. However, we have $x \in I \Leftrightarrow 0\mathcal{E}x$ hence $\mathcal{E}$ is not Σ^0_k (that is, the membership in I is the specialisation of $\mathcal{E}$ with the first argument equal to 0).

The reasoning is the same for the other cases. □

Corollary 340. *Let S be a time or space bound. The set of S-decidable equivalence relations is not closed under arbitrary meet.*

In particular, $\mathrm{Equ}(\mathbb{N})_{\mathrm{r.e.}}$, $\mathrm{Equ}(\mathbb{N})_{\mathrm{dec}}$, $\mathrm{Equ}(\mathbb{N})_{\mathrm{p.r.}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{ExpTime}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{Pspace}}$, $\mathrm{Equ}(\mathbb{N})_{\textsc{Ptime}}$, and $\mathrm{Equ}(\mathbb{N})_{\textsc{LogSpace}}$ are not closed under arbitrary meet (they form lower subsemilattices but not complete lower subsemilattices).

6.3 Arithmetical meet and join

The previous results assume that the set of equivalence relations may be chosen arbitrarily. Especially, it is based on a set I of arbitrary difficulty. This essentially means that the results boil down to the fact that there are uncountably many such I, while there are only countably many arithmetical relations.

We now look at what happens if the set of relations itself is to have some bound. Specifically, we are concerned with the meet and join of a Σ^0_k set of equivalences. Σ^0_k (and especially, recursively enumerable) is a sensible bound: It means that one can enumerate each of the equivalences (with a proper oracle, or none for recursively enumerable) until all the needed ones have been found. On the other hand, a Π^0_k set of equivalences would mean that one can enumerate (with oracle) the complement to this set, which is less practical.

First, note that a given relation may be expressed as several different formulas. The usual logic point of view is to concentrate on formulas and thus say, for example, that the formula $\neg(x \mid\mid y)$ is *equivalent* to the formula $\neg x \,\&\&\, \neg y$. However, if we look at the *relations*, that is the subsets of $\mathbb{N} \times \mathbb{N}$, then these two formulas describe the exact same relation. This is exactly the usual difference between syntax and semantics common in computer science, namely that there exists different *programs* (or TMs) that describe the exact same *function*. Formulas (or programs) are a convenient (especially, finite) way to represent relations (or functions) but using them costs us unicity.

Here we consider given a computable encoding of formulas (in the arithmetical hierarchy) and we note φ_n the formula corresponding to number n. The encoding is computable in the sense that given n there exists a Turing Machine that outputs the text of φ_n in a human-readable format (such as the one used in this text), and conversely. Obviously, because we handle relations that are themselves not computable, we cannot expect a TM to do more than a translation between representations. We also note Φ_n the *relation* described by number n (formula φ_n). Obviously, the (non-injective) map $n \mapsto \Phi_n$ is not computable. All this is similar to numbering programs and have $\mathtt{p}_i$ be the i^{th} program computing a function $[\![\mathtt{p}_i]\!]$. Even if $i \mapsto \mathtt{p}_i$ is injective and computable (depending on the encoding of program texts, obviously), $i \mapsto [\![\mathtt{p}_i]\!]$ is neither injective nor computable.

Definition 341. Let Γ, Λ be two sets in the arithmetical hierarchy. A set $A \subseteq \mathbb{N}$ of equivalence relations is said to be a Γ *set of* Λ *equivalence relations* if there exists a Γ set $I \subset \mathbb{N}$ such that (i) $\{\, \Phi_i \mid i \in I \,\} = A$ and (ii) each Φ_i is a Λ relation.

We shall use an index set I as a representative of A. That is, I is considered as Γ set of Λ *formulas* φ_i that describe exactly the relations Φ_i. Because $i \mapsto \Phi_i$ is not injective, I is not unique and it is possible that $\Phi_i = \Phi_j$, that is the same relation (in A) has several representative formulas (in I).

Proposition 342. *For all $k \geq 1$, the join of a Σ^0_k set of Σ^0_k equivalences is a Σ^0_k equivalence.*

Proof. Let I be a Σ^0_k set of integers such that each Φ_i is a Σ^0_k equivalence. Let $\mathcal{E} = \bigvee \Phi_i$. By definition, we have $x\mathcal{E}y$ if and only if

$$\exists n, a_1, \ldots, a_n, x_1, \ldots, x_n. \quad a_1 \in I \;\&\&\; \ldots \;\&\&\; a_n \in I \quad \&\&\; x\Phi_{a_1}x_1 \;\&\&\; x_1\Phi_{a_2}x_2 \;\&\&\; \ldots \;\&\&\; x_n\Phi_{a_n}y$$

Because I is Σ^0_k, so is its membership relation, and each Φ_{a_i} is Σ^0_k by hypothesis. Hence, the relation after the quantifier is Σ^0_k and this level is not changed by adding extra existential quantifiers in front.

So, $\mathcal{E}$ a Σ^0_k relation. □

Note that because $i \leq j$ implies $\Sigma^0_i \subset \Sigma^0_j$, we immediately have that the join of a Σ^0_i set of Σ^0_j equivalences is Σ^0_j. In particular, the join of a recursively enumerable set of Σ^0_j equivalences is a Σ^0_j equivalence rekation.

Corollary 343. *For all $k \geq 0, j \geq 1$, the join of a Σ^0_k set of Π^0_j equivalence relation is not necessarily a Π^0_j equivalence relation.*

Proof. By Proposition 332, Π^0_j is not closed under finite join. Since any finite set is also Σ^0_k, Π^0_j cannot be closed under Σ^0_k join. □

Proposition 344. *For all positive integers n, there exists a decidable set of Σ^0_n equivalence relations whose meet is not Σ^0_n.*

Proof. Let I be the set of integers such that φ_i is a unary Σ^0_n formula. Given a suitable canonical representation of formulas, it is decidable by simply counting the quantifiers in front of the formula.

Let $A_k = \{\, i \in I \mid \Phi_i(k) \,\}$, the set of numbers of Σ^0_n formulas describing a predicate that accepts k. It is Σ^0_n by construction. Let $\mathcal{A}_k$ be the singular equivalence with non-singleton set A_k. The set of all the $\mathcal{A}_k$ is a decidable set of Σ^0_n equivalence relations.

Now, look at $\mathcal{A}_j \wedge \mathcal{A}_k$. It is easy to see that this is the singular equivalence with non-singleton set $A_j \bigcap A_k = \{\, i \in I \mid \Phi_i(j) \;\&\&\; \Phi_i(k) \,\}$, the set of numbers of Σ^0_n formulas describing a predicate that accepts both j and k.

Similarly, we can see that $\bigwedge \mathcal{A}_k$ is the singular equivalence with non-singleton set $\bigcap A_k = \{\, i \in I \mid \forall k \in \mathbb{N}, \Phi_i(k) \,\}$, that is the set of numbers of Σ^0_k formulas describing a predicate accepting all numbers, *i.e.* a tautology.

However, this set is Π^0_{k+1}-complete, thus $\bigwedge \mathcal{A}_k$ cannot be Σ^0_k. □

Proposition 345. *For all k, the meet of a Σ^0_k set of Π^0_k equivalence relations is Π^0_k.*

Proof. Let I be a Σ^0_k set of integers such that each Φ_i is a Π^0_k equivalence. Let $\mathcal{E} = \bigwedge \Phi_i$. By definition, we have $x\mathcal{E}y$ if and only if

$$\forall i, i \in I \Rightarrow x\Phi_i y$$

which is equivalent to

$$\forall i, i \notin I \,||\, x\Phi_i y$$

Because $i \in I$ is Σ^0_k, then its complement $i \notin I$ is Π^0_k. Next, each of the Φ_i is also Π^0_k, hence the right part is. Adding a $\forall$ in front of a Π^0_k formula does not change its level in the hierarchy.

Hence, $\mathcal{E}$ is Π^0_k. □

Here also, by inclusion of the hierarchy, $i \leq j$ implies that any Σ^0_i set of Π^0_j equivalence relations has a Π^0_j meet. Especially, the set of Π^0_k equivalence relations is closed under recursively enumerable meet.

Closure properties of arithmetical sets of limited equivalence relations

We briefly consider Σ^0_k sets of equivalence relations from the other classes considered elsewhere in this paper.

Proposition 346. *Let $k \geq 1$. The meet of a Σ^0_k set of automatic equivalence relations is not necessarily automatic.*

Proof. Let, for each $i \geq 1$, $\mathcal{E}_i$ be the automatic equivalence relation containing the two classes $\{i\}$ and $\mathbb{N} \setminus \{i\}$. Then, $\wedge_i \mathcal{E}_i = \perp$, which is not automatic. □

Similarly, Σ^0_k sets of subrecursive equivalence relations that include $\text{Equ}(\mathbb{N})_{\text{LogSpace}}$ are not necessarily closed under *join*: by Proposition 322 there is a finite set (hence, *a fortiori*, an r.e. set) of equivalence relations decidable in logarithmic space whose join is undecidable.

