Mathematical Informatics

Thomas Seiller

Mathematical Informatics

Outline of a mathematical theory of computer science

Thomas Seiller

Defended on June 18th 2024

Composition of the jury

Valérie Berthé	CNRS & Université Paris Cité	Examinatrice
Anuj Dawar	University of Cambridge	Rapporteur
Christian Ikenmeyer	University of Warwick	Rapporteur
Ugo dal Lago	University of Bologna	Rapporteur
Meena Mahajan	Institute of Mathematical Science	Examinatrice
Nabil Mustafa	Sorbonne Paris Nord University	Examinateur
Joël Ouaknine	Max Plank Institute	Rapporteur
Laure Petrucci	Sorbonne Paris Nord University	Examinatrice

Acknowledgments

I want to thank the reviewers of this thesis for accepting this task and the members of the jury for accepting to play this role. I thank them for taking the time to read this document even though some parts were necessarily far from their domain of expertise.

Before thanking the many researchers that have had an impact on my work, I want to thank our most essential colleagues, that is the administrative staff. Without them, we would not be able to do our job as researchers. They bear the weight of the overcomplicated procedures, they manage to make things work even if when they do not have enough time for this, and they somehow rarely get to be told that they are essential. This is why I want to thank them first, because without their dedication, I would not have had the occasion to meet and profit from my interactions with all the persons listed below. Thanks to Aimé Bayonga, Ali Zaman, Anne-Claire Binetruy, Antonia Wilk, Brigitte Guéveneux, Christopher Sturrock, Estelle Hutschka, Fatima Abbas, Helle Norup, Jaime Arias, Jorge Garcia Flores, Laure Thiébault, Mamadou Sow, Marc Lavaux, Marie Fontanillas, Marisol Rodriguez, Odile Ainardi, Omar Kebli, Véronique Criart.

During my time as a PhD, then postdoc, then CNRS researcher, I have been surrounded by many people who have played essential roles. Among those, I wish to make three names stand out for the particularly huge impact they had on me and my work: Laurent Regnier, Tom Hirschowitz, and Jakob Grue Simonsen. They have shaped me into the scientist I am today more than they think, they have taught me much more than just mathematics. I learnt to question the most basic assumptions on which our work relies, I learnt that a seemingly naive question can shed light on fundamental asperities of a seemingly perfectly smoothed theory, I learnt that being a researcher is still just a job and that it should not take over more important aspects of our lives, I learnt that one can still succeed while staying curious in research, wandering in other fields, enjoying research without constraining to a single domain of expertise, without caring about the cost it automatically incurs (because – it is well established – interdisciplinarity has a cost, even when remaining within computer science).

I also wish to thank more specifically the students that chose to embark on a PhD or a postdoc under my (usually shared) supervision. They trusted me, and I hope that I have been (or will prove to be) worthy of it. My discussions with them, whether scientific or not, whether or not they ended on an agreement, enriched my work more than anything else. They forced me to questions choices (whether mine or others) and either justify or (reconsider) them. While most of the results I selected to write in this document were not result of joint work with them, they contributed fully to the fabrication of the vision I perfected and I am now trying to convey. At those times when my administrative duties were taking over almost all my working time, my weekly interactions with them have kept me close to the research. They are: Adrien Ragot, Alexandre Lucquin, Boris Eng, Caterina Mosca, Derek So, Eliès Harington, Lê Thành Dũng (Tito) Nguyễn, Maxime Lucas, Mohamed Maizia, Morgan Rogers, Paul Séjourné, Pierre-Yves Coursolles, Seng Beng Goh, Thomas Rubiano, Ulysse Léchine, Valentin Maestracci, William Troiani, Yasmine Laghjichi.

Another group of people that have shaped my understanding of theoretical computer science, its numerous subdomains with their specific questions and techniques, their differences in how they function, how they differ in their habits of publication, their criteria. These people are the members of the section 06 of CoNRS, which I joined in January 2021, continuing for the next mandate and acting as scientific secretary from September 2021; in (first name) alphabetical order: Alain Tchana, Anastasia Paparrizou, Anca Muscholl, Antoine Genitrini, Arnaud Legrand , Christophe Rey, Clarisse Dhaenens, Damian Markham, Dominique Lavenier, Gilles Villard, Hélène Waeselynck, Hubert Comon-Lundh, Igor Walukiewicz, Jérémie Bourdon, Johanne Cohen , Katia Jaffrès-Runser, Laurence Duchien, Leo Liberti, Maria Potop-Butucaru, Michael Poss, Nathalie Appel, Nathalie Gilles, Nicolas Bousquet, Pablo Arrighi, Patricia Georgeon, Philippe Owezarski, Pierre Aboulker, Pierre Clairambault, Pierre Senellart, Pierre Sens, Romain Rouvoy, Sandrine Blazy, Simon Perdrix, Valérie Berthé, Ye-Qiong Song, Yolande Sallent, Yves Grandvalet.

In a similar way, the members of the interdisciplinary commission 53 of CoNRS have expanded my understanding of science beyond informatics. While sometimes challenging, working as part of such an interdisciplinary committee was (and still is) extremely enriching. I therefore thank in (first name) alphabetical order: Baptiste Mélès, Cedric Paternotte, Charlotte Bigg, Clément Bosquet, Emanuel Bertrand, Fanny Meunier, Francoise Immel, Frédéric Keck, Hervé Pennec, Isabelle Krzywkowski, Jean-Noël Jouzel, Lorenzo Barrault-Stella, Lucie Laplane, Natasha Collomb, Nathalie Tanchoux, Pablo Jensen, Patrice Abry, Patrick Blanco, Perig Pitrou, Pierre Blavier, Rodolphe Defiolle, Sara Angeli-Aguiton.

Beyond those, I would like to thank all researchers I have met and interacted with during the (now somewhat numerous) years since I embarked on my PhD, in the context of collaborations, scientific discussions (sometimes short, but always impactful), organisation tasks, administrative tasks, etc. Obviously, this includes everyone from the different communities I have enjoyed contributing to and I cannot hope to provide an exhaustive list, especially given my tendency to disperse. I will nevertheless try to name most of them (and if you are not part of the list and should be, I will happily add you): Adeline Nazarenko, Adrien Champougny, Adrien Guatto, Adrienne Lancelot, Afonso Ferreira, Alberto Naibo, Alejandro Díaz-Caro, Alex Bredariol Grilo, Alexander Shen, Alexandre Dupont-Bouillard, Alexis Saurin, Aloys Dufour, Amaury Pouly, Anastasia Volkova, André Seznec, Andreas Sportiello, Andrei Romashchenko, Anne Siegel, Anupam Das, Assia Mahboubi, Aude Gretzka, Aurore Alcolei, Axel Kerinec, Baptiste Chanus, Beniamino Accattoli, Benoit Valiron, Carola Doerr, Caroline Collange, Céline Rouveirol, Chantal Keller, Christian Jutten, Christine Tasson, Christophe Fouqueré, Christophe Raffalli, Christophe Tollu, Claudia Faggian, Clément Aubert, Clovis Eberhart, Colin Riba, Cynthia Kop, Damiano Mazza, Damien Pous, Daniel Hirschkoff, Daniel Murfet, Davide Barbarossa, Delia Kesner, Denis Merigoux, Dimitri Ara, Edgar Lejeune, Edwige Cyffers, Emmanuel Beffara, Emmanuel Hainry, Emmanuel Haucourt, Erven Rohou, Etienne André, Etienne Miquey, Etienne Moutot, Federico Olimpieri, Flavien Breuvart, Florent Koechlin, Florian Jaton, Frederique Bassino, Gabriel Scherer, Gilles Trédan, Giulio Manzonetto, Giuseppe Primiero, Guilhem Jaber, Guillaume Bonfante, Guillaume Geoffroy, Guillaume Malod, Guillaume Melquiond, Guillaume Munch-Maccagnoni, Guillaume Theyssier, Henri Stephanou, Hong-Linh Le, Hugo Férée, Hugo Herbelin, Hugo Paquet, Isabelle Puaut, Jaco van de Pol, James Avery, Jean-Baptiste Joinet, Jean-Yves Marion, Jean-Yves Moyen, Jérôme Lang, Joanna Ochremiak, John Terilla, Jonas Frey, Joseph Ben Geloun, Juan-Luis (Gianni) Gastaldi, Julien Cervelle, K. V. Subrahmanyam, Karine Chemla, Kasper Hornbæk, Kazushige Terui, Kostia Chardonnet, Krzysztof Worytkiewicz, Ksenia Ermoshina, Ksenia Tatarchenko, Laetitia Laversa, Lars Kristiansen, Laura Fontanella, Lionel Pournin, Lionel Vaux Auclair, Lorenzo Tortora de Falco, Luc Pellissier, Luidnel Maignan, Luiz-Carlos Pereira, Łukasz Czajka, Marc de Visme, Marco Panza, Marianna Antonutti Marfori, Marie Alauzen, Marie-Christine Rousset, Marylou Le Roy, Massimo Airoldi, Matteo Acclavio, Matthieu Lacroix, Mattia Petrolo, Melissa Antonelli, Meven Lennon-Bertrand, Michael Walter, Michaela Mayero, Michele Pagani, Mikaël Monet, Mitsuhiro Okada, Mohand-Saïd Hacid, Myriam Quatrini, Natasha Portier, Nathalie Pernelle, Neea Rusch, Neeraj Kayal, Neil D. Jones, Neil (Julien) Ross, Nguyen Viet Hung, Nicolas Tabareau, Noam Zeilberger, Nutan Limaye, Olivier Bodini, Olivier Bournez, Olivier Laurent, Olivier Serre, Paolo Pistone, Pascal Vannier, Pascal Weil, Patrick Baillot, Paul-André Melliès, Paula Quinon, Paulin de Naurois, Perceval Pillon, Pernille Bjørn, Peter Selinger, Philippos Papagiannopoulos, Pierre Boudes, Pierre Depaz, Pierre Fouilhoux, Pierre Guillon, Pierre Hyvernat, Pierre Ohlmann, Pierre Valarcher, Pierre Wagner, Pierre-Evariste Dagand, Pierre-Louis Curien, Pierre-Marie Pédrot, Raphaëlle Crubillé, Rayya Roumanos, Roberto Maieli, Roberto Wolfler Calvo, Rodolphe Lepigre, Samantha Jarvis, Samuel Mimram, Sarah Lawsky, Sébastien Tavenas, Shin-Ya Katsumata, Siddarth Bhaskar, Simon Mirwasser, Simone Martini, Sonia Marin, Sophie Huiberts, Sophie Toulouse, Srikanth Srinivasan, Stefano Guerrini, Sylvain Perifel, Théo Winterhalter, Thierry Joly, Thierry Monteil, Thomas Ehrhard, Thomas Rubiano, Tiphaine Viard, Titouan Carette, Valentin Blot, Valeria Vignudelli, Virgile Mogbil, Visu Makam, Walter Dean, Yuri Gurevich, Yves Lafont, Zeinab Galal.

Obviously, the most important support of all is that of my family. I thank my two children for joyfully precluding me from working outside of ~~working~~ school hours. Their presence made me take most (if not all) my week-ends off in such a way that it did not even count as resting time. They entered my life with a ferocious energy deflecting me from work. While I cannot thank them for the contents of this documents, I

owe them much more: all these beloved moments outside of work which belittle whatever I wrote in those pages.

Finally, I wish to thank Marie*. While she has filled my non-working time with priceless memories, she also helped me in achieving the results here in so many ways that I cannot start recounting them. As such, she supported fully both this work and everything else that was not this work. I would not have been able to ~~finish~~ start writing this document without her support. But more importantly: I would not have the opportunity to say that this manuscrit, which I am proud of†, may in fact be the part of my life that I am the least proud of.

* I have waited long enough, right?

† At least while I am finishing to write it; I will surely find many defaults in it as the time comes.

Contents

1. Genesis

In this section, I want to do the exercise of retracing the flow of production of the material contained in later section. For (good) pedagogical reasons, the presentation completely messes up the causality relations, hiding the process behind the research. This section will try to expose this process, at the same time making clear the fact that what has been written corresponds to a snapshot of an object in movement. A picture of the sea and waves, as the one on the title page, can show very clearly some details, but it will always keep some parts hidden; the same picture taken a few seconds later could be incomparable. It is the same here: this document has been written at a very specific moment in time, and represents a still picture of an ever-moving object. Although it may not be enough, I try in this section to uncover – or at least sketch – the hidden movement of the waves underneath made invisible by the format of a written report.

I am by training (by design) a mathematician. But I have found myself thinking more and more as a computer scientist over the years. The first part of the document somehow embraces that change. It takes its origins in a criticism of the theory of computability: that it was designed by mathematicians to answer mathematical questions about mathematical objects. As such, it can be argued that it does not provide a proper foundational theory for computer science. In particular, it does not define the fundamental notions studied by the latter: models of computation, programs, algorithms, etc.

I have found over the years that some of the structures that were underlying a large part of my research could lead to another approach to computability, one that does not focus on **which** functions are computable but rather on **how** those can be computed. All the material in the corresponding chapters (from chapter 3 to chapter 6) is new. In those, I present basic notions and examples. But this should be understood as a first approximation of a general theory to be detailed and developed further in future work. To mimic a well-known book title in category theory, this is just a preliminary sketch of a very large animal, but some underwater, unseen creature, like a giant squid (but a nice and welcoming one) with far-reaching tentacles disappearing in the dark. I still expect this sketch to provide a sound and solid basis for future extensions.

The second part of the document contains a selection of the work I have done since my PhD, presented in light of the first part. While the contents of the first part have taken shape gradually over the years, with some preliminary definitions appearing in the most recent published papers, the underlying point of view has always been present – though in a vague and intangible way – and has had an impact on my research. The corresponding chapters correspond to published work presented – most of the time – synthetically. New material nevertheless appears in these chapters. In particular:

- this document explains the connection between the dynamic point of view on computability presented in the first part and the de-

pendency analyses originally inspired by Jones and Kristiansen's *mwp-flows* analysis [2];
- the connection between the graphings interpretation of programs and the operator-theoretic *geometry of interaction* construction of Girard is detailed for the first time;
- the linear realisability models I have worked on for years are presented in a more axiomatic way, extracting the essential properties and providing generic constructions (which specialise to the several models of linear logic I have been working on in the past).

[2]: Jones et al. (2009), *A Flow Calculus of Mwp-bounds for Complexity Analysis*

The last part is about (some of) the roads lying ahead of me, or at least the ones that I cherish the most at the moment. The first has been a motivating idea for several years now, while the second is new, and arose while writing Part 1. The second takes a step further in criticising the ever-widening gap between the theory of computation and computing itself, and opens a road (surely dangerous) to try and bridge this gap. Obviously, those two directions should be complemented with fundamental work to extend the first part of this document.

Before entering the core part of this document, I present in the following sections the flow of ideas that have led to the results presented here in a chronological way. I also take this opportunity to summarise some of the results that have not made their way in the document.

Background: Curry-Howard

In the background of my work lies the proofs-as-programs (or Curry-Howard) correspondence. This correspondence states that notions in computer science (more specifically lambda-calculus or functional programming languages) correspond to notions in proof theory (a subfield of mathematical logic that studies the notion of proof as mathematical objects). The correspondence's core can be expressed as below:

Computer Science	Logic
Type	Formula
Program	Proof
Execution	Cut elimination

This correspondence has been extended in many ways, and is essential in the fields of Semantics of Programming Languages and the Theory of Programming Languages. Among the outcomes of the Curry-Howard correspondence, we find refinements of logical systems accounting for specific aspects of computation: linear logic for instance accounts for the notion of resources, differential linear logic accounts for a notion of differentiation of programs. These refinements and extensions usually arise from a careful study of *models*: the normal functor model [3] for linear logic [4], Köthe spaces [5] for differential linear logic [6]. Here the notion of model is synonymous with *denotational models* or even *categorical models*. The above correspondence may be extended to the so-called Curry-Howard-Lambek correspondence by adding category theory in the picture.

[3]: Girard (1988), *Normal functors, power series and λ-calculus*

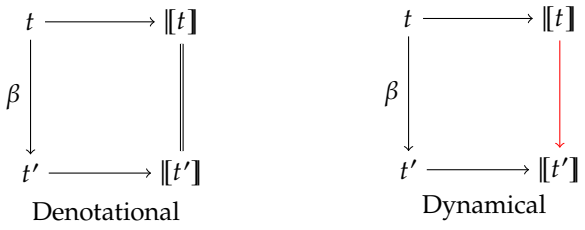
[4]: Girard (1987), *Linear logic*

[5]: Ehrhard (2002), *On Köthe Sequence Spaces and Linear Logic*

[6]: Ehrhard et al. (2006), *Differential interaction nets*

Computer Science	Logic	Category Theory
Type	Formula	Object
Program	Proof	Morphism
Execution	Cut elimination	Identity

However, as one can see, the category-theoretic perspective (at least when restricted to 1-categories) trivialises the level of dynamics. In fact, denotational semantics more generally consists in equating a program given an argument and the result of its execution. Intuitively, this is the same as identifying the expression $4^2 + 7^4$ with 2417. Formally, in lambda-calculus this corresponds to identifying a λ-term t with its *normal form* t' (w.r.t. β-reduction). My work has been focussed from the beginning on a different family of mathematical models of programs in which execution is not represented as the identity but as a non-trivial mathematical operation.



This approach is very close to *game semantics* for programming languages[1] and more recent work defining 2-categorical (or higher) semantics (an extension of categories in which one has a notion of 2-morphisms between morphisms) but differs in the employed techniques. While those start from the notion of type, from which one gets a notion of *typed* program, the models I have worked on are usually constructed the other way: we start from a set of untyped programs, and show that they naturally possess the structure to define types[2]. I now use the terminology *Linear Realisability models* to describe such models, in order to emphasise their connection with realisability (both in the standard sense of Kleene and in the sense of Krivine's classical realisability).

1: In fact game semantics originate in the earlier models of this type.

2: This point of view on *types as descriptors* or *classifiers*, as opposed to the traditional view of *types as constraints* have been explored in a work with Jean-Baptiste Joinet mentioned below (subsection 2.6).

Linear realisability and von Neumann algebras

My PhD was about the newest (at the time) linear realisability construction introduced by Jean-Yves Girard [7]. This model was exploiting a previous result [8] that established that the functional equation (Equation 1.1) corresponding to cut-elimination in linear logic (or β-reduction in lambda-calculus) always had a solution when the operators considered were taken in the unit ball of a von Neumann algebra. This quite technical result consisted in showing that the concrete solution, called the execution formula, which works for operators of norm strictly less than 1, could be extended by continuity without breaking associativity. The 'geometry in the hyperfinite factor' construction then exploited this extended solution to propose a realisability model whose underlying set of 'programs' is the set of all operators in the unit ball of a von Neumann algebra.

[7]: Girard (2011), *Geometry of Interaction V: Logic in the Hyperfinite Factor.*

[8]: Girard (2006), *Geometry of Interaction IV: the Feedback Equation*

Traditionally, a program in geometry of interaction is represented as an operator[3] acting on a Hilbert space $\mathbb{I} \oplus \mathbb{O}$, which is used to represent

3: Other presentations exists, in particular based on *flows*, but this change in the presentation of objects turns out to be mostly aesthetic: all models can be presented in an operator-theoretic setting. This is done for instance in my PhD thesis [9] (in french).

inputs $\mathbb{I}$ and outputs $\mathbb{O}$. Given such a representation P of a program and a representation $I \in \mathbb{I}$ of an input, the representation of the computed output O is defined as the solution to a functional equation, named the *feedback equation*.

$$O(\xi) = \xi' \Leftrightarrow \exists \eta, \eta' \in \mathbb{I}, \begin{cases} P(\eta \oplus \xi) &= \eta' \oplus \xi' \\ I(\eta') &= \eta \end{cases} \tag{1.1}$$

It was then shown by Girard [8] that if one restricts to operators P, I belonging to the unit ball of a von Neumann algebra $\mathfrak{M}$, this equation always has a solution[4], i.e. there is an operator O satisfying this equation, and O belongs to the unit ball of $\mathfrak{M}$. This lead to the introduction of the *geometry in the hyperfinite factor*, or GoI5, construction [7] in which a linear realisability model of (a fragment of) linear logic is constructed over the unit ball of the hyperfinite factor of type II_∞. This was a large generalisation of previous geometry of interaction models, in which the underlying set was restricted to a subset of operators in a C*-algebra generated by a few simple operators. This had two main impacts:

- the notion of orthogonality, traditionally based on nilpotency, needed to be modified; for this, Girard introduced a notion based on the Fuglede-Kadison determinant [10], a generalisation of the determinant of matrices to operators in (some) von Neumann algebras;
- the notion of *proof-like* operators, i.e. those operators that can be the interpretation of proofs, required to be defined with respect to a maximal abelian von Neumann subalgebra [11] – called a *viewpoint* –, representing a chosen basis for the underlying space.

My work was concerned with both these aspects. Concerning the first, it lead me to introduce a combinatorial version of linear realisability constructions motivated by a combinatorial interpretation of the determinant. The corresponding work on *Interaction graphs*, will be presented later in this document. My work on the second aspect led me to what remains to this day my own favourite result showing a connection between a classification of maximal abelian von Neumann algebras $\mathfrak{A}$ due to Dixmier [12] and the fragment of linear logic that one can interpret soundly with respect to the viewpoint $\mathfrak{A}$. This result is presented in chapter 11.

[8]: Girard (2006), *Geometry of Interaction IV: the Feedback Equation*

4: And that this solution is order-continuous and satisfies an *associativity property* [8].

[7]: Girard (2011), *Geometry of Interaction V: Logic in the Hyperfinite Factor.*

[11]: Seiller (2018), *A Correspondence between Maximal Abelian Sub-Algebras and Linear Logic Fragments*

[12]: Dixmier (1954), *Sous-anneaux abéliens maximaux dans les facteurs de type fini*

A guiding result

This result (Theorem 11.1.27) turned out to be guiding result for my subsequent work. Indeed, previous and subsequent work showed the obtained geometry of interaction (GOI) model interprets, depending on the choice of the von Neumann algebra $\mathfrak{M}$ and the viewpoint $\mathfrak{A}$, either full linear logic [13], the constrained system Elementary Linear logic (ELL) which characterises elementary time computable functions [9, 11, 14], or smaller fragment such as multiplicative additive linear logic (MALL). This naturally leads to the following informal conjecture.

Informal Conjecture 1 There is a correspondence between complexity classes and pairs $(\mathfrak{A}, \mathfrak{N})$ of a von Neumann algebra $\mathfrak{N}$ and a maximal abelian von Neumann subalgebra $\mathfrak{A} \subset \mathfrak{N}$.

[13]: Girard (1995), *Geometry Of Interaction III: Accommodating The Additives*

[9]: Seiller (2012), *Logique dans le facteur hyperfini : géometrie de l'interaction et complexité*

[11]: Seiller (2018), *A Correspondence between Maximal Abelian Sub-Algebras and Linear Logic Fragments*

[14]: Seiller (2019), *Interaction Graphs: Exponentials*

But once this informal conjecture formulated, how could one try to establish or refute it? The framework is in fact extremely difficult to exploit further for two reasons. First, the theory of maximal abelian sub-algebras in von Neumann algebras is an involved subject matter still containing large numbers of basic but difficult open problems [15]. Second, even though some results were obtained, no intuitions would be gained about how the choice of the couples $(\mathfrak{A}, \mathfrak{N})$ actually restricts the computational power of the programs represented in the models.

[15]: Sinclair et al. (2008), *Finite von Neumann algebras and Masas*

From operators to graphings

It is while considering this (seemingly) dead-end that I realised that my other work relating to realisability models of linear logic, *Interaction graphs models*, could provide an alternative, more tractable, approach. Interaction graphs can be understood as a concrete definition of (some) operators in a specified von Neumann algebra considered with a specific abelian sub-aglebra.

Indeed, I had shown in earlier work [16, 17] how one can construct models of MALL where proofs are interpreted as graphs. This construction relied on a single property, the *trefoil property*, which relates two simple notions:

- the *execution* $F :: G$ of two graphs, a graph defined as a set of paths;
- the *measurement* $[\![F, G]\!]_m$, a real number computed from a set of cycles.

[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*
[17]: Seiller (2016), *Interaction graphs: Additives*

A more general construction was then built upon a generalization of graphs [18], named *graphings* [19–21].

[18]: Seiller (2017), *Interaction Graphs: Graphings*
[19]: Adams (1990), *Trees and amenable equivalence relations*
[20]: Levitt (1995), *On the cost of generating an equivalence relation*
[21]: Gaboriau (2000), *Coût des relations d'équivalence et des groupes*

Graphings can be understood either as *geometric realisations* of graphs on a measure space $(X, \mathcal{B}, \mu)$, as measurable families of graphs, or as generalized measured dynamical system. The notion is parametrized by a monoid describing the model of computation and a map describing the realisability structure:

- a monoid $\mathfrak{m}$ of measurable maps[5] from $(X, \mathcal{B}, \mu)$ to itself;
- a map $m : \Omega \to \bar{\mathbf{R}}_{\geqslant 0}$ defining *orthogonality* – accounting for linear negation.

5: In practice, one requires those maps to be bimeasurable non-singular transformations, i.e. a measurable map f which sends measurable sets to measurable sets, and such that $f(A)$ is negligible if and only if A is negligible.

An $\mathfrak{m}$-*graphing* is then defined as a directed graph F whose vertices are measurable subsets of the measurable space $(X, \mathcal{B})$, and whose edges are *realised* by elements of $\mathfrak{m}$, i.e. for each edge e there exists an element ϕ_e in $\mathfrak{m}$ such that $\phi_e(s(e)) = t(e)$, where s, t denote the source and target maps[6]. Based on this notion, and an orthogonality relation defined from the map m, I obtained a systematic method for constructing realisability models for linear logic [18]. This is presented in chapter 10.

6: In practice, graphings are defined without target maps, and we deduce $t(e)$ from ϕ_e and $s(e)$.
[18]: Seiller (2017), *Interaction Graphs: Graphings*

But at the core of the construction, one finds a monoid action $\alpha : \mathfrak{m} \to \mathrm{End}(X, \mathcal{B}, \mu)$ which can be related to the operator-theoretic setting in the specific case in which $\mathfrak{m}$ is a group of measure-preserving maps. Indeed, by a standard construction in the theory of von Neumann algebras called the *group measure space construction*, a measure-preserving group action $\alpha : \mathfrak{m} \to \mathrm{End}(X, \mathcal{B}, \mu)$ gives rise to a pair $\mathfrak{A} \subset \mathfrak{N}$ of a von Neumann algebra $\mathfrak{N}$ and a maximal abelian von Neumann subalgebra $\mathfrak{A}$. The construction works as follows:

- one considers the Hilbert space $\mathbb{K} = L^2(X, \mathcal{B}, \mu) \otimes L^2(\mathfrak{m})$;
- $\mathfrak{A}$ is defined as the algebra $L^\infty(X, \mathcal{B}, \mu)$ of essentially-bounded complex-valued functions on X, represented on $\mathbb{K}$ by left multiplication on $L^\infty(X, \mathcal{B}, \mu)$;
- the algebra $\mathfrak{R}$ is generated by operators on $L^2(X, \mathcal{B}, \mu) \otimes L^2(\mathfrak{m})$ induced by the elements of $\mathfrak{m}$.

Intuitively, graphings are therefore[7] a description of operators in $\mathfrak{R}$ that preserve $\mathfrak{A}$ in some way (it can be shown that operators induced by $\mathfrak{R}$ are in the normalising groupoid of $\mathfrak{A}$). They are therefore concrete realisations of some operators in $\mathfrak{R}$, leading to the following reformulation of the informal conjecture.

7: When one restricts to the case of group actions by measure-preserving transformations; the general case of a monoid action should thus be considered as a generalisation of this situation in the case the measure-space construction is not applicable.

Informal Conjecture 2 There is a correspondence between complexity classes and monoid actions $\alpha : \mathfrak{m} \to \mathrm{End}(X, \mathcal{B}, \mu)$, where $(X, \mathcal{B}, \mu)$ is a measure space.

This informal conjecture have then been the main motivation behind many results over the years, as I was trying to answer the following questions:

1. Can we establish this correspondence for specific complexity classes and monoid actions? The results in that direction are presented in (chapter 11);
2. Can invariants for the monoid action be used to prove that two complexity classes are not equal? This lead me to investigate Mulmuley's geometric complexity theory programme, as well as trying to understand numerous lower bounds techniques. This lead in particular to the results presented in chapter 8, where we show how topological entropy can be leveraged to obtain known (and some new) lower bounds results.
3. Can the result about entropy mentioned in item 2 be *lifted* to the realisability models to be applied to the classes characterisations mentioned in item 1? This lead me to investigate the structure of the linear realisability models, and in particular the orthogonality relation. This resulted in exposing how the latter is related to *zeta functions* for graphs [22] and dynamical systems [23] (chapter 9). This question is also part of the perspectives for future work, presented in chapter 12.

[22]: Ihara (1966), *On discrete subgroups of the two by two projective linear group over $\mathfrak{p}$-adic fields*

[23]: Ruelle (1976), *Zeta-functions for expanding maps and Anosov flows*

An ontology of computer science

As part of my work on these questions, I have come to the realisation that underlying the approach lies a mathematical theory of computers, programs, and more. In the last two years, I have therefore started investigating this aspect independently from the questions above. In particular, I started working with philosophers to understand if this point of view could lead to a new perspective on questions such as: what is computation? what is a model of computation? what is a program?

I realised that while computability theory offers a nice theory for talking about computable functions, it is in the end just about that: which functions are computable. It does not define what a program is, or what an algorithm is, and one may ask: can computer science be a science on its own if it cannot define properly the objects it studies? These interactions

with philosophers were extremely enriching, from the discussions we had to the papers – tackling those exact questions – we read together.

The first part of this document presents computability from this alternative point of view in details. I propose definitions of models of computations, machines, programs, data structures and their representations, complexity, algorithms. These definitions, and the overall principle of putting forward the *dynamics* of computer programs, are directly inspired from the specific approach to semantics presented above. But they represent a new, separate, line of work. This is a huge project, still a work in progress. But I believe I have now gathered and arranged enough stones to ensure proper foundations.

The second part of the document then presents all the results mentioned previously (and some others) in a coherent and homogenised way. However, before diving into the technical details, the next chapter provides a complete view on all the work I have done, in particular summarising some results that will not be detailed later in the document.

2. A complete timeline

This timeline is organised thematically rather than chronologically. I however tried to provide some information on the connections and motivations between results when those could be explained easily. The different themes I have identified are:

1. Semantics (section 2.1)
2. Implicit Computational complexity (section 2.2)
3. Computability and algebraic complexity (section 2.3)
4. Automata and randomness (section 2.4)
5. Verification (section 2.5)
6. Philosophy of logic and philosophy of computer science (section 2.6)

2.1. Semantics

A consequent part of my work since my PhD thesis concerns contributions in and around these areas. I separate these contributions into two groups: one that concerns the historical (denotational) model that lead to the introduction of linear logic, and a second that concerns linear realisability models. The latter part being discussed further down in the document, I only quickly summarise the former.

Denotational semantics

Even if my main interest lies in *dynamic semantics*, and particularly linear realisability models, I have also worked in the last years on denotational semantics. Both works originate in the *normal functors* model of lambda-calculus from which linear logic originates. This model is based on heavy category theory. It represents formulas/types as discrete presheaves categories, i.e. categories of functors from a set A (viewed as a category) to the category of sets. Proofs/programs are then interpreted as functors between those presheaves categories. The main takeout from the model is that to interpret lambda-calculus, one can restrict to so-called *normal functors*. Those are defined as functors commuting to certain limits (hence being continuous in a certain sense). One of the major results about those functors is that they are isomorphic to a categorical equivalent of power series (hence called analytic by J.-Y. Girard, but this terminology will not be used here because of the conflict with Joyal's analytic functors related to generating series). As such, they are a special case of *polynomial functors*, a notion that was introduced much more recently. My contribution related to normal functors semantics is two-fold, and try to clarify the structure of this model.

Polynomial functors in groupoids [24]

[24]: Finster et al. (2021), *A Cartesian Bicategory of Polynomial Functors in Homotopy Type Theory*

The first work was done in collaboration with Eric Finster, Maxime Lucas, and Samuel Mimram. Our work focusses on one issue related

to this model, namely that the corresponding category is not cartesian closed in a satisfying sense: while there is a workaround, the resulting model is not completely satisfying from category-theoretic standards. Our contribution was to show how this issue can be avoided by working with polynomial functors in *groupoids* instead of sets. This result was moreover formalised in Agda (although I did not contribute to the formalisation[1]).

1: Moreover, the formalisation was not completely finalised. It was properly finalised by Elies Harington.

Simplifying normal functors [25]

[25]: Rogers et al. (2024), *Simplifying normal functors: an old and a new model of λ-calculus*

The second work was done in collaboration with Morgan Rogers and William Troiani, and somehow takes things in the opposite direction. Instead of moving to a more complex setting (from sets to groupoids), we worked on the model's basic constructions to try and keep to a minimal setting. We end up with a much simplified model in which proofs/programs are represented as continuous functions on multisets. This simplification allows us to exhibit for the first time to interpretation of linear logic in the normal functor model, and shed light on how the interpretation circumvents the lack of cartesian closure (explained in the previous paragraph). In particular, the interpretation of the application of a lambda-term to another differs from standard interpretations. In particular, the resulting model is not captured by the standard definition of a *categorical model of linear logic*, showing that this definition motivated by having a 'nice' categorical structure ended up leaving aside perfectly correct denotational models. We are currently working on aa generalisation of this work in which we will provide an axiomatisation of a family of denotational models of lambda-calculus that do not fit the standard category-theoretic definition.

Game Semantics for Concurrent Processes [26, 27]

[26]: Eberhart et al. (2015), *An Intensionally Fully-abstract Sheaf Model for pi*
[27]: Eberhart et al. (2017), *An intensionally fully-abstract sheaf model for π (expanded version)*

An older result, with T. Hirschowitz and C. Eberhard, investigated sheaf game semantics for concurrent processes. We propose a compositional model for the pi-calculus in which processes are interpreted as sheaves on certain simple sites. Such sheaves are a concurrent form of innocent strategies, in the sense of Hyland-Ong/Nickau game semantics [28, 29]. We define an analogue of fair testing equivalence in the model and show that our interpretation is intensionally fully abstract for it. That is, the interpretation preserves and reflects fair testing equivalence; and furthermore, any innocent strategy is fair testing equivalent to the interpretation of some process. The central part of this work is the construction of sites, relying on a combinatorial presentation of pi-calculus traces in the spirit of string diagrams.

[28]: Hyland et al. (2000), *On full abstraction for PCF: I, II, and III*
[29]: Nickau (1994), *Hereditarily Sequential Functionals*

Linear realisability

Linear realisability was the topic of my PhD. I therefore naturally continued to dabble in this domain. Most of the results, in particular the series of work on Interaction Graphs [14, 16–18, 30, 31], are presented in chapter 10. More recent work with A. Ragot and B. Eng on more syntactical linear realisability constructions will be mentioned but not presented

[14]: Seiller (2019), *Interaction Graphs: Exponentials*
[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*
[17]: Seiller (2016), *Interaction graphs: Additives*
[18]: Seiller (2017), *Interaction Graphs: Graphings*
[30]: Seiller (2016), *Interaction Graphs: Full Linear Logic*
[31]: Nguyên et al. (2018), *Coherent Inter-*

[32–34]. The work with V. Maestracci [35] tries to formally establish the 2-cocycle property (presented in chapter 9) as a higher-dimensional associativity. It is however still ongoing work but I hope it will lead to a proper understanding of this essential property in terms of (co)homology and/or homotopy.