Proving that Σ^0_k sets of subrecursive equivalence relations that include $\text{Equ}(\mathbb{N})_{\text{LogSpace}}$ are not necessarily closed under *meet* requires substantially more work. The result is contained in the following proposition.

Proposition 347. *There is an r.e. set of elements of $Equ(\mathbb{N})_{LogSpace}$ whose meet is not decidable.*

Proof. Consider a standard representation of Turing machines as elements of $\{0,1\}^*$ such that it can be checked in logarithmic space whether $x \in \{0,1\}^*$ is a valid representation of a Turing machine (see e.g., [Pap94, Ch. 3]). Consider, for every $j \in \mathbb{N}$, the singular equivalence relation $\mathcal{E}_j$ whose unique non-singleton set A_j consists of those numbers n such that the binary expansion of n is the encoding of a Turing machine that *does not* halt in at most j steps. Observe that $A_1 \supseteq A_2 \supseteq \cdots$.

Define $\mathcal{E}_\infty$ as the singular equivalence relation whose unique non-singleton set, A_∞, consists of those numbers whose binary expansion is the encoding of a Turing machine that does not halt. Then, $\mathcal{E}_\infty = \wedge_j \mathcal{E}_j$.

Now, for each $j \in \mathbb{N}$, we have $\mathcal{E}_j \in \text{Equ}(\mathbb{N})_{\text{LogSpace}}$: Fix i; for every pair $(m,n) \in \mathbb{N}$, a Turing machine may check in logarithmic space whether the binary representation of m and n represents a Turing machine and then use a universal Turing machine to simulate running of both m and n for j steps. By proper construction of the universal machine, the space overhead required can be made constant in the space used by the machines that m and n represents, which is bounded above by j cells as both machines are run for at most j steps; the universal machine needs only a fixed number of counters beyond this overhead (see, e.g. [Pap94, Ch. 3]; in essence, the universal machine works by simulating one step of the simulated machine at a time and keeping a representation of its configuration in memory). Hence, each $\mathcal{E}_j \in \text{Equ}(\mathbb{N})_{\text{LogSpace}}$. Further observe that there is a Turing machine that, on input (the binary representation of) j produces a Turing machine for deciding $\mathcal{E}$ in logarithmic space: it simply specializes a universal Turing machine to j and outputs this along with some fixed (i.e., independent of j) operations for comparing binary representations and checking whether the inputs represent Turing machines. Hence, the set $\{equone_j : j \in \mathbb{N}\}$ is r.e.

But $\mathcal{E}_\infty$ is not decidable (indeed, is not even r.e.: if it were, we could enumerate the set of all non-halting Turing machines by fixing a single such machine and using it's corresponding natural number m to recursively enumerate all pairs $(m,n) \in \mathcal{E}_\infty$, hence recursively enumerate the set of all non-halting machines, a contradition). □

Corollary 348. *Let $k \geq 1$ and let $\mathcal{A}$ be a set of decidable equivalence relations such that $Equ(\mathbb{N})_{LogSpace} \subseteq \mathcal{A}$. Then there is a Σ^0_k set of equivalence relations from $\mathcal{A}$ such that $\wedge \mathcal{A}$ is not decidable.*

The same proof method used for the proof of Proposition 347 can be used to prove

Proposition 349. *Let $i \geq 1$. Then there is a Σ^0_i set of Δ^0_i equivalence relations whose meet is not Δ^0_i*

Proof. As the proof of Proposition 347, replacing the A_j by those numbers whose corresponding Turing machines are oracle machines with an oracle to a Σ^0_{j-1}-complete set and that do not halt in at most j steps. The proof now proceeds *mutatis mutandis*, $\mathcal{E}_\infty$ not being a Δ^0_j set (indeed, not even a Σ^0_j set). □

Part III

Experiments in Implicit Complexity

Detection of Non-Size Increasing Programs in Compilers

Jean-Yves Moyen, Thomas Rubiano

Developments in Implicit Computational Complexity, 2016

Abstract:

Implicit Computational Complexity (ICC) aims at giving machine-free characterisations of complexity classes. Because it is usually sound but not complete, it actually provides certificates that a given program can be run within a given amount of resources. ICC is usually applied on toy languages with restricted expressivity, we show here that it can be performed on real programming languages.
Because it is usually a static, syntactical analysis of the programs, ICC is well-suited to be performed at compile time. The bounds given by ICC can then be used to fuel some optimisation or to produce certificates of good behaviour. Modern compilers do most of their work in a modular sequences of passes done on some Intermediate Representation (IR) language. The IR is a generic typed assembly-like language and thus very well suited to express ICC criteria. The modularity of the passes make it easy to add one and to re-use existing ones at will.
We focus here on the relatively simple analysis of Non-Size Increasing (NSI) programs. We've implemented a NSI analysis for the LLVM compiler. This can be seen as a proof of concept that ICC and compilers are able to interact productively.

1 Introduction

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

1.1 Context and Motivations

ICC aims at finding syntactic criterion on programs that guarantee some semantic property (usually some complexity bound). It emerged with the *Bounded Recursion* of Cobham [Cob62] but was really created by the breakthrough result on *Safe Recursion* by Bellantoni and Cook [BC92]. Since then, many different directions have been studied in ICC. The main ideas revolve around following dataflow (the *Tiering* of Leivant and Marion [LM95], the *Size Change Termination* of Lee, Jones and Ben-Amram [LJBA01], the *Non-Size Increasing* programs of Hofmann [Hof99], . . .), performing a static check on values (the *Quasi-interpretations* of Bonfante, Marion and Moyen [BMM11], the *mwp*-polynomials of Kristiansen and Jones [KJ09], . . .) or enforcing a strict type checking (variations on Girard's Linear Logic [Gir87] such as Baillot and Terui's DLAL [BT09]). Schöpp introduced a more restricted Bounded Linear Logic: the Stratified Bounded Affine Logic [Sch07]. Hofmann and Jost [HJ03] furnish upper bounds on the heap usage in functional programming by accepting some restrictions.

Most of these results usually concern "toy" languages such as Term Rewriting Systems [AM13], λ-calculus or the Loop language. 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, . . .) which make analysis complicated. Thus, even with 20 years of ICC, it is not possible today to apply its results on actual programs. We start filling the gap.

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), it is perfectly suited to happen at compile time.

From the other end of the gap, we'll use the *intermediate representation* in a compiler. During the compilation process, the source code is first translated in an intermediate language where optimisations are performed before being translated again into the target code in assembly language. This intermediate representation has few constructions and is simple enough to perform all the optimisations steps. Especially for our practical case, it strips all the constructions of the programs but keep the control flow and the allocations that we want to study.

Moreover, compilers already contain many analysis 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 just 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.

1.2 Analysis and Optimisation

Compilers are usually focused on optimisation. Indeed, the goal is to produce an efficient code in order to have a fast program. ICC mostly provides analysis without much optimisation of the code. However, analysis and optimisation are not so far apart. . .

Firstly, 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. 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.

Secondly, 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 [Nec97]. 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.

Next, some ICC analysis are known to also embed some program transformation in them. Notably, the *Quasi-Interpretations* method guarantee 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 an analysis in compiler will indicate which part of the code should be transformed by which method.

Lastly, these are also first steps in experimenting ICC into compilers. 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. Thus, this can be seen as a proof of concept: yes, ICC and compilers can work together and can fuel each other fruitfully. This opens the way for future works.

2 Non Size Increasing programs

2.1 Safe Recursion and Non Size Increasing

In Safe Recursion, Bellantoni and Cook analysed repeated iterations as a source of exponential growth. Typically, exponentiation can be computed by iterating doubling, itself an iteration.

In order to prevent repeated iterations, they designed a syntactical criterion (hence, a static analysis) based on splitting variables into *normal* and *safe* ones, which can be interpreted as with/without energy. Next, iteration must be performed on a normal variable (which provides the "energy" to run the program) and the result must be safe (the energy has been used). Thus, when computing the exponential with the usual recurrence $2^n = 2 \times 2^{n-1}$, the result of the recursion (2^{n-1}) must be safe and cannot be used to control the doubling ($2\times$).

However, 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, insertion sort works by iterating the insertion of an element into a sorted list, itself an iteration. The Safe Recursion prevents writing the insertion sort (or rather requires to write it in a non natural way).