[32]: Ragot et al. (2023), *Linear Realisability Over Nets and Second Order Quantification (short paper)*
[33]: Ragot et al. (2024), *Linear realisability on untyped nets*
[34]: Eng et al. (2022), *Multiplicative linear logic from a resolution-based tile system*
[35]: Maestracci et al. (2023), *Linear Realisability and Cobordisms*

2.2. Implicit Computational complexity

Linear realisability models are particularly interesting from the point of view of computational complexity. This lead me to characterisations of complexity classes that borrows from *implicit complexity* which are explained in chapter chapter 11. Implicit computational complexity (ICC) is a subfield of computational complexity in which complexity classes are characterised without references to explicit machine models. There are (at least) three distinct approaches to ICC.

- One can study mathematically the function algebras corresponding to a complexity class. For instance Bellantoni and Cook [36] showed that the set of functions computable in polynomial time can be defined as the class consisting of basic functions (projections, constants, successors, predecessor), and closed under conditionals, safe composition and predicative recursion on notations (a variant of the recursion scheme that limits its expressivity).
- One can study logical systems which, through the proofs-as-programs correspondence, will correspond to programs computing exactly the functions in a given complexity class. For instance, Girard introduced a variant of linear logic, called *Bounded Linear Logic* [37], which corresponds to the class of functions computable in polynomial time.
- One can study rewriting systems and annotations of programs. This is stems from the so-called Copenhagen school. For instance the size-change principle [38] stems from this line of research. Another major result is the mwp-flow analysis [2] which introduces a derivation system that ensure polynomial bounds on the growth of variables in a simple (toy) imperative language.

[36]: Bellantoni et al. (1992), *A new recursion-theoretic characterization of the polytime functions*

[37]: Girard et al. (1992), *Bounded linear logic: a modular approach to polynomial-time computability*

[38]: Lee et al. (2001), *The size-change principle for program termination*
[2]: Jones et al. (2009), *A Flow Calculus of Mwp-bounds for Complexity Analysis*

Implicit complexity [39–42]

I developed, in a joint work with C. Aubert, a completely new approach to computational complexity that was proposed by Girard [43]. Based on insights and ideas from the hyperfinite geometry of interaction construction, this approach is based on the use of the crossed product* construction to characterise complexity classes as sets of operators on the hyperfinite factor of type II_1. Introducing a suited notion of abstract machines — the pointer machines, we show how they can be represented by operators induced by a group action. In a first work, we showed how this approach yields a characterisation of the non-deterministic space complexity class coNLogspace. This work quickly lead us to a characterisation of the class Logspace of deterministic logarithmic space

[39]: Aubert et al. (2016), *Characterizing co-NL by a group action*
[40]: Aubert et al. (2016), *Logarithmic Space and Permutations*
[41]: Aubert et al. (2014), *Logic Programming and Logarithmic Space*
[42]: Aubert et al. (2016), *Unary Resolution: Characterizing Ptime*
[43]: Girard (2012), *Normativity in Logic*

* Or rather a particular case of it: the wreath product construction.

predicates by showing that the restriction to deterministic machines corresponds to a restriction on the norm of operators considered.

I then worked in collaboration with C. Aubert, M. Bagnol and P. Pistone to adapt these techniques to a more syntactic setting. A work by C. Aubert and M. Bagnol [44] mimic these techniques by replacing operators with abstract term rewriting systems. Although this work in itself is a simple rephrasing of previously obtained results obtained with Aubert, it led us — C. Aubert, M. Bagnol, P. Pistone and I — to a natural, less *ad-hoc*, generalisation of the results which can be interesting for applications in logic programming [41]. The result was then extended and we obtained characterisation of the class Ptime as the set of logic programs using only unary predicates [42].

[44]: Aubert et al. (2014), *Unification and Logarithmic Space*

[41]: Aubert et al. (2014), *Logic Programming and Logarithmic Space*

[42]: Aubert et al. (2016), *Unary Resolution: Characterizing Ptime*

While this line of work is interesting, I identified two important defects. First, although it uses the same language as Girard's hyperfinite GoI model the characterisations thus obtained are not related to the GoI model. In other terms, the computational complexity results has nothing to do with the realizability models except for the employed techniques. Secondly, the move to a more syntactic framework seems to me a step back after the operator-theoretic characterisations as it throws away the mathematical structures behind the characterisations.

I therefore focussed on the objective of obtaining a deeper understanding of how complexity classes can be related to these mathematical objects in order to provide complexity theorists with new techniques and invariants. The insights and intuitions gained from these work as well as a reflexion about their limitations lead me to a more interesting approach based on a refinement of my own earlier work on realizability models for linear logic.

von Neumann algebras and complexity [11]

[11]: Seiller (2018), *A Correspondence between Maximal Abelian Sub-Algebras and Linear Logic Fragments*

Somehow in parallel of these results, I obtained one of my preferred results which I detail in chapter 11. It shows the correspondence between the classification of maximal abelian sub-algebras of von Neumann algebras and the expressivity of the fragment of linear logic one can interpret w.r.t. this sub-algebra. I have some to understand this work as a kind of prefiguration of the semantics approach to implicit complexity that followed and which improved on the work presented in the previous two subsections.

Characterising complexity classes [45, 46]

[45]: Seiller (2018), *Interaction Graphs: Nondeterministic Automata*

[46]: Seiller (2023), *Implicit complexity through linear realisability: polynomial time and probabilistic classes*

The results obtained here are discussed in chapter 11, and relate to the different results explained above. It somehow exhibits characterisations of complexity classes as in the work described in subsection 2.2 and subsection 3 while preserving the connection with (linear logic): the characterisation makes use of the linear realisability models based on graphings. This somehow continues the work relating maximal von Neumann algebras and logical expressivity, through the Murray and

von Neumann group measure space construction as explained in chapter 1. It lead me to the following result, relating complexity classes and group/monoid actions.

Monoid Action	deterministic model	non-deterministic model		probabilistic model
$\mathfrak{m}_1$	Regular	Regular	Regular	s.t.ochastic
⋮	⋮	⋮	⋮	⋮
$\mathfrak{m}_k$	D_k	N_k	co-N_k	P_k
⋮	⋮	⋮	⋮	⋮
$\mathfrak{m}_\infty$	Logspace	NLogspace	coNLogspace	PLogspace
$\mathfrak{n}_\infty$	Ptime	Ptime	Ptime	PPtime

Here the intermediate classes between regular languages and logarithmic space are defined in terms of classes of automata.

2.3. Computability, Complexity

As explained above the latter results lead me to conjecture a strong connection between the classification of monoid actions and the corresponding complexity classes. Part of my subsequent work has been dedicated to understanding if such a connection exists and if it can be exploited. As part of this investigation, I tried to understand if the techniques involved were related to the *geometric complexity theory* program of Mulmuley and Sohoni, leading me to work on Mulmuley's result establishing lower bounds for 'PRAMs without bit operations'. This then lead me further to work on non-implicit approaches to complexity theory, and more specifically algebraic complexity and Kolmogorov complexity.

Entropy and algebraic lower bounds [47]

[47]: Seiller et al. (2022), *Unifying lower bounds for algebraic machines, semantically*

In trying to understand Mulmuley's lower bound result, I realised that the proof could be reformulated using the representation of programs as graphings that underlies the implicit complexity approach. As we tried to formalise this, we realised that this reformulation could in fact be applied to several different lower bounds results in algebraic complexity, namely lower bounds results of Steele and Yao on algebraic decision trees, of Ben Or on lower bounds for algebraic computational trees, of Cucker on the separation between $\mathrm{NC}_{\mathbf{R}}$ and $\mathrm{Ptime}_{\mathbf{R}}$.

The result is presented in more details in chapter 8.

Kolmogorov complexity [48]

[48]: Léchine et al. (2024), *Kolmogorov time hierarchy and novelty games*

I was also led to work on time-bounded Kolmogorov complexity. This was sparked both from my wish to understand different lower bounds methods in computational complexity, and a specific interest in the notion because of its relation with topological entropy. The Kolmogorov complexity of a word $w \in \{0,1\}^n$ is defined as the size of the (description of) the minimal Turing machine that outputs w [49]. The notion depends on the choice of a universal Turing machine, but is shown to be sufficiently

[49]: Li et al. (1993), *An introduction to Kolmogorov complexity and its applications*

stable (for almost all strings, the choice only modifies the value of the Kolmogorov complexity by an additive constant). As part of a work with Ulysse Léchine, we investigate an old problem related to *time-bounded Kolmogorov complexity*, that is the Kolmogorov complexity of words when the time complexity of the machine producing the string is bounded. In general we fix a universal Turing machine $\mathbb{U}$ and define:

$$K[f(n), t(n), s(n)] = \{w \in \{0,1\}^n \mid \exists \phi \in \{0,1\}^{f(n)}, \mathbb{U}(\phi) \overset{\leqslant t(n), \leqslant s(n)}{\longrightarrow} w\},$$

where $\mathbb{U}(\phi) \overset{\leqslant t(n), \leqslant s(n)}{\longrightarrow} w$ means that $\mathbb{U}(\phi)$ computes w in time bounded by $t(n)$ and space bounded by $s(n)$. We can then define the corresponding complexity classes:

$$\mathbf{K}[f(\cdot), t(\cdot), s(\cdot)] = \bigcup_n K[f(n), t(n), s(n)].$$

One natural question is that of separation: can we prove that allowing more time will lead to the computation of more words? The question can be formulated as follows: can we prove that[2] $\mathbf{K}[f(\cdot), t(\cdot), s(\cdot)] \subsetneq \mathbf{K}[f(\cdot), \alpha(\cdot)t(\cdot), \beta(\cdot)s(\cdot)]$ for some functions α, β? One instance of this question was answered positively by Longpré in 1986 [50] for $\alpha(n) = 2^{f(n)}$ when no space bounds are given (we write $s(\cdot) = \infty$ in this case). The method is a standard diagonalisation argument based on the possibility of enumerating all machines $\phi \in \{0,1\}^{f(n)}$ and running them. This result was not improved since Longpré thesis[3].

2: Here the symbol $\subseteq$ means that there exists an infinite number of values of n for which the inclusion is strict.

[50]: Longpré (1986), *Resource Bounded Kolmogorov Complexity, A Link between Computational Complexity & Information Theory*

3: It is in fact given as an open problem in exercise 7.1.15 in Li and Vitany [49] since the first edition of the book.

The approach Ulysse and I follow is simple, and uses a very natural strategy: is it possible to run only part of the set of all machines and still work out a diagonalisation argument? More formally: if $\alpha(n)$ is a function smaller than $2^{f(n)}$ the number of $\phi \in \{0,1\}^{f(n)}$ one can compute is necessarily not exhaustive. But considering a family of $\frac{2^{f(n)}}{\alpha(n)}$ machines can be exhaustive. How can one adapt a diagonalisation argument in this case? We should be able to define a function S that takes as input $\alpha(n)$ values $\vec{a} = a_1, \ldots, a_{\alpha(n)}$ (corresponding to the values of ϕ a single machine can check) and produces a new value $S(\vec{a})$, with the following constraint:

$$\forall \vec{a}^1, \ldots, \vec{a}^{\frac{2^{f(n)}}{\alpha(n)}}, \exists i \in [1, \ldots, \frac{2^{f(n)}}{\alpha(n)}], \forall j \in [1, \ldots, \frac{2^{f(n)}}{\alpha(n)}], S(\vec{a}_i) \notin \vec{a}_j.$$

The question then boils down to a simple (to express) combinatorial game. Given p players, each player receives a (private) set of k values $a_1, \ldots, a_k$ in $[0, \ldots, M]$, and produces a value[4] $S_p(a_1, \ldots, a_k) \in [0, \ldots, M]$: the players win the game if at least one of the produced values is not contained in the set of all private value initially received by the players. If a winning strategy exists and can be computed in reasonable time, then we can improve Longpré's result.

4: Note that here we generalise the previous paragraph by allowing each player to have a different strategy; such a solution would work as well, but it turns out that our solution does not distinguish between players.

We spent a long time looking for solutions. For $k = 1$, a simple winning strategy is given by $S(x) = x + 1$. A solution for $p = 2$ and arbitrary values of k was found based on graph theory: one can prove by combinatorial arguments that there exists a graph G such that for every k-tuple of vertices $\vec{v} = v_1, \ldots, v_k$, there exists vertices $s_0(\vec{v})$ and $s_1(\vec{v})$ such that for all v_i, there exists an edge between v_i and $s_0(\vec{v})$ and there are no edges between v_i and $s_1(\vec{v})$. Defining the strategies of the two players

as outputting $s_0(\vec{v})$ and $s_1(\vec{v})$ respectively. But no generalisation of this method can be found for more players. In particular, the natural approach based on reasoning on coloured graphs fails because of the constraint that no polychromatic cycle appear in the graph makes the probabilistic reasoning fail (the probability that a random 3-coloured graph does not contain a polychromatic cycle goes to 0 as the size of the graph grows).

In the end, I found a general strategy, but the method uses a lot of space: the value of M needs to be very large w.r.t. the values of p and k. More precisely, the method gives $(p^k)^{p^k}$ as a lower bound for M. While a bit involved, the technique only uses basic operations on the digits used in writing the arguments in the base p^k. As a consequence, we have the following partial strengthening of Longpré: for $f(n) = o(\log(n))$, and $\alpha(n) = \frac{f(n)2^{f(n)}}{\log(n)}$, we have:

$$\mathbf{K}[f(\cdot), t(\cdot), \infty] \subsetneq \mathbf{K}[f(\cdot), \alpha(\cdot)t(\cdot), \infty].$$

We however hope that the result can be improved by finding a more efficient strategy, maybe exploiting the fact that some relaxations of the problem could be considered[5] (i.e. different strategies for the different players, but also strategies in which players may provide more than one answer). Additionally, the combinatorial problem seems general enough that it could be of use in different contexts.

5: Since first writing this, it appeared in a discussion with Peter Selinger, that it should be possible to improve the lower bound for M to $O((k^k)^p)$.

2.4. Automata and randomness

Somehow related to Kolmogorov complexity (although this part of my work does not originate from the latter), I have collaborated with Jakob G. Simonsen on Agafonov's theorem. This result concerns *normal sequences*, which are sequences that are *random* in a weak sense. I.e., a sequence $\alpha \in \{0,1\}^\omega$ is normal if and only if for all word $w \in \{0,1\}^n$, the following limit is well defined and:

$$\mathrm{freq}_w(\alpha) = \lim_{n\to\infty} \frac{\mathrm{Card}\{i \leqslant n \mid \alpha_i\alpha_{i+1}\ldots\alpha_{i+\mathrm{len}(w)-1} = w\}}{n} = 2^{-\mathrm{len}(w)}.$$

Agafonov's theorem is concerned with the *selection* of normal subsequences: given a language $L \subset \{0,1\}^*$ and a sequence $\alpha = \alpha_1\alpha_2\ldots$, one can define the selected subsequence $L(\alpha)$ as the sequence of bits $\alpha_{i_1}\alpha_{i_2}\ldots$ where $i_1 < i_2 < \ldots$ and $\{i_k \mid k \in \mathbf{N}\} = \{i \in \mathbf{N} \mid \alpha_1\alpha_2\ldots\alpha_{i-1} \in L\}$. In less formal terms, $L(\alpha)$ is the sequence of bits in α that follow a prefix that belong in L. Note that this sequence may not be infinite (for instance if the language contains only words starting with a 1). The theorem states that a sequence α is normal if and only if for all finite-state automata A the subsequence $A[\alpha]$ is either finite or normal (where $A[\alpha]$ denotes the subsequence selected by the language accepted by A).

Agafonov's proof of Agafonov's theorem [51]

Our first contribution on the topic was to find, translate, and modernise Agafonov's own proof of the theorem. Indeed, the known proofs are usually based on more involved results (of which Agafonov's theorem is

[51]: Seiller et al. (2020), *An Embellished Account of Agafonov's Proof of Agafonov's Theorem*

a corollary), and the community did not know of Agafonov's original proof but only of the translated two-page announcement of the result that appeared in the AMS Translation volumes. Some articles even claimed that Agafonov did not publish a proof of his theorem. After finding the original Agafonov paper containing the proof (in russian), we translated it into english before detailing and modernising the proof (the original paper was 4 pages long, and used outdated versions of some results).

Agafonov's theorem for infinite alphabets and arbitrary distributions [52]

[52]: Seiller et al. (2022), *Agafonov's Theorem for finite and infinite alphabets and probability distributions different from equidistribution*

The above translated proof was however instrumental in a generalisation of Agafonov's result. This generalisation answers two questions:

1. Can the result be extended to arbitrary (including infinite) alphabets?
2. Can the result be extended to arbitrary distributions (over finite words)?

We therefore considered the case of arbitrary probability distributions μ over the finite words over an alphabet Σ (this in fact corresponds to choosing a distribution on the set Σ^n for all n). Given such a distribution, we define the notion of μ-distributed sequence as a generalisation of normal sequences, namely α is μ-distributed if and only if for all word $w \in \Sigma^n$, the following limit is well defined and:

$$\mathrm{freq}_w(\alpha) = \lim_{n\to\infty} \frac{\mathrm{Card}\{i \leqslant n \mid \alpha_i\alpha_{i+1}\dots\alpha_{i+\mathrm{len}(w)-1} = w\}}{n} = \mu(w).$$

We then proved the following result that settles the question of characterising exactly the situations in which (the equivalent of) Agafonov's theorem hold for μ-distributed sequences. For this, let us call a probability distribution *Bernoulli* if it is induced by a distribution on the alphabet: there exists $\tilde{\mu}$ a probability distribution on Σ such that $\mu(a_1\dots a_n) = \prod_{i=1}^n \tilde{\mu}(a_i)$. Our result can then be stated as follows (note the alphabet is allowed to be infinite).

Theorem 2.4.1 *Agafonov's theorem holds for μ-distributed sequences if and only if μ is Bernoulli.*

Extensions

We are still working on the topic, on two distinct remaining questions.

- First: we recently showed how this result can be proved (with a slightly modified statement) for selection by (some classes of) *probabilistic automata* [53]. Agafonov's theorem (even on $\{0,1\}$ with the equidistribution) was not known to hold for the selection by probabilistic automata before that.
- Second: we tried to understand how far the set of languages preserving normality can be extended beyond regular languages. This question is a very hard open question in the small community of people working on normality. We do not have proper results for the moment, but we have made some progress, with a number

[53]: Léchine et al. (2024), *Agafonov's theorem for probabilistic selectors*

of counterexample of languages that are very close to regular languages and still do not preserve normality.