Hofmann identified that the problem does not come from the exponential or sorting function but from the doubling or insertion function. Indeed, doubling produces an output twice as large as its input while insertion produces an output basically as large as the input. Thus, it is not harmful to iterate insertion, which does not increase the size of its data (or only by a constant factor) while it is harmful to iterate doubling which drastically increases the size of data. This justifies the detection of *Non Size Increasing* (NSI) programs.

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 in the result of functions. Thus, $\diamond$ in the result can be seen as "price" to be paid to compute and that can only be paid by diamonds already present in the arguments.

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 $\diamond, \alpha, \alpha\ list \to \alpha\ list$.

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

2.2 NSI and imperative programs

The diamonds have a very nice and natural interpretation in imperative programs, as shown by Hofmann.

The classical representation of lists in an imperative language is to have cells containing a value and a pointer to the next cell, the list itself being a pointer to the first cell. When performing a *cons*, a new cell must be created. For this, new memory must be allocated (`malloc`). This new memory is exactly the diamond we need to perform the *cons*! Indeed, if *cons* is given as a third argument a pointer (to a place in memory which is assumed to be free), then it does not need to allocate memory and can use the one that is provided.

Hofmann shown that NSI programs can be compiled into `malloc`-free `C` programs. The diamonds are *essentially* pointers and a program that is NSI does not need extra diamonds, hence does not need to allocate new memory.

Having a program which is guaranteed to be NSI is not only a complexity analysis. It also gives some security properties (the program won't overflow memory and won't cause memory leaks) and some possibilities for optimisation. It becomes indeed possible to completely remove the `malloc` from the code and let the program efficiently reuse its memory. This will prevent several system calls and calls to the standard library that can slow down the program execution.

2.3 A Control Flow Graph analysis

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. Consider, for example, the following insertion sort function (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))
```

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's the good place 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, `d'`). 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**.

It's easy to do this analysis using a Control Flow Graph (`CFG`). A Control Flow Graph is a graph representation of all paths that might be traversed by a program during its execution. We can see each node as a program state and each edge as an instruction.

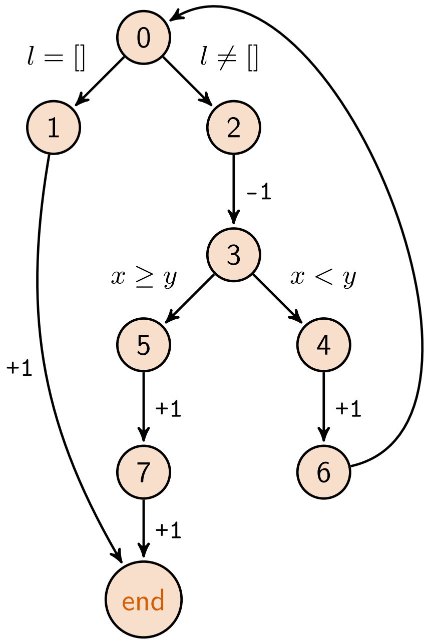


Figure 1: Resources Control Graph of the insertion function

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 [Moy09] (RCG) of Figure 1. 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* 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's 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 Bellman-Ford algorithm can be used here to provide the shortest path instead of the Dijkstra's one which is not able to deal with negative edge weights and detect negative loop.

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

3 Compiler Infrastructure

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 these analysis and transformations are done on a typed assembly like language: the *Intermediate Representation.*

Because this *Intermediate Representation* (IR) is a good abstraction level we can do our analysis directly in compilers. Compiler comes with a lot of tools working at different compilation times. Compilers are designed to sequentially make analysis and transformations called passes on the sources code.

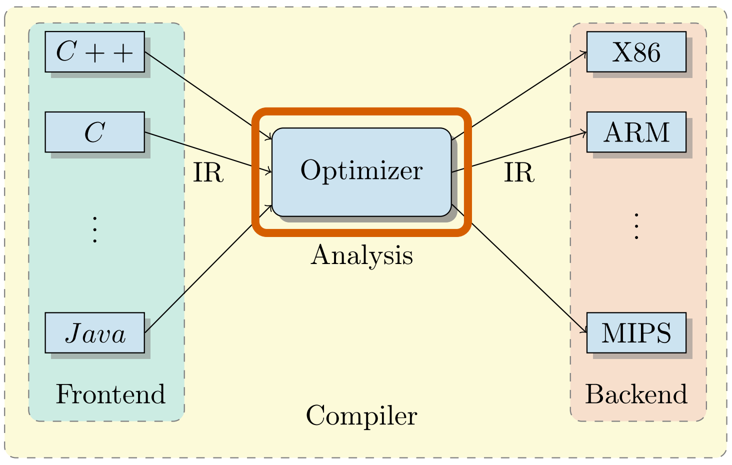


Figure 2: Compiler architecture

3.1 Compiler design

Compilers are generally composed of three parts, shown on Figure 2:

- one *front-end* for each source language, it's composed by a *lexer* and a *parser* which finally build an IR. This translation simplifies the job of the rest of the compiler which doesn't want to deal with each expressivity of each programming language.

- a *middle-end* also called optimizer or Pass Manager which provides information and/or transforms IR to semantically *equivalent* IR supposed to be *better/faster*.
- and a *back-end* for each architecture which produces a machine code.

The *Intermediate Representation* is pretty similar to an assembly language. It's a lower-level programming language than the input one but it's a higher-level than real assembly language.

The *optimizer* (`opt` in LLVM) provides optimizations but also analysis, both are called *passes*.

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

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 easy for other-ends to use them.

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's modular; it's more and more used. For instance, more, and more efforts have been done to build Debian with LLVM[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 analysis are, of course, feasible in GCC, Compcert[2], etc. Compcert is a certified compiler using the Coq proof assistant, it guarantees that any transformations during the compilation cannot alter the program's semantics. The produced assembly will compute exactly what the source said before compilation.

The LLVM Project [Lat02] 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.

LLVM is almost well designed for our work because it:

- Offers modularity, simplicity and a good research environment for compilers developers.
- Operates transparently to the developer.
- Provides a *multi-stage* optimization strategy.

3.2 LLVM IR, instruction set and Data structure

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.

A lot of well known optimizations are already 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 "$add\ \ X, 0 \to X$", "$xor\ \ X, X \to 0$" etc... and can detect and replace them.

The LLVM Intermediate Representation is source-language-independent, mainly because it uses a low-level instruction set slightly richer than assembly languages, it's a RISC-like virtual instruction set. The instruction set consists of 31 opcodes, just enough to don't loose type expressivity but still a *low-level* representation. Most of these operations are in a *three-address* form: that's means that they take one or two operands and produce one result. But, unlike RISC instructions, LLVM-IR is strictly typed, then 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. Instructions can be polymorphic: for instance `add` can operate on different types, this widely reduces the number of opcodes. Here, we will only be interested in instructions for typed memory allocation.

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, this instructions are converted to the appropriate native function calls, allowing also customizations. There are no implicit accesses to memory, this simplifies all memory access analysis.

LLVM has shown that an efficient low-level representation enriched with type information can support high-level analysis and transformations.

[1]sylvestre.ledru.info/blog/2014/09/11/rebuild-of-debian-using-clang-3-5
[2]compcert.inria.fr/compcert-C.html

4 RCG computation and positive loops detection

LLVM already builds the *CFG* of every function. 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 analysis.

Each node in the CFG represents a *basic block*, i.e. a succession of instructions without any branching. 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.

The *RCG* can be built by traversing the entire *CFG* once and counting the number of memories allocations and deallocations on each node. This can be done independently of the order of the blocks.

In order to do this, we use a LLVM tool: Basic Blocks visitor which goes through each *basic block* on the *CFG*. We can add a function to run for each basic block. Here we just compute their weight and map this to be used by another pass.

Now we can compute the maximal weight or worst case space that might be used by each function. We can use the Bellman-Ford's algorithm to find the heaviest path for our weighted graph in a polynomial time.

Basic Blocks 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 need to use a Depth-First Search to obtain our nodes in the correct order.

If we reconsider the analysis, it just provides an answer to the following question: "is the program NSI?". We actually don't provide the accurate amount of space needed, but we detect if this amount is fixed. That is, we need to detect positive loops without regarding how many times they will occur. Thus we consider all positive loops as occurred a non-determined number of time. In fact we can be more precise by detecting static loops and upper bounds but it already exists passes that find invariants and unroll loops.

5 Conclusions and further works

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 doesn't provide accurate bounds.

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

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[3] 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's only added, this could work on any pre-existent code. By this way we could globalize the "proof-carrying code" [Nec97] 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 of `malloc` calls is not a new idea [Hof00b] 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 just simulate them without any system call.

We can also, by studying more accurately relations between input and bounds [AAG+08], approximate a *Space Complexity* [ASM13] and, maybe, the *termination* [LJBA01] because this last work is also based on weighted Control Flow Graphs or Resources Control Graphs [Moy09].

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

Loop Quasi-Invariant Chunk Motion by peeling with statement composition

Jean-Yves Moyen, Thomas Rubiano, Thomas Seiller

Submitted to the 26th International Conference on Compiler Construction, 2017

Abstract:

Several techniques for analysis and transformations are used in compilers. Among them, the peeling of loops for hoisting quasi-invariants can be used to optimize generated code, or simply ease developers' lives. In this paper, we introduce a new concept of dependency analysis borrowed from the field of Implicit Computational Complexity (ICC), allowing to work with composed statements called "Chunks" to detect more quasi-invariants. Based on an optimization idea given on a `WHILE` language, we provide a transformation method - reusing ICC concepts and techniques [Kri, KKP$^+$81] - to compilers. This new analysis computes an invariance degree for each statement or chunks of statements by building a new kind of dependency graph, finds the "maximum" or "worst" dependency graph for loops, and recognizes if an entire block is Quasi-Invariant or not. This block could be an inner loop, and in that case the computational complexity of the overall program can be decreased.

In this paper, we introduce the theory around this concept and present a proof-of-concept tool that we developed: an analysis and a transformation passes on a toy C parser written in Python called "pycparser" implemented by Eli Bendersky[a]. In the near future, we will implement these techniques on real compilers, as some of our optimisations are visibly better than current optimisations implemented in both GCC and LLVM.

[a]https://github.com/eliben/pycparser

1 Introduction

Loop optimization techniques based on quasi-invariance are well-known in the compilers community. The transformation idea is to peel loops a finite number of time and hoist invariants until there are no more quasi-invariants. As far as we know, this technique is called "peeling" and it was introduced by Song *et al.* [SFGH00].

The present paper offers a new point of view on this work. From a proven optimization on a `WHILE` language by Lars Kristiansen [Kri], we provide a redefinition of peeling and another transformation method based on techniques developed in the field of Implicit Computational Complexity.

Implicit Computational Complexity (ICC) studies computational complexity in terms of restrictions of languages and computational principles, providing results that do not depend on specific machine models. Based on static analysis, it helps predict and control resources consumed by programs, and can offer reusable and tunable ideas and techniques for compilers. ICC mainly focuses on syntactic [Cob62, BC92], type [Gir87, BT09] and Data Flow [LJBA01, Hof99, KJ09, Moy09] 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" [LJBA01] or "Resource Control Graphs"[Moy09] which track data values' behavior and use a matrix notation inspired by [AA02], or "mwp-polynomials" [KJ09] to provide bounds on data size.

For our analysis, we focus on dependencies between variables to detect invariance. Dependency graphs [KKP+81] can have different types of arcs regarding to represent different kind of dependencies. Here we will use a kind of Dependence Graph Abstraction [Coc70] that can be used to find local and global quasi-invariants. Based on these techniques, we developed an analysis and a transformation passes on a toy `C` parser written in `Python`. Initial tests show a visible improvement over current `GCC` and `LLVM` optimisations, and we will be implementing our techniques on real compilers in the near future.

We propose a tool, seen as a proof-of-concept, which is notably able to peel and hoist an inner loop, thus decreasing the complexity of a program from n^2 to n.

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

Invariants are basically detected using Algorithm 1.

```
Data: List of Statements in the Loop
Result: List of Loop-invariants LI
Initialization;
while search until there is no new invariant... do
    for each statement s do
        if each variable in s
        has no definition in the loop or
        has exactly one loop-invariant definition or
        is constant then
            Add s to LI;
        end
    end
end
```

Algorithm 1: Basic invariants detection

A *dependency graph* around variables is needed to provide relations between statements. For quasi-invariance, we need to couple dependence and *dominance* informations. In [SFGH00], 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`. In the present paper, this *length* is called *invariance degree.*

2 Rethinking the theory

In this section, we redefine our own types of relations between variables to build a new dependency graph and apply a composition inspired by the *size-change graph* [LJBA01].

2.1 Relations and Data Flow Graph

We work with a simple imperative `WHILE`-language (the grammar is shown in Figure 1), with semantics similar to `C`.

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 represent any command which does not modify its variables but use them and should not be moved around carelessly (typically, a `printf`). *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*.

(Variables)	X	$::=$	$X_1 \mid X_2 \mid X_3 \mid \dots\ X_n$
(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* `fi` $\mid$
			`use(`X_1`,...,`X_n`)`

Figure 1: Grammar

We start by giving an informal but intuitive definition of the notion of *Data Flow Graph* (DFG). A DFG represents dependencies between variables as a bipartite graph as in Figure 2. Each different types of arrow represents different types of dependencies.

`C` $:= [x = x + 1;$

$y = y;$

$z = 0;]$

$$\begin{tikzcd}
x \arrow[r, "\text{dependence}"] & x \\
y \arrow[r, dashed, "\text{propagation}"] & y \\
z \arrow[r, phantom, "\text{reinitialization}"] & z
\end{tikzcd}$$

Figure 2: Types of dependence

Each variable is showed twice: the occurrence on the left represents the variable before the execution of the command while the occurrence on the right represents the variable after the execution. Dependencies are then 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 x then corresponds to the absence of arrows ending on the right occurrence of x. Figure 2 illustrates these types of dependencies; let us stress here that the DFG would be the same if the assignment $y = y$; were to be removed from `C` since the value of y is still propagated.

More formally, a DFG of a command `C` is a triple $(V, \mathcal{R}_{\text{dep}}, \mathcal{R}_{\text{prop}})$ with V the variables involved in the command `C` and a pair of two relations on the set of variables. These two relations 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*. 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*.

More technically, we will work with an alternative definition in terms of matrices. While less intuitive, this formal definition allows for more natural definitions, based on standard linear algebra operations. Before providing the formal definition, let us introduce the semi-ring $\{\emptyset, 0, 1\}$: the addition $\oplus$ and multiplication $\otimes$ are defined in Figure 3. Let us remark that, identifying $\emptyset$ as $-\infty$, this is a sub-semi-ring of the standard *tropical semi-ring*, with $\oplus$ and $\otimes$ interpreted as max and $+$ respectively.

$\oplus$	$\emptyset$	0	1
$\emptyset$	$\emptyset$	0	1
0	0	0	1
1	1	1	1

$\otimes$	$\emptyset$	0	1
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
0	$\emptyset$	0	1
1	$\emptyset$	1	1

Figure 3: Addition and Multiplication in the semi-ring $\{\emptyset, 0, 1\}$.

Definition 350. A *Data Flow Graph* for a command `C` is a $n \times n$ matrix over the semi-ring $\{\emptyset, 0, 1\}$ where n is the number of variables involved in `C`.

We write M(`C`) the DFG of `C`. At line i, column j, we have a $\emptyset$ if the output value of the jth variable does not depend on the input value of the ith; a 0 in case of propagation (unmodified variable); and a 1 for any other kind of dependence.

Definition 351. Let `C` be a command. We define In(`C`) (resp. Out(`C`)) as the set of variables *used* (resp. *modified*) by `C`.

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) are done in the obvious way, generalising slightly the definitions of DFGs shown in Figure 2.

Composition and Multipath

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 352. Let 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 $\mathrm{M}(\mathtt{C_1})\mathrm{M}(\mathtt{C_2})\dots\mathrm{M}(\mathtt{C_n})$.

Following the usual product of matrices, the product of two matrices A, B is defined here as the matrix C with coefficients: $C_{i,j} = \bigoplus_{k=1}^{n}(A_{i,k} \otimes B_{k,j})$.

This operation of matrix multiplication corresponds to the computation of *multipaths* [LJBA01] in the graph representation of DFGs. We illustrate this intuitive construction on an example in Figure 4.

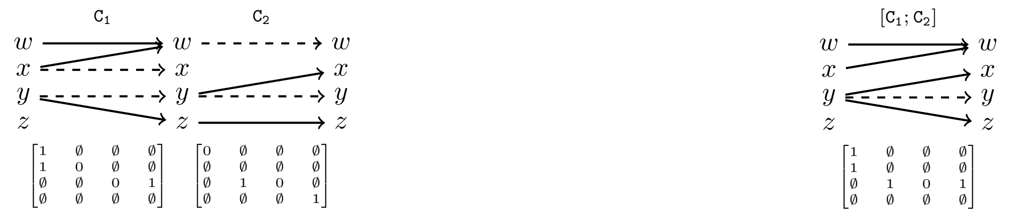


Figure 4: DFG of Composition.

Here $\mathtt{C_1} := [w = w + x; z = y + 2;]$ and $\mathtt{C_2} := [x = y; z = z * 2;]$

Condition

We now explain how to compute the DFG of a command $\mathtt{C} :=$ `if E then` $\mathtt{C_1}$;, from the DFG of the command $\mathtt{C_1}$.

Firstly, we notice that in C, all modified variables in $\mathtt{C_1}$, i.e. in $\mathrm{Out}(\mathtt{C_1})$, will depend on the variables used in E. Let us denote by $\mathrm{M}(\mathtt{C})^{(\mathtt{E})}$ the corresponding DFG, i.e. the matrix $\mathrm{M}(\mathtt{C}) \oplus (E^tO)$, where E (resp. O) is the vector representing variables in[1] $\mathrm{Var}(\mathtt{E})$ (resp. in $\mathrm{Out}(C_1)$), and $(\cdot)^t$ denotes the transpose.

Secondly, we need to take into account that the command $\mathtt{C_1}$ may be skipped. In that case, the overall command C should act as an empty command, i.e. be represented by the identity matrix Id (diagonal elements are equal to 0, all other are equal to $\emptyset$).

Finally, the DFG of a conditional will be computed by summing these two possibilities, as in Figure 5.

Definition 353. Let C be a command of the form `if E then` $\mathtt{C_1}$;. Then $\mathrm{M}(\mathtt{C}) = \mathrm{M}(\mathtt{C_1})^{\mathtt{E}} \oplus \mathrm{Id}$.

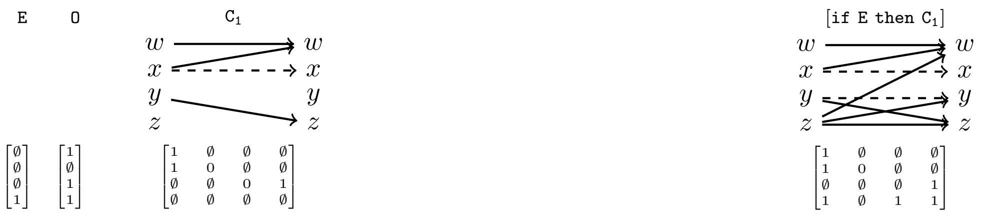


Figure 5: DFG of Conditional.

Here $\mathtt{E} := z \geq 0$ and $\mathtt{C_1} := [w = w + x; z = y + 2; y = 0;]$;

While Loop

Finally, let us define the DFG of a command C of the form $\mathtt{C} :=$ `while E do` $\mathtt{C_1}$;. This definition splits into two steps. First, we define a matrix $\mathrm{M}(\mathtt{C_1^*})$ representing iterations of the command $\mathtt{C_1}$; then we deal with the condition of the loop in the same way we interpreted the conditional above.