2.5. Verification

One quite different aspect of my research in the last years has been about applying methods and ideas coming from implicit computational complexity to produce implementable static analyses of programs. All results summarised here are detailed in chapter 7.

Automatic Code Optimisation in Compilers [54]

[54]: Moyen et al. (2017), *Loop Quasi-Invariant Chunk Detection*

In a work with J.-Y. Moyen and T. Rubiano, while we were all working at the University of Copenhagen, we showed how techniques from Implicit Computational Complexity can be applied in a completely different field, namely in compiler construction. More precisely, we showed how abstract graph representations defined for static analysis of complexity can be used to automatically detect invariants and quasi-invariants in `while` loops. This leads to code optimisation, as invariants and quasi-invariants can then be pulled out of the loop to avoid being unnecessarily executed at each iteration. This code optimisation, on our initial tests, proves to be more efficient that currently implemented methods in the popular compilers LLVM and GCC. We are currently implementing the method in LLVM.

This work was only the start of a potentially fruitful series of adaptation of techniques of Implicit Computational Complexity in compiler construction. Borrowing more involved techniques, we can hope to provide compiler passes that could statically analyse the complexity of the program being compiled to provide certificates of complexity bounds. For instance Jones and Kristiansen's `mwp`-polynomials [2] use a clever improvement over the dependency graphs we used to detect quasi-invariants, which we will use to provide certificate of polynomial time bounds of compiled programs.

[2]: Jones et al. (2009), *A Flow Calculus of Mwp-bounds for Complexity Analysis*

An implementable version of MWP [55, 56]

[55]: Aubert et al. (2022), *mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity*
[56]: Aubert et al. (2023), *pymwp: A Static Analyzer Determining Polynomial Growth Bounds*

The next first goal was therefore to implement the Jones and Kristiansen mwp-flow analysis. This work was the inspiration behind our first work on loop quasi-invariants. Indeed, the dependency analysis was a simplification of Jones and Kristiansen's mwp calculus. However, there was a number of issues to be dealt with. The first (minor) difficulty was the fact that the analysis could in some cases fail. The second (major) difficulty was that the original system was non-deterministic and polynomial bounds on the growth of variables in a program was ensured by the existence of a correct derivation. The last difficulty, related to the non-determinism, was the feasibility of the analysis as the computation of the mwp-bounds was conjectured by Jones and Kristiansen to be an NP-complete problem. We solved the first two issues by designing an alternative analysis which was deterministic: this was done by representing non-deterministic choices by variables. Instead of a

family of matrices with coefficients in the semiring $\{m, w, p\}$, our analysis produce a single matrix with coefficients in 'polynomials' in these choice variables (with scalars in $\{m, w, p, i\}$ – i standing for infinite to deal with non-terminating branches of the initial analysis). We implemented this analysis in a tool, in which we separated the *decision problem* (the existence of a mwp-bound) from the *computation problem* (computing specific mwp-bounds) to deal with feasibility. The resulting tool is quite fast, using both efficient algorithms (inspired by techniques akin to what is done with Gröbner bases) and the possibility of not computing specific bounds.

Automatic parallelisation [57]

[57]: Aubert et al. (2023), *Distributing and Parallelizing Non-canonical Loops*

We also published a paper showing how the dependency analysis used in the first work on peeling loops can be used to split loops. Indeed, it is possible to automatically analyse C code to detect the tree of dependencies in loops' bodies. From this, one can detect separated branches that could be parallelised. This induces a splitting of loops that is in some sense orthogonal to the usual loop parallelisations techniques. We showed on (a reduced version of) standard benchmarks that the automatically produced C code, when annotated properly with OpenMP annotations, compares positively to standard tools.

2.6. Philosophy

Lastly, I have been collaborating with philosophers since the start of my PhD thesis. This collaboration has been essential for me, as it has forced me to consider my current work and perspectives from a higher vantage point. This reflexivity has enriched my reflexion, and will certainly remain an important part of my approach to research. Generally speaking, my contributions in this direction consists in understanding how recent techniques and results in mathematics and computer science can shed new light on old problems from philosophy of science.

The computational meaning of axioms [58]

[58]: Naibo et al. (2016), *On the computational meaning of axioms*

In this paper with A. Naibo and M. Petrolo, we investigate an anti-realist theory of meaning suitable for both logical and proper axioms. As opposed to other anti-realist accounts, like Dummett-Prawitz verificationism, the standard framework of classical logic is not called into question. In particular, semantical features are not limited solely to inferential ones, but computational aspects also play an essential role in the process of determination of meaning. In order to deal with such computational aspects, a relaxation of syntax is shown to be necessary. This leads to a general kind of proof theory — abstracting the approach of realizability models for linear logic –, where the objects of study are not typed objects like deductions, but rather untyped ones, in which formulas have been replaced by geometrical configurations.

Verificationism and classical realizability [59]

[59]: Naibo et al. (2015), *Verificationism and classical realizability*

In this work, we investigated the question of whether Krivine's classical realizability can provide a verificationist interpretation of classical logic. We argue that this kind of realizability can be considered an adequate candidate for this semantic role, provided that the notion of verification involved is no longer based on proofs, but on programs. On this basis, we show that a special reading of classical realizability is compatible with a verificationist theory of meaning, insofar as pure logic is concerned. Crucially, in order to remain faithful to a fundamental verificationist tenet, we show that classical realizability can be understood from a single-agent perspective, thus avoiding the usual game-theoretic interpretation involving at least two players

Logical Constants [60]

[60]: Naibo et al. (n.d.), *Logical Constants From a Computational Point of View*

This work was started at the end of my PhD, in collaboration with A. Naibo and M. Petrolo. We study how linear realisability techniques can bring a new perspective on the old question: "what is a logical constant?" (note here that 'constant' should be understood as arbitrary *connective*, not only 0-ary). Our proposal stands in the line of proof-theoretic semantics, exploiting in an essential way the proofs-as-programs correspondence and the techniques from the field of realizability models for linear logic. We first present the inferentialist position in order to point out some of its problems. Their analysis and solutions are carried out in the light of the Curry-Howard correspondence. It is on this basis that we can formulate a necessary condition for logicality, which naturally emerges from the computational point of view adopted here.

The work has been presented in many different venues throughout the years, but we never managed to produced a finished version of the paper (even though I believe it has been cited at least once). Hopefully, I can focus on doing so after finishing up the current document.

Understanding Weyl [61]

[61]: Joinet et al. (2021), *From abstraction and indiscernibility to classification and types: revisiting Hermann Weyl's theory of ideal elements*

The linear realisability techniques involve a definition of the notion of type based on some *orthogonality relation* corresponding to negation. This construction can be abstracted and simplified to define a notion of type based on sets X, Y equipped with a binary relation $\mathcal{R} \subset X \times Y$. This simplified construction is related to work by Birkhoff [62] and has already been introduced in computer science under the name *formal concept analysis* [63]. In a collaboration with Jean-Baptiste Joinet, we showed how these techniques can be understood as a *theory of classification*. We also discuss how the techniques allow to define new concepts. The generally accepted and well studied theory for doing so in philosophy of logic is the so-called 'definition by abstraction', which is based on equivalence relations. We explain in this work the constructions in linear realisability and formal concept analysis are more general than definition by abstraction and formalise in a way Weyl's *theory of ideal elements* [64, 65], an alternative proposal for defining concepts.

[62]: Birkhoff (1940), *Lattice theory*

[63]: Wille (1982), *Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts*

[64]: Weyl (1910), *Über die Definitionen der mathematischen Grundbegriffe*
[65]: Weyl (1949), *Philosophy of Mathematics and Natural Sciences*

Epistemology of computer science [66, 67]

[66]: Naibo et al. (n.d.), *Algorithms*
[67]: Naibo et al. (n.d.), *An ontology of computer science*

In the last three or four years, I have been collaborating with Alberto Naibo on the philosophy of computer science. The field is still underdeveloped, and work focus on anecdotic questions (such as so-called 'deviant encodings'). We have been building up a small team consisting of both computer scientists and philosophers over the years, which meets regularly (once every two weeks) to discuss foundational notions in computer science. This was initiated by a technical development in my work: I realised that the notion of monoid action underlying my work on linear realisability could be understood as a formal definition of the notion of *model of computation*. This leads to an abstract theory of computability providing formal definitions of the basic notions of computer science: model of computation, machine, program, algorithm, etc. This specific aspect is detailed in chapter 6.

Part I.

A mathematical theory of computer science

Abstract models of computation 3.

3.1. Definition

Throughout the years, I have proposed several small variants of the notion of "abstract model of computation". The principle is essentially the same: a model of computation is a monoid action on a space $M \curvearrowright \mathbf{X}$. The space is thought of as a space of configurations, i.e. the potential states the machine can find itself in. Each available instruction in the machine then defines a endomorphism of $\mathbf{X}$. The monoid structure naturally arises from the ability of performing sequences of instructions. This notion however requires a number of subtle modifications.

The first is that while all endomorphisms described by the action are potentially performed by the machine, computability puts special interest in "basic instructions" or *atomic instructions* – that correspond to a single instruction in the machine model. This is to be opposed to composed ones. As a simple example, one wants to distinguish between moving the head of a Turing machine to the next bit on the right, and moving the tape head to the second bit to the right – an operation which requires two steps of computation. As a consequence, we will work with *presentations of monoids* in order to distinguish the set of atomic instructions which generates the action.

The second is the fact that models of computation also include the possibility to read/check some properties of the configuration. I.e. a Turing machine allows to verify the value on the tape at the head's position, and perform different instructions based on this value. Mathematically, this means programs are allowed to use instructions only on subspaces. In first iterations of this approach, this was included in the notion of *graphing*. This is somehow explained by the fact that graphings arose from a different context and were only a posteriori understood as abstract machines. But this definition is too permissive: allowing for just any subspace to be considered would allow to encode uncomputable sets. I.e. in the basic representation of Turing machine presented below, it would be possible to define a machine that accepts exactly on the subspace of configurations containing a binary integer n such that the n-th Turing machine terminates on n, and rejects otherwise. To avoid this, previous versions of this very text considered the addition of a set of "atomic conditions" – some partitions of the space allowed to be used in the machines. This addition was neither practical nor aesthetically pleasing[1]. It only later appeared to me that these conditions could be incorporated within the monoid action itself, since a subspace is defined by the projection onto it.

1: The approach will nonetheless be detailed in a later section to bridge the new definition with the definition used in my published papers.

These considerations lead me to the following definition.

Definition 3.1.1 (AMC) *An* abstract model of computation *(AMC) is a monoid action $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ where:*

1. *$\mathbf{X}$ is a space of configurations, together with a notion of morphisms;*
2. *I is a set of instructions, generating the monoid action α;*

We first note that the monoid $\mathbb{M}(I)$ should be understood as the monoid defined by the set of generators I and the relations satisfied by the maps $\alpha(i)$ for $i \in I$. This can also be understood as follows: one picks a generating set I and maps $\alpha(i)$ for $i \in I$, then considers the submonoid of $\mathrm{Hom}(\mathbf{X}, \mathbf{X})$ generated by αI.

Remark 3.1.1 This definition of AMC can be formalised using category theory as follows: an abstract model of computation is a functor from a category into a category of *spaces*. While this categorical definition may seem more general, the two formalisms are in fact equivalent: considering the coproduct of the spaces in the image of the functor, one recovers a monoid of endomorphisms over a single space.

Before going through (many) examples, I want to address the question of restrictions. Some will view this notion of model of computation too general. And indeed, it is very general and captures some models that some may not consider as models of *computation*.

One standard restriction, which is considered for instance by Gandy [68] or Gurevich [69], is that of 'bounded exploration', or some notion of *locality*. This restriction, while never explicitly motivated, may arise from questions of physical realisability: if a physical device were to be built in order to compute according to the principles defined by the model of computation, then laws of physics would impose some constraints. In particular, a step of computation may not affect a value that is arbitrarily 'far away'. These conditions could be imposed in the proposed formalism by imposing a topology on the underlying space and considering only continuous actions. I must say, however, that these considerations do not really mean a lot once the model of computation is separated from the *cost model* (accounting for complexity), which is the case here (complexity is discussed in chapter 5). Indeed, one could very well allow an instruction to modify an arbitrary far-away value while giving it a cost proportional to the distance. In that case, the model remains 'realistic' while not satisfying the locality constraints.

[68]: Gandy (1980), *Church's Thesis and Principles for Mechanisms*

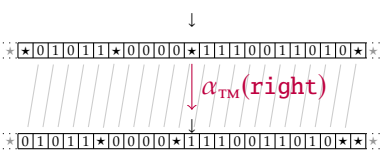
[69]: Gurevich (2000), *Sequential Abstract State Machines capture Sequential Algorithms*

The point of view taken here is that the proposed definition is in some way 'maximal', and offers a mathematical framework in which restrictions may be discussed and defined formally. I do not oppose to define, question, and study subtle conditions that models of computations should satisfy. I believe the current approach will offer a way to do so in a proper and precise way.

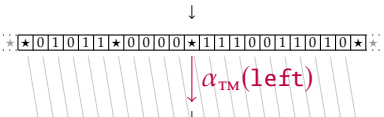
3.2. Examples

Machine models

We start with a detailed example. The machine model is that of single-tape Turing machines over the alphabet $\{0, 1\}$. While the following AMC captures this model, it is not unique. Other AMCs corresponding to the same model will be discussed later.



(a) Instruction `right`.



Example 3.2.1 The (single tape) Turing machines model can be represented by the AMC

$$\alpha_{\text{TM}} : \mathbb{M}(\text{Instr}_{\text{TM}}) \curvearrowright \mathbf{X}_{\text{TM}}$$

defined as follows.

- The space of configuration $\mathbf{X}_{\text{TM}}$ is defined as the space of $\mathbf{Z}$-indexed sequences in $\{0, 1, \star\}$ which are almost-always equal to $\star$:

  $$\mathbf{X}_{\text{TM}} = \{(s_i)_{i\in\mathbf{Z}} \mid s_i \in \{0, 1, \star\}, \text{Card}\{i \in \mathbf{Z} \mid s_i \neq \star\} < \infty\}.$$

  Here $\star$ represents the empty symbol, and it is implied that the head of the Turing machine points to the 0-th indexed element. The notion of morphism is here that of partial maps, allowing to define *read* instructions.
- The set Instr_{TM} of atomic instructions of the Turing machine model is defined as

  $$\{\texttt{right}, \texttt{left}\} \cup \{\texttt{write}_i \mid i \in \{0, 1, \star\}\} \cup \{\texttt{read}_i \mid i \in \{0, 1, \star\}\},$$

  and the action α_{TM} is induced by the following actions realising these instructions:
  - move the head to the right; this is represented as the shift:

    $$\alpha_{\text{TM}}(\texttt{right}) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_{i-1})_{i\in\mathbf{Z}};$$

  - move the head to the left; this is represented as the inverse of the shift:

    $$\alpha_{\text{TM}}(\texttt{left}) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_{i+1})_{i\in\mathbf{Z}};$$

  - write a symbol at the location pointed by the head; this is represented as three different maps:

    $$\alpha_{\text{TM}}(\texttt{write}_\star) : (s_i)_{i\in\mathbf{Z}} \mapsto (t_i)_{i\in\mathbf{Z}} \text{ s.t. } \begin{cases} t_0 = * \\ t_i = s_i \quad \text{for } i \neq 0 \end{cases};$$

  - check the value read by the head; this is represented by adding the following three (partial) maps:

    $$\alpha_{\text{TM}}(\texttt{read}_\star) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_i)_{i\in\mathbf{Z}} \text{ if and only if } s_0 = *;$$

Example 3.2.2 (One-way finite automata) We can define a much simpler model of computation: one-way finite automata. For this, consider the action $\alpha_{\text{FA}} : \mathbb{M}(\text{Instr}_{\text{FA}}) \curvearrowright \mathbf{X}_{\text{FA}}$ defined by the space of pairs (s, h) consisting in a finite sequence s of 0 and 1 (of length k) and a position $h \in \{1, \ldots, k\}$:

$$\mathbf{X}_{\text{FA}} = \sum_{k\in\mathbf{N}} \{0, 1\}^k \times \{0, k-1\},$$

together with the following action:

- $\alpha_{\text{FA}}(\texttt{right}) : (s, h) \mapsto (s, h+1)$ if $h \neq \text{len}(s)$ and undefined otherwise;
- $\alpha_{\text{FA}}(\texttt{read}_*) : (s, h) \mapsto (s, h)$ if $s_h = *$ and undefined otherwise.

Example 3.2.3 (Two-way finite automata) We can also define two-way finite automata. For this, consider the action $\alpha_{\text{2FA}} : \mathbb{M}(\text{Instr}_{\text{2FA}}) \curvearrowright \mathbf{X}_{\text{FA}}$

defined by the same space $\mathbf{X}_{\text{FA}}$ as for the one-way automata, and the following action:

- $\alpha_{\text{FA}}(\texttt{right}) : (s, h) \mapsto (s, h+1)$ if $h \neq \text{len}(s)$ and (s, h) otherwise;
- $\alpha_{\text{FA}}(\texttt{left}) : (s, h) \mapsto (s, h-1)$ if $h \neq 1$ and (s, h) otherwise;
- $\alpha_{\text{FA}}(\texttt{read}_*) : (s, h) \mapsto (s, h)$ if $s_h = *$ and undefined otherwise.

Example 3.2.4 (Pushdown automata) As an example of stack machine, we consider two-way automata with a pushdown stack. For this, consider the action $\alpha_{\text{2FA+S}} : \mathbb{M}(\text{Instr}_{\text{2FA+S}}) \curvearrowright \mathbf{X}_{\text{2FA+S}}$ defined by the space

$$\mathbf{X}_{\text{2FA+S}} = \mathbf{X}_{\text{FA}} \times \left(\sum_{k \in \mathbf{N}} \{0,1\}^k \right),$$

and the following action:

- $\alpha_{\text{2FA+S}}(\texttt{right}) : (s, h, p) \mapsto (s, h+1, p)$ if $h \neq \text{len}(s)$ and (s, h, p) otherwise;
- $\alpha_{\text{2FA+S}}(\texttt{left}) : (s, h, p) \mapsto (s, h-1, p)$ if $h \neq 1$ and (s, h, p) otherwise;
- $\alpha_{\text{2FA+S}}(\texttt{read}_\star) : (s, h, p) \mapsto (s, h, p)$ if and only if $p = * \cdot \bar{p}$ $(* \in \{0,1\})$;
- $\alpha_{\text{2FA+S}}(\texttt{pop}) : (s, h, p) \mapsto (s, h, \bar{p})$ if $p = a \cdot \bar{p}$;
- $\alpha_{\text{2FA+S}}(\texttt{push}_*) : (s, h, p) \mapsto (s, h, * \cdot p)$ $(* \in \{0,1\})$;

Example 3.2.5 (Multiple stack automata) We now extend the previous example with additional stacks. I.e. we consider the AMC $\alpha_{\text{stack}[k]} : \mathbb{M}(\text{Instr}_{\text{stack}[k]}) \curvearrowright \mathbf{X}_{\text{stack}[k]}$ defined by:

$$\mathbf{X}_{\text{stack}[k]} = \mathbf{X}_{\text{FA}} \times \left(\sum_{k \in \mathbf{N}} \{0,1\}^k \right)^k,$$

and the following action:

- $\alpha_{\text{stack}[k]}(\texttt{right}) : (s, h, p_1, \ldots, p_k) \mapsto (s, h+1, p_1, \ldots, p_k)$ if $h \neq \text{len}(s)$ and $(s, h, p_1, \ldots, p_k)$ otherwise;
- $\alpha_{\text{stack}[k]}(\texttt{left}) : (s, h, p_1, \ldots, p_k) \mapsto (s, h-1, p_1, \ldots, p_k)$ if $h \neq 1$ and (s, h, p) otherwise;
- $\alpha_{\text{stack}[k]}(\texttt{read}_*^\ell) : (s, h, p_1, \ldots, p_k) \mapsto (s, h, p_1, \ldots, p_k)$ if and only if $p_\ell = * \cdot \bar{p}$ $(* \in \{0,1\})$;
- $\alpha_{\text{stack}[k]}(\texttt{pop}^\ell) : (s, h, p_1, \ldots, p_k) \mapsto (s, h, \bar{p}_1, \ldots, \bar{p}_k)$ where $\bar{p}_i = \tilde{p}_i$ if $i = \ell$ and $p_i = a \cdot \tilde{p}_i$, and $\bar{p}_i = p_i$ otherwise;
- $\alpha_{\text{stack}[k]}(\texttt{push}_*^\ell) : (s, h, p_1, \ldots, p_k) \mapsto (s, h, \bar{p}_1, \ldots, \bar{p}_k)$ where $\bar{p}_i = * \cdot p_i$ if $i = \ell$ and $\bar{p}_i = p_i$ otherwise $(* \in \{0,1\})$;

Circuits and parallel random access machines (PRAMS) can be represented as abstract models of computation. We do not describe how here, but chapter 8 details on the representation of algebraic circuits and algebraic PRAMS which can easily be adapted for the boolean versions.

Algebraic models

As a first example of a model of computation of a more algebraic nature, we will represent BSS models. These abstract models of computation

introduced by Blum, Smale and Shub are very close to Turing machines and parametrised by the choice of an algebraic structure (in practice an ordered field or ring, but in theory weaker structures could be considered) whose elements can be stored and accessed in the same way bits are accessed in Turing machines, and basic operations of the algebraic structure are available as instructions.

It is an interesting example on a conceptual level. This exemplifies that an abstract model of computation is just that: abstract. Whether this model is *reasonable* or *realistic* is not considered here. In fact, it is the point of view of the approach that any model could be both realistic and unrealistic because the question implicitly assumes the existence of a *cost model* (section 5.4 and section 5.5). Cost models will be introduced later and will be used to define computational complexity in an abstract manner. But a given model, say the BSS model on the rational numbers, could be endowed with both an unrealistic cost model or a realistic one.

Example 3.2.6 (BSS machine) We here base our definition on the original Blum, Smale and Shub paper [70]. We fix an ordered ring or field $\mathbb{A} = (A, +, \times, \dots)$. We define the *BSS model over* $\mathbb{A}$ as the AMC $\alpha_{\text{BSS}[\mathbb{A}]} : \mathbb{M}(\text{Instr}_{\text{BSS}[\mathbb{A}]}) \curvearrowright \mathbf{X}_{\text{BSS}[\mathbb{A}]}$, where:

[70]: Blum et al. (1988), *On a Theory of Computation over the Real Numbers; NP Completeness, Recursive Functions and Universal Machines (Extended Abstract)*

$$\mathbf{X}_{\text{BSS}[\mathbb{A}]} = A_0^{\mathbf{Z}} = \{f : \mathbf{Z} \to A \mid \text{Card}(\{i \in \mathbf{Z} \mid f(i) \neq 0\}) < \infty\}$$

and the monoid action is induced by:

- $\alpha_{\text{BSS}[\mathbb{A}]}(\text{op}_\star^{\vec{i};j}) : f \mapsto \bar{f} : \mathbf{Z} \to A, \bar{f}(j) = \star(f(i_1), \dots, f(i_k))$ and $\bar{f}(i) = f(i)$ when $i \neq j$, where $\star$ is a polynomial map (or rational map in case $\mathbb{A}$ is a field) of arity k, $\vec{i} \in \mathbf{Z}^k$, and $j \in \mathbf{Z}$;
- $\alpha_{\text{BSS}[\mathbb{A}]}(\texttt{test}^i_{\leq 0}) : f \mapsto f$ if and only if $f_i \leq 0$;
- $\alpha_{\text{BSS}[\mathbb{A}]}(\texttt{test}^i_{\geq 0}) : f \mapsto f$ if and only if $f_i \geq 0$.

We will now give a more general definition of *algebraic machines* which will be equivalent to the definition above when it comes to computability. The main difference is that while more complex operations are allowed in one step here, we will only allow atomic operations to be performed in our modified version. It should be clear to the reader that any polynomial or rational map can be computed in several steps using the basic operations of the ring (or field), showing intuitively that the models are equivalent.

Example 3.2.7 (algebraic machines) Consider now an algebraic structure

$$\mathbb{A} = (A, \text{op}_1, \dots, \text{op}_m, \text{rel}_1, \dots, \text{rel}_n, \text{c}_1, \dots, \text{c}_l),$$

where op_i are operations – functions of a fixed arity a_i –, and rel_i are relations – subsets of A^{k_i} for a fixed integer k_i, and c_i are constants. We consider here that m, n, and p can be infinite with m and n at most equal to $2^{\text{Card}(A)}$ and p at most $\text{Card}(A)$; this allows for abstract models of computation with instructions for all possible operations and relations, and in which any constant may be introduced. We define the *model of algebraic machines over* $\mathbb{A}$ as the AMC $\alpha_{\text{ALG}[\mathbb{A}]} : \mathbb{M}(\text{Instr}_{\text{ALG}[\mathbb{A}]}) \curvearrowright \mathbf{X}_{\text{ALG}[\mathbb{A}]}$, where:

$$\mathbf{X}_{\text{ALG}[\mathbb{A}]} = A_0^{\mathbf{Z}} = \{f : \mathbf{Z} \to A \mid \text{Card}(\{i \in \mathbf{Z} \mid f(i) \neq 0\}) < \infty\}$$

and the monoid action is induced by:

- $\alpha_{\text{ALG}[\mathbb{A}]}(\texttt{right}) : f \mapsto \bar{f} : \mathbf{Z} \to A, \bar{f}(i) = f(i+1)$;
- $\alpha_{\text{ALG}[\mathbb{A}]}(\texttt{left}) : f \mapsto \bar{f} : \mathbf{Z} \to A, \bar{f}(i) = f(i-1)$;
- for all $i = 1, \dots, l$, $\alpha_{\text{BSS}[\mathbb{A}]}(\texttt{const}_{\texttt{c}_\texttt{i}}) : f \mapsto \bar{f}$ where $\bar{f}(0) = c_i$ and $\bar{f}(p) = f(p)$ for $p \neq 0$;
- for all $i = 1, \dots, m$, $\alpha_{\text{ALG}[\mathbb{A}]}(\texttt{op}_\texttt{i}) : f \mapsto \bar{f} : \mathbf{Z} \to A, \bar{f}(0) = \text{op}_i(f(0), \dots, f(a_i - 1))$ and $\bar{f}(i) = f(i)$ when $i \neq 0$;
- for all $i = 1, \dots, n$, $\alpha_{\text{ALG}[\mathbb{A}]}(\texttt{rel}_\texttt{i1}) : f \mapsto f$ defined if and only if $\text{rel}_i(f(0), \dots, f(k_i - 1))$;
- $\alpha_{\text{ALG}[\mathbb{A}]}(\texttt{copy}) : f \mapsto \bar{f}$ where $\bar{f}(0) = f(1)$ and $\bar{f}(p) = f(p)$ for $p \neq 0$.

This can be understood an 'intuitionistic' version of the algebraic machine. A 'classical' version is defined by adding the following instruction that allows to compute the complement of relations:

- for all $i = 1, \dots, n$, $\alpha_{\text{ALG}[\mathbb{A}]}(\texttt{rel}_\texttt{i0}) : f \mapsto f$ defined if and only if $\neg\text{rel}_i(f(0), \dots, f(k_i - 1))$.

Another algebraic model is that of Iterated Matrix Multiplication. Note that this model is parametrised by a size which represents the number of available registers. While all polynomials are computable in this model as soon as there are more than 3 available registers, a result of Allender and Wang [71] shows that there exists polynomials that cannot be computed in the model of 2×2 matrices (with the understanding that a polynomial is computed by a sequence of matrices if it appears as the top-left coefficient of the product). It is not complicated to see this model as a special case of $\alpha_{\text{ALG}[A]}$ for a well-chosen algebraic structure A.

[71]: Allender et al. (2016), *On the power of algebraic branching programs of width two*

Example 3.2.8 (Iterated matrix multiplication models) We consider a fixed size $\ell \in \mathbf{N}$ and a fixed ring $\mathbf{k}$. Then the Iterated matrix multiplication model is defined as the AMC $\alpha_{\text{IMM}} : \mathbb{M}(\text{Instr}_{\text{IMM}}) \curvearrowright \mathbf{X}_{\text{IMM}}$, where:

- the space $\mathbf{X}_{\text{IMM}}$ is the algebra $\mathcal{M}_{\ell,\ell}(\mathbf{k}[X_1, \dots, X_k, \dots)$ of $\ell \times \ell$ matrices over the polynomial ring $\mathbf{k}[X_1, \dots, X_k, \dots]$;
- the set of instructions is $\text{Instr}_{\text{IMM}} = \mathcal{M}_{\ell,\ell}(\mathbf{k}_1[X_1, \dots, X_k, \dots])$, i.e. the set of $\ell \times \ell$ matrices whose coefficients are polynomials in $\mathbf{k}[X_1, \dots, X_k, \dots]$ of degree at most 1;
- the instructions in $\text{Instr}_{\text{IMM}}$ are acting on $\mathbf{X}_{\text{IMM}}$ by left multiplication.

Other examples of instances of $\alpha_{\text{ALG}[A]}$ include quantum models of computation. Indeed if $A = \oplus_{i=0}^{\omega} \mathbf{C}$, one can consider a model in which any unitary operator acting on a finite dimensional subspace can be considered as an instruction. Other models based on different spaces, such as $\ell^2(\mathbf{N})$ or equivalently $L^2(\mathbf{R}, \lambda)$ (λ being the Lebesgue measure), can also be considered to represent models allowing to consider infinite-dimensional spaces or continuous-variable models [72, 73]. Or one can restrict the model to a finite set of instructions, such as the Clifford group extended with the T gate, to obtain a more realistic model. In cases like this one, a notion of *approximate simulation* should be considered (although it will not be detailed in this document).

[72]: Braunstein et al. (2005), *Quantum information with continuous variables*
[73]: Weedbrook et al. (2012), *Gaussian quantum information*

Abstract State Machines

The above example of *algebraic machines* seems close to Gurevich's notion of *Abstract State Machines* (ASM) [69], initially introduced as *evolving*

[69]: Gurevich (2000), *Sequential Abstract State Machines capture Sequential Algorithms*

algebras [74]. However, while both use first order structures, abstract state machines rely on those to define states (i.e. the underlying space of the AMC), while we use it here to describe the set of instructions (i.e. the monoid and its action). One consequence of this is that first order structure in abstract state machines evolves, while in the AMC just described it remains the same throughout the computation. This does not prevent us from defining an AMC that captures Gurevich's model of computation.

[74]: Gurevich (1995), *Specification and Validation Methods*

Example 3.2.9 We define the AMC $\alpha_{\text{ASM}[A,\mathcal{S}]} : \mathbb{M}(\text{Instr}_{\text{ASM}[A,\mathcal{S}]}) \curvearrowright \mathbf{X}_{\text{ASM}[A,\mathcal{S}]}$ of abstract state machines over a domain A and a first order signature $\mathcal{S}$ as follows[2]. For simplicity we will identify the set Const of constants with 0-ary functions. We write $\text{Fun}(\mathcal{S})$ the set of function symbols, denoting by $\text{ar}(f)$ the arity of an element $f \in \text{Fun}(\mathcal{S})$. We write $\text{Rel}(\mathcal{S})$ the set of relation symbols, and denote by $\text{ar}(R)$ the arity of a relation $R \in \text{Rel}(\mathcal{S})$. The space is defined as:

2: We here only define *sequential* abstract state machines, but extending it with parallelism is straightforward (one simply allows multiple updates to occur simultaneously).

$$\mathbf{X}_{\text{ASM}[A,\mathcal{S}]} = \left(\prod_{f \in \text{Fun}(\mathcal{S})} A^{\text{ar}(f)} \to A \right) \times \left(\prod_{R \in \text{Rel}(\mathcal{S})} A^{\text{ar}(R)} \to A \right).$$

Elements of this space will be written as pairs $((\bar{f})_{f \in \text{Fun}(\mathcal{S})}, ((\bar{R})_{R \in \text{Rel}(\mathcal{S})}))$.

The set of instructions is then defined from the set of *closed terms* CT, i.e. terms defined by the following grammar:

$$c = c \in \text{Const} \mid f(c, \dots, c) \text{ for } f \in \text{Fun}(\mathcal{S}) \text{ with } \text{ar}(f) = n.$$

Given an element $((\bar{f}), (\bar{R}))$ in $\mathbf{X}_{\text{ASM}[A,\mathcal{S}]}$ and a closed term $c \in \text{CT}$, we define $\bar{c}$ as the element of A defined inductively as follows:

- if $c \in \text{Const}$, then $\bar{c}$ is defined as in the pair $((\bar{f}), (\bar{R}))$ (remember that c is a unary function symbol, hence interpreted as a map $* \to A$, i.e. as an element of A);
- if $c = g(c_1, \dots, c_n)$ for closed terms $c_1, \dots, c_n \in \text{CT}$ and a function symbol $g \in \text{Fun}(\mathcal{S})$ of arity n, we define $\bar{c}$ as the value of $\bar{g}$ when evaluated as $\bar{c}_1, \dots, \bar{c}_n$, i.e. $\bar{c} = \bar{g}(\bar{c}_1, \dots, \bar{c}_n)$.

The instructions and their actions are then defined as follows:

- for all $t = g(c_1, \dots, c_n) \in \text{CT}$ and $t_0 \in \text{CT}$, there exists an instruction $\texttt{update}(t \leftarrow t_0)$. It is realised by the endomorphism $\alpha_{\text{ASM}[A,\mathcal{S}]}(\texttt{update}(t \leftarrow t_0)$ which maps $((\bar{f}), (\bar{R}))$ to $((\bar{f}'), (\bar{R}'))$ where:
  - $\bar{g}'(\bar{c}_1, \dots, \bar{c}_n) = \bar{t}_0$;
  - for all $R \in \text{Rel}(\mathcal{S})$ we have $\bar{R} = \bar{R}'$;
  - for all $f \in \text{Fun}(\mathcal{S})$ different from g, we have $\bar{f} = \bar{f}'$;
  - for all $(a_1, \dots, a_n) \in A^n$ different from $(\bar{c}_1, \dots, \bar{c}_n)$, we have $\bar{g}'(a_1, \dots, a_n) = \bar{g}(a_1, \dots, a_n)$.
- for all $R \in \text{Rel}(\mathcal{S})$, $c_1, \dots, c_n \in \text{CT}$ and $t_0 \in \text{CT}$, there is an instruction $\texttt{test}(R(c_1, \dots, c_n), t_0)$. It is realised by the partial map $\alpha_{\text{ASM}[A,\mathcal{S}]}(\texttt{test}(R(c_1, \dots, c_n))$ which maps $((\bar{f}), (\bar{R}))$ to itself *if and only if* $\bar{R}(\bar{c}_1, \dots, \bar{c}_n) = \bar{t}_0$.

One can here understand the differences expressed above. In particular, the abstract state machine model does not identify the set of possible

instructions as properly as in AMCs. This implies that that the structure of instructions, and the possible dynamics induced by abstract state machines, do not naturally arise as mathematical object. In opposition to this, we refer the reader to section 4.3 in which we show how possible computations performed by machines described by an AMC can be described mathematically. This aspect stems from the fact that *AMC focus on transitions/instructions rather than states* and is fundamental in the approach, especially when applying the techniques in verification (chapter 7) and complexity (chapter 7, see also chapter 12).

Rewriting systems

Rewriting systems define in a natural way a monoid action. However, this action does not capture exactly the model of computation defined from rewriting systems. Let us illustrate this on an example. Lambda-calculus considered with a specific reduction strategy defines a natural (partial) monoid action on the space of all λ-terms Λ: a given term t is mapped to t' w.r.t. this action if and only if $t \to_\beta t'$. However, this describes the underlying dynamics of λ-calculus. Lambda-calculus becomes a model of computation once the notion of *application* is defined: this lifts the action defined by beta-reduction to an action of Λ onto itself defined by $t \curvearrowright u = v$ if and only if $(t)u \to_\beta^* v$ with v in normal form. This can easily lead to confusion, since it implies that terms can be considered both as *data* and as *programs*.