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 add all possible values of n. The following expression then expresses the DFG of the (informal) command $\mathtt{C_1^*}$ corresponding to "iterating $\mathtt{C_1}$ a finite (but arbitrary) number of times":

$$\mathrm{M}(\mathtt{C_1^*}) = \mathrm{limit}_{k\to\infty} \bigoplus_{i=1}^{k} \mathrm{M}(\mathtt{C_1})^i$$

[1] *I.e.* the vector with a coefficient equal to 1 for the variables in $\mathrm{Var}(\mathtt{E})$, and $\emptyset$ for all others variables.

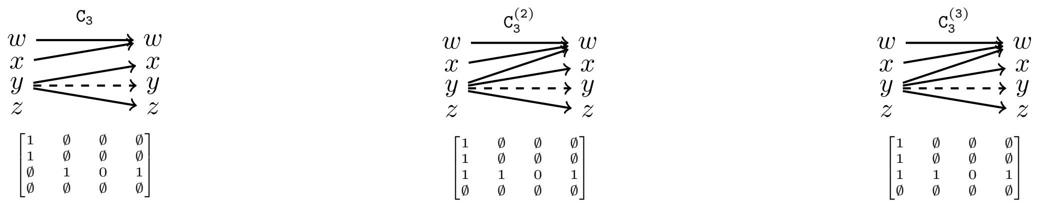


Figure 6: Finding fix point of dependence (simple example)
Here $\texttt{C}_3 :=[w = w + x; z = y + 2; x = y; z = z * 2]$;

To ease notations, we note $\mathrm{M}(\texttt{C}_1^{(\texttt{k})})$ the partial summations $\sum_{i=1}^{k} \mathrm{M}(\texttt{C}_1)^i$.

Since the set of all relations is finite and the sequence $(\mathrm{M}(\texttt{C}_1^{(\texttt{k})}))_{k\leqslant 0}$ is monotonous, this sequence is eventually constant. I.e., there exists a natural number N such that $\mathrm{M}(\texttt{C}_1^{(\texttt{k})}) = \mathrm{M}(\texttt{C}_1^{(\texttt{N})})$ for all $k \geqslant N$. One can obtain the following bound on the value of N.

Lemma 354. *Consider a command* C *and define* $K = \min(i, o)$*, where* i *(resp.* o*) denotes the number of variables in* In(C) *(resp.* Out(C)*). Then, the sequence* $(\mathrm{M}(\texttt{C}^{(\texttt{k})}))_{k\geqslant K}$ *is constant.*

Figure 6 illustrates the computation of $\cdot^*$. The second step then consists in dealing with the loop condition, using the same constructions as for conditionals.

Definition 355. Let C be a command of the form `while E do` $\texttt{C}_1$;. Then $\mathrm{M}(\texttt{C}) = \mathrm{M}(\texttt{C}_1^*)^{\texttt{E}}$.

2.3 Independence

Our purpose is to move commands around: exchange them, but more importantly to pull them out of loops when possible. We allow these moves only when semantics are preserved: to ensure this is the case, we describe a notion of independence.

Definition 356. If $\mathrm{Out}(\texttt{C}_1) \cap \mathrm{In}(\texttt{C}_2) = \emptyset$ then $\texttt{C}_2$ is *independent* from $\texttt{C}_1$. This is denoted $\texttt{C}_1 \prec \texttt{C}_2$.

It is important to notice that this notion is not symmetric. As an example, let us consider Figure 7: Here, $\texttt{C}_2$ is

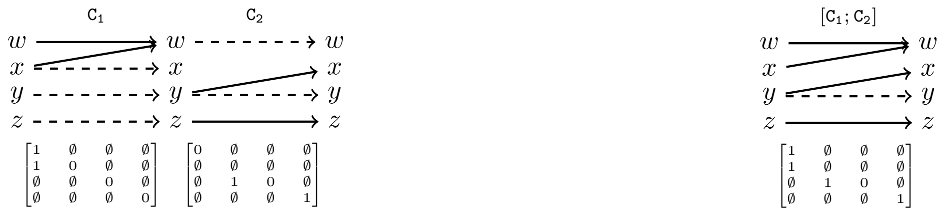


Figure 7: Composition of independent chunks of commands
Here $\texttt{C}_1 :=[w = w + x;]$ and $\texttt{C}_2 :=[x = y; z = z * 2]$;

independent from $\texttt{C}_1$ but the inverse is not true.

A particular case is *self-independence*, i.e. independence of a command w.r.t. itself. In that case, we can find non-trivial program transformations preserving the semantics. We denote by $[\![\texttt{C}]\!] \equiv [\![\texttt{D}]\!]$ the relation "C and D have the same semantics".

Lemma 357 (Specialization for `while`)**.** *If* $\texttt{C}_1$ *is self-independent:*

$$[\![\texttt{while E do C}_1]\!] \equiv [\![\texttt{if E then C}_1\texttt{; While E do skip}]\!]$$

Remark that we need to keep the loop `While` with a skip statement inside because we need to consider an infinite loop if E is always true to keep the semantic equivalent.

In general, we will consider mutual independence.

Definition 358. If $\texttt{C}_2 \prec \texttt{C}_1$ and $\texttt{C}_1 \prec \texttt{C}_2$, we say that $\texttt{C}_2$ and $\texttt{C}_1$ are mutually independents, and write $\texttt{C}_1 \asymp \texttt{C}_2$.

While independence in one direction only, such as in the example above, does not imply that $\texttt{C}_1;\texttt{C}_2$ and $\texttt{C}_2;\texttt{C}_1$ have the same semantics, mutual independence allows to perform program transformation that do not impact the semantics.

Lemma 359 (Swapping commands (or chunks of commands)). *If* $\mathtt{C}_1 \asymp \mathtt{C}_2$*, then* $\mathtt{C}_1; \mathtt{C}_2 \equiv \mathtt{C}_2; \mathtt{C}_1$

Lemma 360 (Moving out of `while` loops). *If* $\mathtt{C}_1$ *is self-independent, i.e.* $\mathtt{C}_1 \asymp \mathtt{C}_1$*, and if* $\mathtt{C}_1 \asymp \mathtt{C}_2$*, then:*

$$\mathtt{while\ E\ do\ } [\mathtt{C}_1; \mathtt{C}_2] \equiv [\mathtt{if\ E\ then\ C}_1; \mathtt{while\ E\ do\ C}_2]$$

Based on those lemmas, we can decide that an entire block of statement is invariant or quasi-invariant in a loop by computing the DFGs. The quasi-invariance comes with an *invariance degree* wich is the number of time the loop needs to be peeled to be able to hoist the corresponding invariant. We can then implement program transformations that reduce the overall complexity while preserving the semantics. In the next section, we will explain how this was implemented on a "pycparser".

3 In practice

Unlike Song *et al.* [SFGH00] who worked on a *Single Static Assignment* form, we decided to work directly on `C` syntax mainly to more rapidly test our examples.

3.1 Preliminaries

To easily integrate our transformation, we decided to use "pycparser". The principal interest was to simply get and manipulate an Abstract Syntax Tree. As in compilers, a visitor allows us to do something every time a pattern is encountered. Using a "WhileVisitor" we list all nested `while`-loops. Using a bottom-up 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 first part aims to list relations. In our implementation we decided to define a relation object by three lists of pairs: one for the direct dependences, the second for the propagations and the last for reinitializations. A relation is computed for each command using a top-down strategy following the dominance tree. The relations are composed when the corresponding command is a sequence of commands. As describe previously, we compute the maximum relations possible for a `while` or `if` statement. For the `if` statement, we consider either if the condition is true or false. Obviously this method can be enhanced by an analysis on bounds around conditional and number of iterations for a loop. With those relations, we compute an invariance degree for each statement in the loop regarding to the relations listed. Finally, we perform the peeling transformation: we remove all Loop-Quasi-Invariants statements out of a given loop by peeling it.

3.2 Invariance degree computation

In this part, we will describe an algorithm – using the previous concepts – to compute the invariance degree of each quasi-invariant in a loop. After that, we will be able to peel the loop at once instead of doing it iteratively. To simplify and as a recall, Figure 8 shows a basic example of peeled loop.

```
Data: Dependency Graph and Dominance Graph
Result: List of invariance degree for each statement
Initialize every degrees to 0;
for each statement s do
    if the current degree cd $\neq 0$ then
        skip
    else
        Initialize the current degree cd to $\infty$;
        if there is no dependence for the current chunk then
            cd = 1;
        else
            for each dependence compute the degree dd of the command do
                if cd $\leq$ dd and the current command dominates this dependence then
                    cd = dd + 1
                else
                    cd = dd
                end
            end
        end
    end
end
```

Algorithm 2: Invariance degree computation.

```
while(x<100){
    b=b+1; //2
    use(b);
    x=x+1;
    b=y+y; //1
    use(b);
}
```

$\xrightarrow{\text{peeling}}$

```
if (x < 100) //1
{
  b_1= b+1;
  use(b_1);
  x = x+1;
  b = y+y;
  use(b);
}
if (x < 100) //2
{
  b_1= b+1;
  use(b_1);
  x = x+1;
  use(b);
}
while (x < 100)
{
  use(b_1);
  x = x+1;
  use(b);
}
```

Figure 8: Example: Hoisting twice.

The invariance degrees are given as comment in front of each Quasi-Invariant statements. So `b=y+y` is invariant of degree equal to one because `y` is invariant, that means it could be hoisted directly in the *preheader* of the loop. But `b` is used before, in `b=b+1`, so it's not the same `b` at the first iteration. We need to isolate this case by peeling one time the entire loop to use the first `b` computed by the initial `b`. If `b=y+y` is successfully hoisted, then `b` is now invariant. So we can remove `b=b+1` but we need to do it at least one time after the first iteration to set `b` to the new and invariant value. This is why the loop is peeled two times. The first time, all the statements are executed. The second time, the first degree invariants are removed. The main work is to compute the proper invariance degree for each statement and composed statements. This can be done statically using the dependency graph and dominance graph. Here is the algorithm. Let suppose we have computed the list of dependencies for all commands in a loop.

This algorithm is fast because it is dynamic. It stores progressively each degree needed to compute the current one and reuse them. Now that we did our analysis, we can peel the current loop.

3.3 Peeling loops

The transformation consists to create as many `if` statements before the loop as needed to remove all quasi-invariants out of the loop. Each `if` will have the same condition as the `while` loop and will contain the incrementation of the iteration variable. The maximum invariance degree is the number of time we need to peel the loop. So we can create as many `if` statements before the loop. For each `if` statement, we include every commands with a higher or equal invariance degree. For instance, the first `if` will contain every commands with an invariance degree higher or equal to 1, the second one, higher or equal to 2 etc... and the final loop will contain every commands with an invariance degree equal to ∞.

Renaming issue. On a non-SSA form, 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's possible to change de semantic. This issue is illustrated in Figure 8.

In this example, the variable `b` can have 4 different values and 3 of them are used by a non-pure function. We thus have to rename all variables assigned in the removed command, being careful with the first occurrence of the renamed variable: if it's an input variable we shall not rename it. We wrote the following algorithm 3.

Implementation details. This implementation is almost 400 lines of `Python`. It is able to compute relations of each commands or sequence of commands. The relations are stored in three different lists corresponding to each type of dependence described previously.

This tool focusses on a restricted `C` syntax and considers all function 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 allow to hoist pure functions as in [SFGH00]. All other side effects can be broken by

```
srand(time(NULL));
int n=rand()%10000;
int j=0;
while(j<n){
    fact=1;
    i=1;
    while (i<=n) {
        fact=fact*i;
        i=i+1;
    }
    j=j+1;
    use(fact);
}
```

peeling

```
srand(time(NULL));
int n = rand() % 10000;
int j = 0;
if (j < n)
{
  fact = 1;
  i = 1;
  while (i <= n)
  {
    fact = fact * i;
    i = i + 1;
  }
  j = j + 1;
  use(fact);
}
while (j < n)
{
  j = j + 1;
  use(fact);
}
```

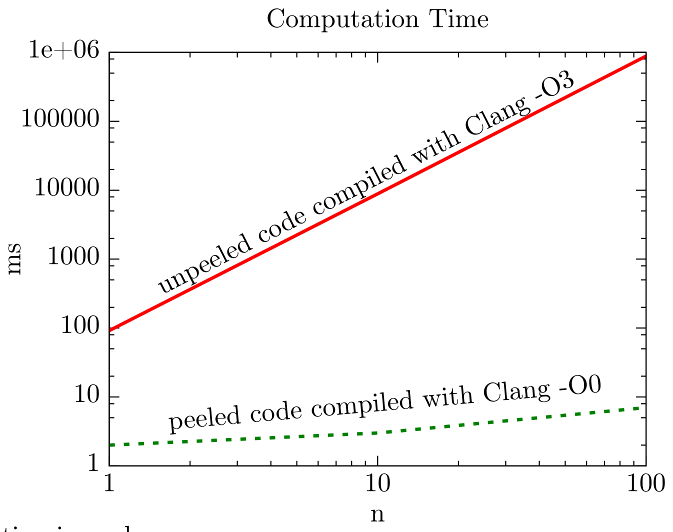


Figure 9: Hoisting inner loop

this transformation. Furthermore, this pass needs to use pointer analysis informations to trace more variables.