Example 3.2.10 (Prefix rewriting systems) Let Σ be an alphabet and R a set of *rules*, i.e. $R \subset \Sigma^* \times \Sigma^*$. The associated term rewriting system $\Sigma^* \to \Sigma^*$ is defined as an AMC as follows. The Space $\mathbf{X}_{\text{PREFIX}}$ is defined as Σ^*, the set of instructions $\text{Instr}_{\text{PREFIX}}$ is defined as R, and for all $r = (w, w') \in R$, the action $\alpha_{\text{PREFIX}}(r)$ is defined by $\alpha_{\text{PREFIX}}(r)(v) = w' \cdot u$ when $v = w \cdot u$ and undefined otherwise.

This example naturally leads to the consideration of more general notions of rewriting. This leads to a fundamental question though: can a model of computation be non-deterministic? Obviously, non-determinism is considered in the literature, but non-determinism is involved at the level of machines, not the model of computation itself. We note moreover that those can be represented through *advice*. More precisely, it is known that the definition of NPTIME as languages decided by a non-deterministic Turing machine with a running time bounded by a polynomial is unsatisfactory. Indeed, the preferred[3] definition is as languages decided by (non-deterministic) Turing machines running in polynomial time with a polynomial advice, i.e.[4]

$$\mathcal{L} \in \text{NPTIME} \iff \exists M, \exists P, \forall x \in \mathcal{L}, \exists y \in \{0,1\}^*, |y| = P(|x|) \wedge M(x,y) \downarrow_{P(|x|)} .$$

In a similar way, a probabilistic machine does not choose which branch to take: this is determined by the environment. In a sense this non-determinism differs from the non-deterministic aspects of term-rewriting systems such as lambda-calculus in which it is not the machines that are non-deterministic (i.e. in some way impacted by their environment) but the instructions themselves.

3: While both definitions are equivalent for polynomially bounded time, the advice definition is the only one that scales down to smaller classes, in particular because it explicitly bounds the size of the advice.

4: We write here $M(x,y) \downarrow_{P(|x|)}$ to express that the machine M given (x,y) as inputs will terminate in at most $P(|x|)$ steps.

The situation is somehow similar in the case of rewriting. More formally, consider the rewriting rule $010 \longrightarrow 100$ on words over the alphabet $\{0,1\}$. Now, the word 0010101 can be rewritten in two ways: $0010101 \longrightarrow 0100101$ and $0010101 \longrightarrow 0011001$. In fact this rewriting system is not confluent and this choice impact the result definitively. The analysis of this example leads to the recognition that we are in the presence of some non-deterministic behaviour at the level of the instructions. Can the same point of view as above be adopted here? This boils down to the question: how can one associate an advice here? While advices can be used to chose between a left and a right branch in a program without more information, the same process cannot be defined here without knowledge of the whole term (or equivalently, of all the possible redexes for applying the chosen instruction).

To be more precise, we refuse the idea that an instruction can lead to underspecified behaviour. If an instruction corresponds intuitively to an idealisation of a physical process, then it cannot be purely non-deterministic: there will always be some information from the context that will make the operation deterministic, or some probabilistic law governing the behaviour of the device. As such, a coin toss instruction is not non-deterministic because it is given a specified behaviour based on a precise probability distribution. Another example is that of race conditions in parallel models of computation: we consider here that the runtime information that leads to either processor A or processor B to be the first to write a symbol in a cell should be represented as an advice string accounting for the environment. The complete behaviour of the system is then represented by an additional choice of a probability distribution on strings.

This is somehow equivalent to introducing a notion of probabilistic strategy.

Definition 3.2.1 *Let (A,R) be a rewriting system. For each $a \in A$ and $r \in R$, we write* $\mathrm{redex}(a,r)$ *the set of* redexes *in a for the rule r. A* strategy *for this rewriting system is a map $S : A \times R \longrightarrow \mathrm{Prob}(\mathrm{redex}(a,r))$, i.e. a map that associate to each term $a \in A$ and rule $r \in R$ a probability distribution $S_{(a,r)}$ over the set* $\mathrm{redex}(a,r)$.

Given S, a, r, and an element $u \in [0,1]$, we write $S(a,r)(u)$ the result of applying to a the rule r on the redex[5] $S^{-1}_{(a,r)}(u)$.

Since the set of redexes is finite, the probability distributions are discrete, and the choice can be imposed by the environment by the production of a real number in the unit interval $[0,1]$. We note however that this notion of strategy contains non-computable examples. This is true for two distinct reasons. First because it uses real numbers without restrictions on the kind of probability distribution allowed. Second because it is simply "a map" and not a computable one.

5: Since $\mathrm{redex}(a,r)$ is finite, we can assume without loss of generality that S maps elements of $\mathrm{redex}(a,r)$ to (disjoint) intervals in $[0,1]$. In that case, sampling over $\mathrm{redex}(a,r)$ is the same as sampling over $[0,1]$. For any $u \in [0,1]$, we then write $S^{-1}_{(a,r)}(u)$ the element of $\mathrm{redex}(a,r)$ corresponding to u.

Definition 3.2.2 *Let (A,R) be a rewriting system and S a strategy. We define the space*

$$\mathbf{X}_{R,S} = \Sigma^* \times [0,1]^\omega.$$

The monoid action is then generated by the transformations $\alpha(r)$ for $r \in R$, acting as follows:

$$\alpha(r) : (t,\rho) \mapsto (S^{-1}_{(t,r)}(u), \rho'),$$

where $\rho = u \cdot \rho'$ *and when* $\mathrm{redex}(t, r)$ *is non-empty (the action is undefined when it is empty).*

Remark 3.2.1 We note that a different definition could be given in the case of deterministic strategies. Consider an alphabet Σ, a set of rewriting rules R, and a *deterministic* reduction strategy S for (Σ^*, R). One can define the following abstract model of computation. The space $\mathbf{X}^{\mathrm{det}}_{R,S}$ is defined as Σ^*, the set of instructions is defined as $\mathrm{Instr}^{\mathrm{det}}_{R,S} = R$, and the action $\alpha^{\mathrm{det}}_{R,S}(r)$ for $r \in R$ is defined as $\alpha^{\mathrm{det}}_{R,S}(r)(x) = \mathrm{red}(x, S(x, r))$ (where $S(x, r)$ denotes the unique element of $\mathrm{redex}(x, r)$ designated by the strategy) if $\mathrm{redex}(x, r) \neq \emptyset$ and undefined otherwise.

This model of computation is not a specific case of the above because $\mathbf{X}^{\mathrm{det}}_{R,S} \neq \mathbf{X}_{R,S}$. But it is in fact a *retract* of the one above (Definition 3.3.2) since $\mathbf{X}^{\mathrm{det}}_{R,S} \times \mathbf{Y}$ and $\alpha^{\mathrm{det}}_{R,S} = \alpha_{R,S} \times s$ where s is the right shift on $\mathbf{Y}$.

Remark 3.2.2 The case of a strongly normalising rewriting system is interesting to consider. In that case, the different models obtained are equivalent in terms of computability: whichever strategy S is considered, the term t will normalise to the same unique normal form t'. However, this equivalence is not quantitative: if one takes into account some notion of cost – for instance the length of reduction – then normalising t may be much more efficient in the model corresponding to a strategy S than in the model built from strategy S'.

Example 3.2.11 One interesting case akin to rewriting is that of cellular automata. Let us start with a space $\mathbf{X}_{c.a.}$, together with a notion of *neighbourhood*, i.e. for all $x \in \mathbf{X}_{c.a.}$, there exists a subspace $V(x)$ called the *neighbourhood of* x, and write π_x the projection $\mathbf{X}_{c.a.} \to V(x)$. Suppose moreover that for any x, $V(x)$ is isomorphic to a generic neighbourhood V and fix a specific isomorphism ϕ_x. For any alphabet Σ, one can define a local-to-global Σ-automata of neighbourhood V by providing a finite set of rules $r : \Sigma^V \to \Sigma$. For any element $c \in \Sigma^{\mathbf{X}_{c.a.}}$, this finite set of rules defines a single instruction $I_r : \Sigma^{\mathbf{X}_{c.a.}}$ defined by:

$$\Sigma^{\mathbf{X}_{c.a.}} \xrightarrow{\Sigma^{\phi_x \circ \pi_x}} (\Sigma^V)^{\mathbf{X}_{c.a.}} \xrightarrow{r^{\mathbf{X}_{c.a.}}} \Sigma^{\mathbf{X}_{c.a.}}$$

This defines a monoid action $\alpha_{\mathrm{c.a.}}$ from the free monoid over the set of rules $(\Sigma^V \to \Sigma)^* \curvearrowright \Sigma^{\mathbf{X}_{c.a.}}$ which generalises the cellular automata model. This will be discussed with more details in the next section where we will analyse the corresponding set of programs (Definition 4.2).

Example 3.2.12 We now consider automata networks models. Let (V, E, s, t) be a directed (simple) graph, endowed with local update rules

$$U : v \in V \mapsto U(v) : \{0,1\}^{s(t^{-1}(v))} \to \{0,1\}.$$

The corresponding abstract model of computation is defined by $\mathbf{X} = \{0,1\}^V$, and the action α is defined by the single instruction `update` realised by $\alpha(\texttt{update})$ which maps $\phi : v \mapsto c$ to:

$$\left(v \mapsto U(v)(\phi(a_1), \phi(a_2), \ldots, \phi(a_k))\right),$$

where $\{a_1, a_2, \ldots, a_k\} = s(t^{-1}(v))$.

Functional programming

In all the examples above, the monoid action is generated by a single instruction. But some particular models are more interesting, namely cases in which the terms themselves are not only the data but also the programs. In order to do so, it is however needed that some constructions on terms corresponds to *applying one term to the other*. In case of lambda-calculus, this is very clearly defined as the application rule. I.e. applying a term t to another term u has a dual nature: it can be understood as constructing a new term $(t)u$, or as considering t as a program to which we are giving u as argument. While these two aspects are usually conflated, they here appears clearly separated.

Indeed, application allows one to lift the monoid action generated by β-reduction to an action of terms on themselves. More precisely: let Λ be the set of lambda-terms, and write $\beta : \mathbb{M}(\beta) \curvearrowright \Lambda$. The application lifts β to an action $\alpha : \mathbb{M}(\Lambda) \curvearrowright \Lambda$ by letting $\alpha(t)(u) = \mathrm{nf}((t)u)$ when the latter exists, and $\alpha(t)(u)$ undefined otherwise. Note that in this case, one can use the definition of β-reduction to deduce some equalities. For instance in a leftmost reduction setting: $\alpha(((\lambda x).t)u)(v) = \alpha(t[x := u])(v)$.

More generally, any term construction operator that allows to combine two (or more) terms can be used to lift the action of a rewriting system to an action of the monoid of terms.

Example 3.2.13 (Lambda-calculus) Now, lambda-calculus is an example of a more general rewriting system. Beta-reduction together with a reduction strategy defines an abstract model of computation in the same way as described above, as long as one is careful about the definition of beta-reduction. Indeed, the full definition of β-reduction requires α-renaming of closed terms with a 'fresh' variable. This fresh variable should be deterministically chosen to properly define the rewriting system; this is however easily done by considering a total order on variables and choosing the smallest variable that does not appear in the term. I.e. one needs to determine exactly the substitution.

This can be done as follows. We fix a total order on the variable names: we will write those as $x_1, x_2, \ldots$. We then define inductively $t[x_i := s]$ as:

- s if $t = x_i$;
- x_j if $t = x_j \neq x_i$;
- $(u[x_i := s])v[x_i := s]$ if $t = (u)v$;
- $\lambda x_i.u$ when $t = \lambda x_i.u$;
- $\lambda x_j.u[x_i := s]$ when $t = \lambda x_j.u$ with $x_j \neq x_i$ and $x_j \notin \mathrm{FreeVar}(u)$;
- $\lambda x_n.(u[x_j := x_n])[x_i := s]$ when $t = \lambda x_j.u$ with $x_j \neq x_i$ and $x_j \in \mathrm{FreeVar}(u)$, where x_n is the smallest variable such that $x_n \notin \mathrm{Variables}(u) \cup \mathrm{Variables}(s)$;

This definition can be used to define a map on lambda-terms $\to_{\beta,S} : \Lambda \to \Lambda$ for each reduction strategy S that determine which redex should be reduced at each step. The induced dynamical system can then be lifted, through application, to an action of[6] Λ onto itself:

$$\Lambda \curvearrowright \Lambda : t \mapsto u \mapsto \mathrm{nf}((t)u).$$

6: We in fact get an action of $\Lambda*$, the free monoid over Λ, similarly to the case of cellular automata (Example 3.2.11). This is discussed later but mainly implies that lambda-calculus is represented by *stateless programs* (Definition 4.2.4).

This defines an AMC $\alpha_\lambda(S) : \Lambda^* \curvearrowright \Lambda$.

Obviously, abstract machines can also used to define abstract models of computation. We here only detail the construction for the Krivine abstract machine [75], but other machines – such as the SECD [76, 77] – lead to the definition of an AMC in a similar manner.

[75]: Krivine (2007), *A call-by-name lambda-calculus machine*

Example 3.2.14 Recall the mutually recursive definitions of closures $c \in \mathcal{C}$ and environments $e \in E$:

$$c := (t, e) \qquad e := \epsilon \mid [x \leftarrow c] :: e$$

We also recall the definition of stacks $\pi \in \mathcal{S}$:

$$\pi := \alpha \mid c \cdot \pi.$$

The states $s := (c, \pi)$ of the Krivine Abstract Machine are made of a closure $c \in \mathcal{C}$ and a stack $\pi \in \mathcal{S}$. We will write states as $s := (t, e) \star \pi$ to specify the components of the closure. Transitions of the machine are defined as:

$$\begin{aligned} ((t)u, e) \star \pi &\to (t, e) \star (u, e) \cdot \pi \\ (\lambda x.t, e) \star c \cdot \pi &\to (t, [x \leftarrow c] \cdot e) \star \pi \\ (x, e) \star \pi &\to (t, e') \star \pi \qquad \text{if } e \text{ contains } [x \leftarrow (t, e')] \end{aligned}$$

We thus choose the space

$$\mathbf{X}_{\text{KAM}} = \{(t, e) \star \pi \mid (t, e) \in \mathcal{C}, \pi \in \mathcal{S}\}.$$

This is equipped with the action of a single partially-defined map $\to$ defined by the above equations (note they are partially defined on disjoint domains). This can be extended with a partially defined map `isNF` acting as a partial identity on the subspace of terms for which $\to$ is not defined.

This can then be used to define the orbit representing the computation of a λ-term: a term t and a closure c give rise to the orbit starting at $t \star \pi$.

This example of the Krivine abstract machine naturally leads to the definition of an AMC corresponding to the *processes* used as the underlying model of computation of Krivine's *classical realisability* [78, 79]. In view of the *linear realisability* models presented later, this example is particularly interesting and could lead (though not in this document) to a better understanding of the relationship between classical and linear realisability constructions.

[78]: Krivine (2001), *Typed lambda-calculus in classical Zermelo-Fraenkel set theory*
[79]: Krivine (2009), *Realizability in classical logic*

Example 3.2.15 We co-inductively define the spaces of stacks Π and the set Λ of λ-terms containing stack constants:

$$\begin{aligned} &(\Lambda)\ t := x \mid k_\pi \mid (t)t \mid \lambda x.t && (x \in \text{Var}, \pi \in \Pi, \mathrm{c} \in C) \\ &(\Pi)\ \pi := \alpha \mid t \cdot \pi && (t \in \Lambda, \alpha \in \mathcal{B}) \end{aligned}$$

where Var and $\mathcal{B}$ are countably infinite sets of variables and stack constants respectively and C is a set of instructions containing cc.

One can then form the space of *processes*:

$$\mathbf{X}_{\text{KR}} = \Lambda \times \Pi,$$

whose elements are written as $t \star \pi$ for $t \in \Lambda$ and $\pi \in \Pi$.

The four basic reduction rules considered by Krivine are the following:

$$\begin{aligned}
(\texttt{push})\ & (t)u \star \pi \succ t \star u \cdot \pi \\
(\texttt{grab})\ & (\lambda x.t) \star u \cdot \pi \succ t[x := u] \star \pi \\
(\texttt{save})\ & \text{cc} \star t \cdot \pi \succ t \star k_\pi \cdot \pi \\
(\texttt{restore})\ & k_\pi \star t \cdot \pi' \succ t \star \pi
\end{aligned}$$

They can be extended with rules governing the reduction of additional instructions in the chosen set C.

These rules define partial maps with disjoint domains, and therefore combine in a single (partial) map $\alpha_{\text{KR}}(\succ)$ acting on $\mathbf{X}_{\text{KR}}$. In a similar way as before, this can be extended with a partial identity map `isNF` whose domain of definition is the set of all processes for which none of the rules above can be applied.

This lifts to an action of Λ onto $\mathbf{X}_{\text{KR}}$ which associates to each $t \in \Lambda$ and $(u, e) \star \pi \in \mathbf{X}_{\text{KR}}$ the orbit of $((t)u, e) \star \pi$.

Neural networks

We will here explain how to represent the set of neural networks and the training process. We will however need to use notions which will only be introduced in the next chapter, i.e. the notion of program. Indeed, the space of configuration of the training algorithm should be defined as the set of programs in an underlying model, turning the program (usually called *model* in the learning community) into a possible state of the system. For simplicity, we will consider feedforward networks only. We will also consider the size of all layers to be equal to a fixed integer: this is not a restriction of the model since the model allows for networks using only part of a layer.

Example 3.2.16 The first level of the architecture is that of neural networks as a model of computation. I.e. we fix an integer k that represents the maximum size of the layers in the models, and the space considered is

$$\mathbf{X}^{\mathbf{0}}(k) = \mathbf{C}^k$$

Instructions are given by matrices of the right size, i.e. $k \times k$ matrices. These matrices describe the weights of edges between two consecutive layers (or hyperparameters). We will not discuss here the possible choices of threshold functions and consider this choice fixed as part of the model. Based on this choice and a matrix, it is possible to propagate the value of a vector from one layer to the next: given a vector $\vec{v} \in \mathbf{X}^{\mathbf{0}}(k)$ and a matrix $M \in \mathcal{M}_k(\mathbf{C})$, we can compute the value[7] $\{M\}(\vec{v})$.

Now, anticipating on the next section, an abstract program for this model of computation is a general (recurrent) network. But one can consider only feedforward networks by restricting the control structure to partial

7: Notice the notation which is meant to stress that the computation is not in general the simple application of M to the vector $\vec{v}$.

orders (or even total orders); we will write **ffN** the set of such abstract programs (for feedforward networks).

The process of training is then represented as an abstract model of computation built on the space ffN. More precisely, we consider the space:

$$\mathbf{ffN} \times (\mathbf{C}^k \times \mathbf{C}^k)^{|\omega|},$$

where the space $(\mathbf{C}^k \times \mathbf{C}^k)^\omega$ represents finite sequences of pairs consisting of an input vector (first element of the pair) and an expected outcome (second element of the pair). This model is then defined, in the simplest cases, by considering a single instruction `update` which takes an elements $P \in \mathbf{ffN}$ and a pair $(i, o) \in \mathbf{C}^k \times \mathbf{C}^k$ and produces $\texttt{update}(P, (i, o))$ (usually by performing back-propagation, i.e. one evaluates the output $P\{i\}$ and propagates the error $P\{i\} - o$ in the program to update the hyperparameters). The action is thus generated by the following endomorphism:

$$\alpha(\texttt{train})(P, \mathrm{s}) = (\texttt{update}(P, (i, o)), \mathrm{s}') \text{ if } \mathrm{s} = (i, o) \cdot \mathrm{s}'.$$

Computer architectures

Up to this point, all models considered are taken from theory. While Turing machines play a fundamental role in computer science, it is still a much theoretical device with important differences from actual computer architectures. To argue that our mathematical definition of model of computation does not only apply to abstract models, we now provide two examples of more realistic models of computation. To do so, we will show how to represent two *instruction set architectures*. For simplicity, and for its historical importance, the first example will be that of the EDSAC (Electronic Delay Storage Automatic Calculator) instruction set (Example 3.2.17). We will then sketch the interpretation of a more up-to-date example: the ARMv8-A instruction set, chosen because it is the instruction set employed on the CPU of the computer that was used to write this document. We will later discuss how those can be compared to the more theoretical models introduced above, and how these models can be of interest for more refined complexity analysis[8] (chapter 13).

8: For instance introducing cost models distinguishing access costs for different level of caches, or speculation mechanisms.

EDSAC Order Code

Order	Explanation
A n	Add the number in storage location *n* into the accumulator.
S n	Subtract the number in storage location *n* from the accumulator.
H n	Transfer the number in storage location *n* into the multiplier register.
V n	Multiply the number in storage location *n* by the number in the multiplier register and add into the accumulator.
N n	Multiply the number in storage location *n* by the number in the multiplier register and subtract from the contents of the accumulator.
T n	Transfer the contents of the accumulator to storage location *n*, and clear the accumulator.
U n	Transfer the contents of the accumulator to storage location *n*, and do not clear the accumulator.
C n	Collate the number in storage location *n* with the number in the multiplier register, i.e., add a 1 into the accumulator in digital positions where both numbers have a 1 and a 0 in other digital positions.
R 2^{n-2}	Shift the number in the accumulator *n* places to the right, i.e., multiply it by 2^{-n}.
L 2^{n-2}	Shift the number in the accumulator *n* places to the left, i.e., multiply it by 2^{n}.
E n	If the number in the accumulator is greater than or equal to zero, execute next the order which stands in storage location *n*; otherwise, proceed serially.
G n	If the number in the accumulator is less than zero, execute next the order which stands in storage location *n*; otherwise, proceed serially.
I n	Read the next row of holes on the tape, and place the resulting 5 digits in the least significant places of storage location *n*.
O n	Print the character now set up on the teleprinter, and set up on the teleprinter the character represented by the five most significant digits in storage location *n*.
F n	Place the five digits which represent the character next to be printed by the teleprinter in the five most significant places of storage location *n*.
Y	Round off the number in the accumulator to 34 binary digits.
Z	Stop the machine, and ring the warning bell.

Figure 3.2.: EDSAC order code [80]

© University of Cambridge

Example 3.2.17 (EDSAC) The description below is based on the paper by Wilkes and Renwick [80] describing the EDSAC and the documentation [81], and archives [82] of the EDSAC Replica project [9]

The EDSAC possessed 1024 locations, each containing 18 bits of which the first was unusable. It allowed for double words of 35 bits by combining two adjacent storage locations. Each size of data (either 17 or 35 bits) could either represent a signed integer or a signed fraction[10]. Short tanks containing only one number were used as accumulator and multiplier registers in the arithmetical unit, and for control purposes. The accumulator could hold 71 bits to perform exact multiplications. It also contained a sequence control register (program counter) of 10 bits, an order tank (only representing the current order) of 17 bits. The programs were provided to the machine as lists of lines consisting of 5 bits; similarly, the output was composed of lines of 5 bits.

The instructions were represented by 17 bits codes: 5 first bits coding the letter of the instruction, followed by one spare bit, followed by ten bits representing a memory address, and finally a bit indicating whether the instruction should operate on a word or a double word.

The space of configuration is therefore the following:

$$\mathbf{X}_{\text{EDSAC}} = (\{0,1\}^5)^\omega \times (\{0,1\}^{17})^{1024} \times \{0,1\}^{71} \times \{0,1\}^{10} \times \{0,1\}^{35} \times (\{0,1\}^5)^\omega,$$

where $\{0,1\}^{35}$ represents the accumulator and $\{0,1\}^{71}$ represents the

[80]: Wilkes et al. (1950), *The EDSAC (Electronic delay storage automatic calculator)*

[81]: project (n.d.), *Tutorial Guide to the EDSAC Simulator*

[82]: authors (1948), *The EDSAC: general description*

9: `https://www.tnmoc.org/edsac`,

10: Approximations of real numbers were represented as elements of $[-1,1]$.

multiplier. Elements of $\mathbf{X}_{\text{EDSAC}}$ will be represented as tuples

$$(\mathrm{I}, \mathrm{Mem}, \mathrm{pc}, \mathrm{Acc}, \mathrm{mul}, \mathrm{O}),$$

where pc represents the program counter, Acc the accumulator, mul the multiplicand, Mem the main memory, and I and O account for the input and output.

Now, the EDSAC had 17 instructions, shown in Figure 3.2, each represented as a single letter. We will consider the following conventions:

- all operations are performed on the 35 bits memory and in case of overflow the computation continues with the contents of the registers without giving notice,
- the operation $\wedge$ denotes the bitwise conjonction of two strings,
- the operations $\times 2^{-n}$ and $\times 2^{n}$ represent respectively right and left shift of n bits,
- the operations $+2^{35}$ and $+2^{17}$ represent the addition of 1 to rounding to 34 and 16 digits,
- the sequence $\mathrm{I}_{>0}$ is the sequence obtained from I by removing the first element,
- $\cdot$ denotes the concatenation of an element to the head of a list or the concatenation of two sequences of bits.

We can represent the instructions as the following actions[11], which we express here on the input $\mathbf{x} = (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}, \mathrm{Acc}, \mathrm{mul}, \mathrm{O})$:

11: We here only represent the case of instructions used on words, the case of double words is obtained as a straightforward adaptation of those.

$$\begin{aligned}
&\alpha_{\text{EDSAC}}(\texttt{A}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc} + \mathrm{Mem}[n], \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{S}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc} - \mathrm{Mem}[n], \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{H}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc}, \mathrm{Mem}[n], \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{V}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc} + \mathrm{mul} \times \mathrm{Mem}[n], \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{N}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc} - \mathrm{mul} \times \mathrm{Mem}[n], \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{T}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}[n := \mathrm{Acc}], \mathrm{pc}+1, 0, \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{U}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}[n := \mathrm{Acc}], \mathrm{pc}+1, \mathrm{Acc}, \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{C}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{mul} \wedge \mathrm{Mem}[n], \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{R}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc} \times 2^{-n}, \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{L}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc} \times 2^{n}, \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{E}\ n) : \begin{array}{l}\mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, n, \mathrm{Acc}, \mathrm{mul}, \mathrm{O}) \text{ if } \mathrm{Acc} \geq 0\\ \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc}, \mathrm{mul}, \mathrm{O}) \text{ otherwise}\end{array}\\
&\alpha_{\text{EDSAC}}(\texttt{G}\ n) : \begin{array}{l}\mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, n, \mathrm{Acc}, \mathrm{mul}, \mathrm{O}) \text{ if } \mathrm{Acc} \leq 0\\ \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc}, \mathrm{mul}, \mathrm{O}) \text{ otherwise}\end{array}\\
&\alpha_{\text{EDSAC}}(\texttt{I}\ n) : \mathbf{x} \mapsto (\mathrm{I}_{>0}, \mathrm{Mem}[n := 0^{12} \cdot \mathrm{I}[0]], \mathrm{pc}+1, \mathrm{Acc}, \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{O}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc}, \mathrm{mul}, \mathrm{Mem}[n] /\!\!/ 2^{12} \cdot \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{F}\ n) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}[n := \mathrm{O}[0]], \mathrm{pc}+1, \mathrm{Acc}, \mathrm{mul}, \mathrm{O})\\
&\alpha_{\text{EDSAC}}(\texttt{Z}\ 1) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc} + 2^{35}, \mathrm{mul}, \mu)\\
&\alpha_{\text{EDSAC}}(\texttt{Z}\ 2) : \mathbf{x} \mapsto (\mathrm{I}, \mathrm{Mem}, \mathrm{pc}+1, \mathrm{Acc} + 2^{17}, \mathrm{mul}, \mu).
\end{aligned}$$

Note that one aspect of the EDSAC is that self-modifying code was not only tolerated but useful. The dynamics of an execution are given by the following. From a given configuration $\mathbf{x}$, the machine moves the contents of Mem[pc] into the order tank for it to be executed in the next step, then

it proceeds to execute the instruction it represents. I.e. the dynamics are given by:

$$\mathbf{x} \mapsto \alpha_{\text{EDSAC}}(\text{Mem}[\text{pc}])(\mathbf{x}),$$

with initial configuration[12] $(\text{I}, \text{Mem}, 0, \text{Acc}, \text{mul}, \emptyset)$.

12: We suppose here that a computation starts with possible non-zero values in the main memory, accumulator, and multiplier.

Example 3.2.18 (ARMv8-A) We now discuss how to represent the ARMv8-A instruction set. We will not provide all details because it would take at least a hundred pages to write it down, but we will provide enough elements to convince writing down the full interpretation is possible in theory. The information below is mostly taken from the ARMv8-A documentation [13].

13: `https://developer.arm.com/documentation/ddi0487/latest/`

To do so, we first provide some information on the corresponding architecture. The instruction set allows for 32 all-purpose 64 bits registers with the first register always set to zero, and 32 floating point 128 bit registers. These will be represented as $\mathbf{APR} = \{0^{64}\} \times (\{0,1\}^{64})^{31}$ and $\mathbf{FPR} = (\{0,1\}^{128})^{32}$ respectively. In addition, there is a Program Counter represented as $\mathbf{PC} = \{0,1\}^{32}$, System Registers[14] represented as **SR**.On top of this, the machine has access to the main memory. We can consider two version: the idealised version in which the memory is unbounded, and a more realistic, limited, memory of, say, 16Go memory. Those will be represented as $\mathbf{Mem}_\infty = \{0,1\}^\omega$ and $\mathbf{Mem}_{16\text{Go}} = \{0,1\}^{2e37}$ respectively.

14: System registers will not be detailed here; documentation lists 276 AArch32 system registers, 561 AArch64 system registers, and 540 external registers.

Now, considering the space[15]

$$\mathbf{ARM} = \mathbf{APR} \times \mathbf{FPR} \times \mathbf{PC} \times \mathbf{SR} \times \mathbf{Mem}_\kappa$$

15: This does not account for possible simplifications based on the fact that the first register is never modified (and always equal to 0), and the program counter is usually part of the 32 registers.

one can define the actions corresponding the instructions allowed by the instruction set architecture (ISA). Detailing this would be a tedious task: the instruction set comprises[16] 472 base instructions, 425 floating point instructions, 921 SVE instructions, and 310 SME instructions. However it should be clear to the reader that, if considered with enough care, each instruction gives rise to an endomorphism of **ARM**, and that the collection of all these endomorphisms generates a monoid action modelling the ISA.

16: If I counted correctly. These numbers also include multiplie variations of the same instruction.

In fact, this model is but a rough approximation of how an actual processor works. To be more precise, one should take into account at the very least the architecture of cache memories and their exclusion policy, as well as model the prediction mechanism. Providing a more realistic model of computation allowing to obtain results in algorithmics and complexity which are coherent with current architectures is one of my projects for the years to come. This will be discussed in chapter 13.

A more tractable example to be considered in the future is that of RISC-V. Restricting to the base instructions should remain tractable (there are 47 such instructions).

Analog computers

One other interesting example is the General Purpose Analog Computer (GPAC) model. While it is sometimes considered as a continuous time model of computation, it can still be described abstractly as an AMC. Indeed, it can be seen as a circuit composed of basic instructions performing operations on maps $\mathbf{R} \to \mathbf{R}$. This does not imply that the GPAC does not

possess continuous-time aspects, but those appear in stabilisation time, etc. In essence, the model of computation still is discrete in that each instruction is applied or not, differing from models in which an instructions could be applied for a (real) variable amount of time. As a consequence, we still see the GPAC as a discrete-time model of computation acting on a continuous space.

Example 3.2.19 The general purpose analog computer was introduced by Shannon [83] as a mathematical model of analog computers. While presented as a continuous-time model, it is in fact described by circuits, and can be modelled by a monoid action. We let the space of configurations be $\mathbf{X} = \mathcal{C}^\infty(\mathbf{R}, \mathbf{R})^\omega$, i.e. a countable product of the space of smooth functions over the reals. The product will be used to manipulate multiple inputs, i.e. they act as registers. Basic units of the GPAC model are: the constant unit, introducing the constant function equal to some value k, adder and multiplier units, and an integrator unit. We therefore consider the following generating set of instructions:

[83]: Shannon (1941), *Mathematical Theory of the Differential Analyzer*

- $\texttt{const}_{i_0}(k) : \mathbf{X}^\omega \to \mathbf{X}^\omega, (f_i) \mapsto (g_i)$ where $g_{i_0} = k$ and $g_i = f_i$ if $i \neq i_0$;
- $\texttt{add}_{i_0}(j_1, j_2) : \mathbf{X}^\omega \to \mathbf{X}^\omega, (f_i) \mapsto (g_i)$ where $g_{i_0} = f_{j_1} + f_{j_2}$ and $g_i = f_i$ if $i \neq i_0$;
- $\texttt{mult}_{i_0}(j_1, j_2) : \mathbf{X}^\omega \to \mathbf{X}^\omega, (f_i) \mapsto (g_i)$ where $g_{i_0} = f_{j_1} \times f_{j_2}$ and $g_i = f_i$ if $i \neq i_0$;
- $\texttt{int}_{i_0}(j_1, j_2) : \mathbf{X}^\omega \to \mathbf{X}^\omega, (f_i) \mapsto (g_i)$ where $g_{i_0} = \int f_{j_1} df_{j_2}$ and $g_i = f_i$ if $i \neq i_0$.

It is then possible to represent GPAC machines as graphings w.r.t. this action.

Remark 3.2.3 The discussion above however raises the question of how one could represent non-discrete instructions. One natural generalisation of AMC would consider the replacement of monoid actions by *operator semi-groups* [84]. We however will not consider this extension here.

[84]: Davies (1980), *One-parameter Semigroups*

3.3. Computational equivalences

We will now define a few notions of equivalences of AMC, provide examples of such, and prove some properties. The strongest equivalence is the notion of isomorphism. It will become clearer as we investigate further the notions of equivalence that the core notion is rather that of *simulation*. Indeed, the most important results concern the simulation (either computational – Definition 3.3.3, or program-wise simulations – Definition 4.4.2 and Definition 4.4.5) of an AMC by another. Equivalences, defined as two-way simulations, is usually a less natural notion.

Isomorphisms and retraction: equivalences of presentations

Definition 3.3.1 (Isomorphism) *Two models of computation* $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ *and* $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ *are* isomorphic, *written* $\alpha \cong \beta$, *when there exists a*

isomorphism $\varphi : \mathbf{X} \to \mathbf{Y}$ *and a bijection* $\theta : I \to J$ *such that* $\varphi \circ \alpha(i) = \beta(\theta(i)) \circ \varphi$.

This notion of equivalence is particularly strong, and somehow captures modifications in some of the parameters and conventions in the definition of the machine model. For instance, the following model is equivalent to the AMC α_{TM} of Turing machines.

Example 3.3.1 (Definition of Turing machines over $\{-1, 1\}$) Define the $\alpha_{\text{TM}(\mathbf{Z}_2)} : \mathbb{M}(\text{Instr}_{\text{TM}}) \curvearrowright \mathbf{X}_{\text{TM}(\mathbf{Z}_2)}$ by:

$$\mathbf{X}_{\text{TM}(\mathbf{Z}_2)} = \{-1, 1, 0\}_0^{\mathbf{Z}},$$

that is bi-infinite sequences of -1, 1, and 0, which are almost always equal to 0. The action is defined as:

- $\alpha_{\text{TM}(\mathbf{Z}_2)}(\texttt{right}) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_{i+1})_{i\in\mathbf{Z}}$;
- $\alpha_{\text{TM}(\mathbf{Z}_2)}(\texttt{left}) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_{i-1})_{i\in\mathbf{Z}}$;
- $\alpha_{\text{TM}(\mathbf{Z}_2)}(\texttt{write}_\star) : (s_i)_{i\in\mathbf{Z}} \mapsto (\bar{s}_i)_{i\in\mathbf{Z}}$ where $\bar{s}_0 = *$ and $\bar{s}_i = s_i$ when $i \neq 0$;
- $\alpha_{\text{TM}(\mathbf{Z}_2)}(\texttt{read}_\star) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_i)_{i\in\mathbf{Z}}$ when $\bar{s}_0 = *$ and undefined otherwise.