```
Data: The rank $i$ of the removed command
and the list of commands $C_1 \ldots C_n$
Result: Well renamed list of commands
for $j$ from 1 to $i$-1 do
    for all $x \in var(C_j)$ do
        if $x \in \mathrm{Out}(C_i)$ then
            if $x \in \mathrm{In}(C_j)$ and
            it's the first time this variable is encountered then
                Do nothing
            else
                Rename x by a fresh variable
            end
        end
    end
end
```

Algorithm 3: Renaming algorithm.

4 Conclusion and Future work

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 a corresponding loop optimization. With this proof-of-concept, we proved that we can enhance compilers by offering a new simple transformation pass.

In this example (Figure 9), 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 show 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` computes n times the inner loop so the time computation is increasing quadratically as shown on the right of Figure 9.

To provide some real benchmarks on large programs we need to implement it on real compilers. We are currently implementing this tool first in LLVM and secondly in GCC. Furthermore, the Intermediate Representation in a SSA form can simplify the analysis and the transformation, especially w.r.t. the renaming part.

We will take into account the others existing passes and choose the best order for the pass manager.

Acknowledgements

The authors wish to thank L. Kristiansen for communicating a manuscript [Kri] that initiated the present work. Jean-Yves Moyen is supported by the European Commision's Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 655222 - Walgo; Thomas Rubiano is supported by the ANR project "Elica" ANR-14-CE25-0005; Thomas Seiller is supported by the European Commision's Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 659920 - ReACT.

Co-author statements

December 6, 2016

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publication:

- *James Emil Avery and Jean-Yves Moyen and Pavel Ruzicka and Jakob Grue Simonsen.* "Chains, Antichains, and Complements in Infinite Partition Lattices". Under revision at Algebra Universalis.

Best regards,

[illegible]

James E. Avery

October 5, 2016

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publications:

- *Patrick Baillot and Ugo dal Lago and Jean-Yves Moyen.* "On Quasi-interpretations, Blind abstractions and Implicit complexity". Proceedings of the 8th International Workshop on Logic and Computational Complexity (LCC'06).
- *Patrick Baillot and Ugo dal Lago and Jean-Yves Moyen.* "On Quasi-interpretations, Blind abstractions and Implicit complexity". Mathematical Structure in Computer Science 22(4), pp. 549–580. Cambridge University Press 2012.

Best regards,

Patrick Baillot

[illegible]

October 28, 2016

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publication:

- *Guillaume Bonfante, Jean-Yves Marion and Jean-Yves Moyen.* “Quasi-Interpretations: a way to control Resources”. Theoretical Computer Science 412(25), pp. 2776–2796. Elsevier 2011.

Best regards,

Guillaume Bonfante [illegible]

October 5, 2016

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publications:

- *Patrick Baillot and Ugo dal Lago and Jean-Yves Moyen.* "On Quasi-interpretations, Blind abstractions and Implicit complexity". Proceedings of the 8th International Workshop on Logic and Computational Complexity (LCC'06).
- *Patrick Baillot and Ugo dal Lago and Jean-Yves Moyen.* "On Quasi-interpretations, Blind abstractions and Implicit complexity". Mathematical Structure in Computer Science 22(4), pp. 549–580. Cambridge University Press 2012.

Best regards,

[illegible]

Ugo dal Lago

October 28, 2016

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publication:

- *Guillaume Bonfante, Jean-Yves Marion and Jean-Yves Moyen.* "Quasi-Interpretations: a way to control Resources". Theoretical Computer Science 412(25), pp. 2776–2796. Elsevier 2011.

Best regards,

[illegible]

Jean-Yves Marion

December 6, 2016

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publications:

- *Jean-Yves Moyen and Thomas Rubiano.* "Detection of Non-Size Increasing Programs in Compilers". Developments in Implicit Computational Complexity, 2016.
- *Jean-Yves Moyen and Thomas Rubiano and Thomas Seiller.* "Loop Quasi-Invariant Chunk Motion by peeling with statement composition". Submitted to the 26th International Conference on Compiler Construction, 2017.

Best regards,

Thomas Rubiano

[illegible]

January 9, 2017

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publication:

- *James Emil Avery and Jean-Yves Moyen and Pavel Ruzicka and Jakob Grue Simonsen.* "Chains, Antichains, and Complements in Infinite Partition Lattices". Under revision at Algebra Universalis.

Best regards,

[illegible]

Pavel Ruzicka

December 6, 2016

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publication:

- *Jean-Yves Moyen and Thomas Rubiano and Thomas Seiller.* "Loop Quasi-Invariant Chunk Motion by peeling with statement composition". Submitted to the 26th International Conference on Compiler Construction, 2017.

Best regards,

Thomas Seiller

[illegible]

December 6, 2016

Co-author declaration

To whom it may concern

I declare that the authors contributed equally with regard to ideas and writing of the following publications:

- *James Emil Avery and Jean-Yves Moyen and Pavel Ruzicka and Jakob Grue Simonsen.* "Chains, Antichains, and Complements in Infinite Partition Lattices". Under revision at Algebra Universalis.
- *Jean-Yves Moyen and Jakob Grue Simonsen.* "More intensional versions of Rice's Theorem". Developments in Implicit Computational Complexity, 2016.
- *Jean-Yves Moyen and Jakob Grue Simonsen.* "More intensional versions of Rice's Theorem". Long version of previous article.
- *Jean-Yves Moyen and Jakob Grue Simonsen.* "Computability in the Lattice of Equivalence Relations". Work in progress.

Best regards,

2016-12-06

[illegible]

Jakob Grue Simonsen

Bibliography

[AA02] A. Abel and T. Altenkirch. A Predicative Analysis of Structural Recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.

[AAG$^+$08] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, Diana Ramírez, and Damiano Zanardini. The COSTA Cost and Termination Analyzer for Java Bytecode and its Web Interface (Tool Demo). In Anna Philippou, editor, *22nd European Conference on Object-Oriented Programming*, July 2008.

[AC03] D. Aspinall and A. Compagnoni. Heap Bounded Assembly Language. *Journal of Automated Reasoning (Special Issue on Proof-Carrying Code)*, 31:261–302, 2003.

[ACGZJ04] R. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 13th Annual Conference of the EACSL, Karpacz, Poland*, volume 3210 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2004.

[AJ94] N. Andersen and Neil D. Jones. Generalizing Cook's transformation to imperative stack programs. In J. Karhumäki, H. Maurer, and G. Rozenberg, editors, *Results and trends in theoretical computer science*, volume 812 of *Lecture Notes in Computer Science*, pages 1–18, 1994.

[AM08] M. Avanzini and G. Moser. Complexity analysis by rewriting. In *Proceedings of the 9th International Symposium on Functional and Logic Programming*, volume 4989 of *Lecture Notes in Computer Science*, pages 130–146. Springer-Verlag, 2008.

[AM13] M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and Usage. In *Proc. of 24th International Conference on Rewriting Techniques and Applications*, volume 21 of *Leibniz International Proceedings in Informatics*, pages 71–80, 2013.

[Ama03] R. Amadio. Max-plus quasi-interpretations. In *TLCA*, 2003.

[Ama05] R. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65:29–60, 2005.

[AMRS17] J. E. Avery, J.-Y. Moyen, P. Ruzicka, and J. G. Simonsen. Chains, Antichains and Complements in Infinite Partition Lattice. *Algebra Universalis*, 2017. Under revision.

[ASM13] M. Avanzini, M. Schaper, and G. Moser. Small Polynomial Path Orders in TcT. In *Proc. of 12th Workshop on Termination*, pages 3–7, 2013.

[Asp08] Andrea Asperti. The Intensional Content of Rice's Theorem. In *Proceedings of the 35th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, 2008.

[Ave06] James Avery. Size-change termination and bound analysis. In M. Hagiya and P. Wadler, editors, *Functional and Logic Programming: 8th International Symposium, FLOPS 2006*, volume 3945 of *Lecture Notes in Computer Science*. Springer, 2006.

[BA08] Amir M. Ben-Amram. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst.*, 30(3):1–31, 2008.

[BC92] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[BCMT01] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.

[BdLM06] P. Baillot, U. dal Lago, and J.-Y. Moyen. On Quasi-Iinterpretations, Blind Abstractions and Implicit Complexity. In *8th International Workshop on Logic and Computational Complexity (LCC'06)*, Seattle, United States, August 2006.

[BH02] Vladimir Markovich Blinovsky and Lawrence Hueston Harper. Size of the largest antichain in a partition poset. *Problems of Information Transmission*, 38(4):347–353, 2002.

[Bir40] Garrett Birkhoff. *Lattice Theory*, volume 25 of *Colloquium Publications*. American Mathematical Society, 1940.

[Blu67] M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):322–336, April 1967.

[BMM01] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On Lexicographic Termination Ordering with Space Bound Certifications. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Conference on Perspectives of Systems Informatics, 4th International Andrei Ershov Memorial Conference, PSI2001*, volume 2244 of *Lecture Notes in Computer Science*, pages 482–493, Akademgorodok, Novosibirsk, Russia, July 2001. Springer.

[BMM05] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-Interpretations and Small Space Bounds. In J. Giesl, editor, *Term Rewriting and Applications, 16th international Conference,(RTA 2005*, volume 3467 of *Lecture Notes in Computer Science*, pages 150–164, Nara, Japan, April 2005. Springer.

[BMM11] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, June 2011.

[BMP06] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux. A characterization of alternating log time by first order functional programs. In Springer, editor, *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 90–104, 2006.

[BMP07] *Quasi-interpretation synthesis by decomposition*, volume 4711 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[Bon00] G. Bonfante. *Constructions d'ordres, analyse de la complexité*. Thèse, Institut National Polytechnique de Lorraine, 2000.

[BPR96] S. Basu, R. Pollack, and M.-F. Roy. On the combinatorial and algebraic complexity of quantifier elimination. *Journal of the ACM*, 43(6):1002–1045, 1996.

[BT09] P. Baillot and K. Terui. Light types for polynomial time computation in lambda calculus. *Information and Computation*, 201(1):41–62, 2009.

[Can98] E. Rodney Canfield. The size of the largest antichain in the partition lattice. *Journal of Combinatorial Theory, Series A*, 83(2):188 – 201, 1998.

[Cas97] V.-H. Caseiro. *Equations for defining Poly-Time*. PhD thesis, University of Oslo, 1997.

[CKS81] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[Cob62] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.

[Coc70] John Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7), 1970.

[Col98] L. Colson. Functions versus Algorithms. *EATCS Bulletin*, 65:98–117, 1998. The logic in computer science column.

[Coo71] S. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18, January 1971.

[Czé96a] Gábor Czédli. Four-generated large equivalence lattices. *Acta. Sci. Math. (Szeged.)*, 62:47–69, 1996.

[Czé96b] Gábor Czédli. Lattice generation of small equivalences of a countable set. *Order*, 13(1):11–16, 1996.

[Czé99] Gábor Czédli. (1+1+2)-generated equivalence lattices. *Journal of Algebra*, 221(2):439–462, 1999.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science vol.B*, chapter Rewrite systems, pages 243–320. Elsevier Science Publishers B. V. (NorthHolland), 1990.

[DL07] U. Dal Lago. Notes on the Intensional Expressive Power of Bounded Calculi. unpublished note, http://www.cs.unibo.it/~dallago/iepbc.pdf, 2007.

[Eas70] William B. Easton. Powers of regular cardinals. *Annals of Mathematical Logic*, 1(2):139 – 178, 1970.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir98] J.-Y. Girard. Light linear logic. *Information and Computation*, 143:175–204, 1998.

[Grä03] George Grätzer. *General Lattice Theory*. Birkhäuser, second edition, 2003.

[Gri91] Daniel Grieser. Counting complements in the partition lattice, and hypertrees. *Journal of Combinatorial Theory, Series A*, 57(1):144–150, 1991.

[Har05] Egbert Harzheim. *Ordered Sets*, volume 7 of *Advances in Mathematics*. Springer-Verlag, second edition, 2005.

[Hau14] Felix Hausdorff. *Grundzüge der Mengenlehre*. Leipzig, 1914.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 185–197, New York, NY, USA, 2003. ACM.

[Hof92] D. Hofbauer. Termination proofs with Multiset Path Orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.

[Hof99] M. Hofmann. Linear types and Non-Size Increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.

[Hof00a] M. Hofmann. A type system for bounded space and functional in-place update. In *European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 165–179, 2000.

[Hof00b] Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 165–179, London, UK, UK, 2000. Springer-Verlag.

[HSW99] M. Holz, K. Steffens, and E. Weitz. *Introduction to Cardinal Arithmetic*. Birkhäuser Verlag, 1999.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[Jon97] Neil D. Jones. *Computability and Complexity, from a Programming Perspective*. MIT press, 1997.

[Jon99] N. D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 228:151–174, 1999.

[Jon00] N. Jones. The expressive power of higher order types or, life without cons. *Journal of Functional Programming*, 11(1):55–94, 2000.

[KdV03] Jan Willem Klop and Roer de Vrijer. *Term Rewriting Systems*, chapter Examples of TRSs and special rewriting formats. Cambridge University Press, 2003.

[KJ09] L. Kristiansen and N. D. Jones. The flow of data and the complexity of algorithms. *Transactions on Computational Logic*, 10(3), 2009.

[KKP$^+$81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL*, 1981.

[KL80] S. Kamin and J.-J. Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerité de l'Illinois, Urbana, 1980. Note non publiée.

[KN85] M. S. Krishnamoorthy and P. Narendran. On recursive path ordering. *Theoretical Computer Science*, 40(2-3):323–328, 1985.

[Kön05] Julius König. Über die Grundlage der Mengenlehre und das Kontinuumproblem. *Mathematische Annalen*, 61:156–160, 1905.

[Kri] L. Kristiansen. Notes on code motion. manuscript.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[Lei94] Daniel Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffery Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.

[Les92] P. Lescanne. Termination of rewrite systems by elementary interpretations. In H. Kirchner and G. Levi, editors, *3rd International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 1992.

[LJBA01] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The Size-Change Principle for Program Termination. In *Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, January 2001.

[LM93a] D. Leivant and J.-Y. Marion. Lambda Calculus Characterizations of Poly-Time. *Fundamenta Informaticae*, 19(1,2):167–184, September 1993.

[LM93b] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. In *Proceedings of Typed Lambda Calculi and Applicatipons (TLCA)*, volume 664 of *Lecture Notes in Computer Science*, pages 274–388. Springer-Verlag, 1993.

[LM95] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: Substitution and poly-space. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop, CSL '94*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500, Kazimierz, Pologne, 1995. Springer.

[Mar00] Jean-Yves Marion. Complexité implicite des calculs, de la théorie à la pratique, 2000. Habilitation.

[Mar03] Jean-Yves Marion. Analysing the implicit complexity of programs. *Information and Computation*, 183(1):2–18, 2003.

[Mat93] Y. V. Matiyasevich. *Hilbert's 10th Problem.* Foundations of Computing Series. The MIT Press, 1993. MAT y 93:1 1.Ex.

[MM00] J.-Y. Marion and J.-Y. Moyen. Efficient First Order Functional Program Interpreter with Time Bound Certifications. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasonning, 7th International Conference, LPAR 2000*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42, Reunion Island, France, November 2000. Springer.

[Moy03] J.-Y. Moyen. *Analyse de la complexité et transformation de programmes.* Thèse d'université, Nancy 2, December 2003.

[Moy08] Jean-Yves Moyen. Sct and the idempotence condition. Technical report, LIPN, May 2008.

[Moy09] J.-Y. Moyen. Resource Control Graphs. *ACM Transactions on Computational Logic*, 10(4), 2009.

[MP06] J-Y Marion and R. Péchoux. Resource analysis by sup-interpretation. In *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 163–176. Springer, 2006.

[MP09] Jean-Yves Marion and Romain Péchoux. Sup-Interpretations, a Semantic Method for Static Analysis of Program Resources. *Transactions on Computational Logic*, 10(3), 2009.

[MS] J.-Y. Moyen and J. G. Simonsen. Sublattices of Decidable Sets in the Lattice of Equivalence Relations on a Countable Set. Work in progress.

[MS55] John R. Myhill and John Cedric Shepherdson. Effective operations on partial recursive functions. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 1:310–317, 1955.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, January 1997.

[Ner58] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):pp. 541–544, 1958.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.

[NW06] K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM Journal on Computing*, 35(5):1122–1147, March 2006. published electronically.

[Ore42] Øystein Ore. Theory of equivalence relations. *Duke Mathematical Journal*, 9(3):573–627, 1942.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Pla78] D. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical Report R-78-943, Department of Computer Science, University of Illinois, 1978.

[Reu89] C. Reutenauer. *Aspects mathématiques des réseaux de Petri.* Masson, 1989.

[Ric53] Henry Gordon Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967. Reprint, MIT press 1987.

[RS92] Ivan Rival and Miriam Stanford. Algebraic aspects of partition lattices. In Neil White, editor, *Matroid Applications*, volume 40 of *Encyclopedia of Mathematics and its Applications*, pages 106–122. Cambridge University Press, 1992.

[Sav70] W. J. Savitch. Relationship between nondeterministic and deterministic tape classes. *JCSS*, 4:177–192, 1970.

[Sch07] Ulrich Schopp. 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, 2007. IEEE Computer Society.

[Sen95] G. Senizergues. Some undecidable termination problems for semi-thue systems. *Theoretical Computer Science*, 142:257–276, 1995.

[SFGH00] Litong Song, Yoshihiko Futurama, Robert Glück, and Zhenjiang Hu. A loop optimization technique based on quasi-invariance. 2000.

[Sha56] Normann Shapiro. Degrees of computability. *Transactions of the AMS*, 82:281–299, 1956.

[Sie22] Wacław Franciszek Sierpiński. Sur un problème concernant les sous-ensembles croissants du continu. *Fundamenta Mathematicae*, 3:109–112, 1922.

[Smu58] Raymond M. Smullyan. Undecidability and recursive inseparability. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 4(7–11):143–147, 1958.

[SS63] J.C. Shepherdson and H.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963.

[Tar51] A. Tarski. *A Decision Method for Elementary Algebra and Geometry, 2nd ed.* University of California Press, 1951.

[Wei95] Andreas Weiermann. Termination proofs by Lexicographic Path Orderings yield multiply recursive derivation lengths. *Theoretical Computer Science*, 139:335–362, 1995.