Weakening conditions on the space. Note that this is the strongest possible equivalence: the underlying spaces are isomorphic, and the actions of the sets of instructions (which generate the monoid action) are conjugate. It is therefore in some sense a strengthening of the notion of conjugate actions (this will be formalised below). This definition can be weakened along two, quite orthogonal, dimensions. The first is to allow for a less constrained mapping of instructions: an instruction in I could be mapped to an element of $\mathbb{M}(J)$ instead of an element of J. This leads to the notion of *simulation* and *equivalence* defined below. The second dimension is that of the space. We will start by discussing this, with an example.

Example 3.3.2 One example is the following alternative definition of an AMC $\tilde{\alpha}_{\text{TM}}$ of Turing machines. Consider the space $\tilde{\mathbf{X}}_{\text{TM}} = \mathbf{X}_{\text{TM}} \times \mathbf{Z}$ where the second component represents the position of the head on the tape. The action is here induced by:

- $\tilde{\alpha}_{\text{TM}}(\texttt{right}) : ((s_i)_{i\in\mathbf{Z}}, h) \mapsto ((s_i)_{i\in\mathbf{Z}}, h + 1)$;
- $\tilde{\alpha}_{\text{TM}}(\texttt{left}) : ((s_i)_{i\in\mathbf{Z}}, h) \mapsto ((s_i)_{i\in\mathbf{Z}}, h - 1)$;
- $\tilde{\alpha}_{\text{TM}}(\texttt{write}_\star) : ((s_i)_{i\in\mathbf{Z}}, h) \mapsto ((\bar{s}_i)_{i\in\mathbf{Z}}, h)$ where $\bar{s}_h = \star$ and $\bar{s}_i = s_i$ when $i \neq h$;
- $\tilde{\alpha}_{\text{TM}}(\texttt{read}_\star) : ((s_i)_{i\in\mathbf{Z}}, h) \mapsto ((s_i)_{i\in\mathbf{Z}}, h)$ when $\bar{s}_h = \star$ and undefined otherwise.

We will now compare $\tilde{\alpha}_{\text{TM}}$ with α_{TM}. Intuitively, configurations in $\tilde{\alpha}_{\text{TM}}$ are redondant: writing σ the usual left shift on $\mathbf{X}_{\text{TM}}$, we can understand the two configurations $((s_i)_{i\in\mathbf{Z}}, h)$ and $(\sigma((s_i)_{i\in\mathbf{Z}}), h - 1)$ as representing the same situation: the contents of the tape are the same up to a shift, and the tape head points to the same position – up to the same shift. This is captured by the fact that the action of each instruction is invariant w.r.t.

conjugation by τ. More formally:

$$\begin{aligned}
\tilde{\alpha}_{\text{TM}}(\texttt{right}) &= \tau^{-1} \circ \tilde{\alpha}_{\text{TM}}(\texttt{right}) \circ \tau \\
\tilde{\alpha}_{\text{TM}}(\texttt{left}) &= \tau^{-1} \circ \tilde{\alpha}_{\text{TM}}(\texttt{left}) \circ \tau \\
\tilde{\alpha}_{\text{TM}}(\texttt{write}_\star) &= \tau^{-1} \circ \tilde{\alpha}_{\text{TM}}(\texttt{write}_\star) \circ \tau \\
\tilde{\alpha}_{\text{TM}}(\texttt{read}_\star) &= \tau^{-1} \circ \tilde{\alpha}_{\text{TM}}(\texttt{read}_\star) \circ \tau
\end{aligned}$$

The AMC $\tilde{\alpha}_{\text{TM}}$ is therefore an equivariant system w.r.t. the action of $\mathbf{Z}$ by $k \mapsto \sigma^k$, and the AMC α_{TM} can be identified with the quotient of $\tilde{\alpha}_{\text{TM}}$ by this action.

Definition 3.3.2 *Consider two models of computation $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$. We say that α is a* retract *of β when there exists $\tilde{\mathbf{X}}$ and a bijection $u : \tilde{\mathbf{X}} \to \tilde{\mathbf{X}}$ such that $\alpha \times u \curvearrowright \mathbf{X} \times \tilde{\mathbf{X}}$ and β are isomorphic.*

Theorem 3.3.1 *The AMC α_{TM} is a retract of $\tilde{\alpha}_{TM}$.*

Proof. The proof is quite easy: a simple projection of $\tilde{\mathbf{X}}_{\text{TM}}$ into $\mathbf{X}_{\text{TM}}$ works out with the map sending each instruction in the first model to the same instruction in the second model. This embedding is defined as:

$$((s_i)_{i \in \mathbf{Z}}, h) \mapsto (s_{i-h}).$$

It is then easy to show that for each $(\mathbf{s}, h) \in \tilde{\mathbf{X}}_{\text{TM}}$ and all instruction i:

$$\tilde{\alpha}_{\text{TM}}(i)(\mathbf{s}, h) = (\alpha_{\text{TM}}(i)(\mathbf{s}), h),$$

which shows the result. ❀

We note that while $\alpha_{\text{stack}[2]}$ is close to being a retract of α_{TM}, it is not. The reason for this is that on the tape of a Turing machine, there could be several strings in $\{0,1\}^*$ separated by blank symbols. Similarly, the AMC α_{FA} is close to being a retract of α_{TM} restricted to the instructions $\{\texttt{read}_\star, \texttt{read}_0, \texttt{read}_1, \texttt{right}\}$, but it is not for similar reasons.

As a second example, we have the two distinct AMCs defined from a rewriting system equipped by a deterministic strategy, as discussed in Remark 3.2.1.

Simulation and computational equivalence

Weakening conditions on the monoid. We now introduce another weakening of the notion of isomorphism: computational equivalence. Equivalence is defined as pairwise simulation, where simulation is defined as follows. This notion is quite natural for the computer scientists. In the following definition we will use the notion of *algebraic degree*: if M is a monoid and I is a generating set, the algebraic degree $\deg_I(m)$ of $m \in M$ w.r.t. I is defined as the smallest $k \in \mathbf{N}$ such that there exists $a_1, \ldots, a_k \in I^k$ such that $m = a_1 a_2 \ldots a_k$.

Definition 3.3.3 (Simulation) *Given two models of computation $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$, we say that β* simulates *α, written $\alpha \prec \beta$, when*

there exists a monomorphism $\varphi : \mathbf{X} \to \mathbf{Y}$ *and a map* $\theta : I \to \mathbb{M}(J)$ *such that* $\forall i \in I, \varphi \circ \alpha(i) = \beta(\theta(i)) \circ \varphi$.

When ϕ *is an isomorphism and* $\theta(I)$ *is a generating set for* $\mathbb{M}(J)$*, we say that* β strongly simulates α *and write* $\alpha \propto \beta$.

The degree *of the simulation is defined as* $\max_{\iota \in I} \deg_J(\theta(i))$.

Lemma 3.3.2 *Suppose* $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ *and* $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ *are such that* $\alpha \prec \beta$ *as witnessed by* $\varphi : \mathbf{X} \to \mathbf{Y}$ *and a map* $\theta : I \to \mathbb{M}(J)$*. For all element* $m \in \mathbb{M}(I)$*, we have* $\varphi \circ \alpha(m) = \beta(\theta(m)) \circ \varphi$*, where* $\theta(m)$ *denotes the monoid morphism induced by* θ.

Proof. We write $m = i_1 \dots i_k$ as a product of elements of I. Then:

$$\begin{aligned}\varphi \circ \alpha(i_1 \dots i_k)(x) &= \varphi \circ \alpha(i_1) \circ \dots \circ \alpha(i_k)(x) \\ &= \beta(\theta(i_1)) \circ \varphi \circ \dots \circ \alpha(i_k)(x) \\ &= \dots \\ &= \beta(\theta(i_1)) \circ \dots \circ \beta(\theta(i_k)) \circ \varphi(x) \\ &= \beta(\theta(m)) \circ \varphi(x).\end{aligned}$$

This shows the result. ❀

Lemma 3.3.3 *The relation* $\prec$ *(resp.* $\propto$*) is transitive.*

Proof. The proof is straightfoward. Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$, $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$, and $\gamma : \mathbb{M}(K) \curvearrowright \mathbf{Z}$ be AMCs. Suppose $\alpha \prec \beta$ (resp. $\beta \prec \gamma$), as witnessed by the maps $\phi : \mathbf{X} \to \mathbf{Y}$ (resp. $\phi' : \mathbf{Y} \to \mathbf{Z}$), and $\theta : I \to \mathbb{M}(J)$ (resp. $\theta' : J \to \mathbb{M}(K)$). Now consider the map $\phi' \circ \phi : \mathbf{X} \to \mathbf{Z}$ and define $\theta \star \theta'$ as follows: for all $i \in I$, we write $\theta(i) \in \mathbb{M}(J)$ as the product $j_0 j_1 \dots j_k$ of elements of J, and define $\theta' \circ \theta(i) = \prod_{m=0}^{k} \theta'(j_m)$ where the product is taken in $\mathbb{M}(K)$ (i.e. θ composed with the monoid morphism induced by θ'). Then, using the previous lemma, we have:

$$\phi' \circ \phi(\alpha(i)(x)) = \phi'(\beta(\theta(i))(\phi(x))) = \gamma(\theta' \circ \theta(i))(\phi' \circ \phi(x))).$$

If moreover both simulations are strong, $\phi' \circ \phi$ is an isomorphism and $\theta' \circ \theta(I)$ is a generating subset of $\mathbb{M}(K)$. ❀

Definition 3.3.4 (Equivalence) *Two models of computation* $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ *and* $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ *are* computationally equivalent*, written* $\alpha \cong \beta$*, when* α *simulates* β *and* β *simulates* α.

As an example of computational equivalence, I will first define an alternative notion of Turing machines as AMCs. This is a simple example: Turing machines working on an alphabet different from $\{0, 1\}$. We will use the alphabet $\{0, 1\}^k$. A second example of multi-tape Turing machines will be detailed in the next chapter (Proposition 4.4.5) because the corresponding AMC is only program-wise simulated.

The first AMC $\alpha_{\text{TM}(2^k)}$ is easy to define as an adaptation of the AMC α_{TM} of Turing machines over $\{0, 1\}$. We consider the space

$$\mathbf{X}_{\text{TM}(2^k)} = \{(s_i)_{i \in \mathbf{Z}} \mid s_i \in \{0, 1\}^k \cup \{*\}, \exists N \in \mathbf{N}, |i| < N \Rightarrow s_i = *\},$$

and we define the following instructions acting on $\mathbf{X}_{\text{TM}(2^k)}$:

- (left) $\alpha_{\text{TM}(2^k)}(\texttt{left}) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_{i+1})_{i\in\mathbf{Z}}$;
- (right) $\alpha_{\text{TM}(2^k)}(\texttt{right}) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_{i-1})_{i\in\mathbf{Z}}$;
- (write) for all $\star \in \{0,1\}^k$, $\alpha_{\text{TM}(2^k)}(\texttt{write}_\star) : (s_i)_{i\in\mathbf{Z}} \mapsto (\bar{s}_i)_{i\in\mathbf{Z}}$ with $\bar{s}_0 = \star$ and $\bar{s}_i = s_i$ for $i \neq 0$;
- (read) for all $\star \in \{0,1\}^k$, $\alpha_{\text{TM}(2^k)}(\texttt{read}_\star) : (s_i)_{i\in\mathbf{Z}} \mapsto (s_i)_{i\in\mathbf{Z}}$ when $s_0 = \star$ and undefined otherwise.

Theorem 3.3.4 *The* AMC $\alpha_{TM(2^k)}$ *is equivalent to the* AMC α_{TM}*. More precisely,* $\alpha_{TM(2^k)}$ 1*-simulates* α_{TM} *and* α_{TM} $3k-2$*-simulates* $\alpha_{TM(2^k)}$*.*

Proof. The simulation of α_{TM} by $\alpha_{\text{TM}(2^k)}$ is straightforward. On defines the monomorphism

$$\mathbf{X}_{\text{TM}} \to \mathbf{X}_{\text{TM}(2^k)}$$

induced by the injection $\phi : \{0,1\} \to \{0,1\}^k$ defined by $0 \mapsto 0^k$ and $1 \mapsto 1^k$. The corresponding map θ is defined as follows:

$$\begin{aligned}
\theta(\texttt{right}) &= \texttt{right},\\
\theta(\texttt{left}) &= \texttt{left},\\
\theta(\texttt{write}_x) &= \texttt{write}_{\phi(x)},\\
\theta(\texttt{read}_x) &= \texttt{read}_{\phi(x)}.
\end{aligned}$$

The converse simulation is a bit more involved but easy to understand. We simply write down the symbols in $\{0,1\}^k$ directly on the tape of a standard Turing machine. The monomorphism is thus defined as $(s_i^1 s_i^2 \dots s_i^k)_{i\in\mathbf{Z}} \mapsto (\bar{s}_i)_{i\in\mathbf{Z}}$ where $\bar{s}_i = s_q^m$ with q, m the quotient and remainder of the division of i by k. The corresponding map θ is then defined as follows:

$$\begin{aligned}
\texttt{right} &\mapsto \texttt{right}^k\\
\texttt{left} &\mapsto \texttt{left}^k\\
\texttt{write}_{s_1 s_2 \dots s_k} &\mapsto \texttt{left}^{k-1} \circ \texttt{write}_{s_k} \circ \texttt{left} \circ \dots \circ \texttt{write}_{s_2} \circ \texttt{left} \circ \texttt{write}_{s_1}\\
\texttt{read}_{s_1 s_2 \dots s_k} &\mapsto \texttt{left}^{k-1} \circ \texttt{read}_{s_k} \circ \texttt{left} \circ \dots \circ \texttt{read}_{s_2} \circ \texttt{left} \circ \texttt{read}_{s_1}
\end{aligned}$$

This map has degree $3k-2$. ❀

Remark 3.3.1 The notion of algebraic degree somehow captures how complex the simulation is. Indeed, if elements of the generating set I are considered as *atomic* instructions all having unitary cost (in the sense of complexity theory, i.e. atomic instructions are considered to all be performed in a single unit of time), then the algebraic degree of the simulation expresses how many units of time are needed (at most) for simulating one instruction in the original model. However, this notion of complexity is naive and quite unexpressive. A more general approach is detailed in chapter 5.

We leave it to the reader to prove that the AMC $\alpha^{\star}_{\text{stack}[2]}$ of 2-stack machines working on the alphabet $\{0,1,\star\}$ (adapted from Example 3.2.5) computationally simulates the AMC β_{TM} of one-tape Turing machines with a separate input read-only tape (which can be defined as an adaptation of the AMC defined in Example 5.4.2). This simulation has algebraic

degree 3. We also leave it to the reader to convince themselves that no computational simulation of $\alpha^{\star}_{\text{stack}[2]}$ by β_{TM} exists, since the pop and push operations require a sequence of instructions that depends on the current length of the stacks. It is however possible to prove that $\alpha^{\star}_{\text{stack}[2]}$ is *program-wise intentionally simulated* (Definition 4.4.2) by β_{TM}.

3.4. Mathematical equivalences

Conjugation. Conjugation is a mathematical notion of equivalence that, as such, does not involve the generating set of the monoids.

Definition 3.4.1 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be abstract models of computation. We say that α and β are* conjugate *when there exists a monoid isomorphism $\theta : \mathbb{M}(I) \to \mathbb{M}(J)$ and a space isomorphism $\phi : \mathbf{X} \to \mathbf{Y}$ such that $\phi(g \cdot x) = \theta(g) \cdot \phi(x)$.*

A first result is the following: isomorphic models are conjugate, and conjugate models are strongly computationally equivalent.

Proposition 3.4.1 *Isomorphic AMCs are conjugate.*

Proof. By definition, if $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ are isomorphic, there exists a bijection $\theta : I \to J$ and an isomorphism $\phi : \mathbf{X} \to \mathbf{Y}$ with:

$$\forall i \in I, \forall x \in \mathbf{X}, \phi(\alpha(i)(x)) = \beta(\theta(i))(\phi(x).$$

We will show that θ induces a monoid isomorphism $\mathbb{M}(I) \to \mathbb{M}(J)$. Since I and J are generating sets, we only need to check that any relation in $\mathbb{M}(I)$ is also satisfied in $\mathbb{M}(J)$ and conversely. Suppose that w and w' are words in I^* which are identified in $\mathbb{M}(I)$. We will then show that $\beta(\theta(w)) = \beta(\theta(w'))$ which will imply that $\theta(w) = \theta(w')$. Writing $w = w_1 \dots w_n$ and $w' = w'_1 \dots w'_k$, this is shown using the above property:

$$\begin{aligned}
\beta(\theta(w))(\phi(x)) &= \beta(\theta(w_1 \dots w_n))(\phi(x)) \\
&= \beta(\theta(w_1) \dots \theta(w_n)))(\phi(x)) \\
&= \beta(\theta(w_1)) \circ \dots \circ \beta(\theta(w_n))(\phi(x)) \\
&= \beta(\theta(w_1)) \circ \dots \circ \beta(\theta(w_{n-1}))(\beta(\theta(w_n))(\phi(x))) \\
&= \beta(\theta(w_1)) \circ \dots \circ \beta(\theta(w_{n-1}))(\phi(\alpha(w_n)(x))) \\
&= \beta(\theta(w_1)) \circ \dots \circ \beta(\theta(w_{n-2}))(\beta(\theta(w_{n-1}))(\phi(\alpha(w_n)(x)))) \\
&= \beta(\theta(w_1)) \circ \dots \circ \beta(\theta(w_{n-2}))(\phi(\alpha(w_{n-1})(\alpha(w_n)(x)))) \\
&= \dots \\
&= \phi(\alpha(w_1) \circ \dots \circ \alpha(w_n)(x)) \\
&= \phi(\alpha(w_1 \dots w_n)(x)) \\
&= \phi(\alpha(w)(x)) \\
&= \phi(\alpha(w')(x)) \\
&= \phi(\alpha(w'_1) \circ \dots \circ \alpha(w'_k)(x)) \\
&= \beta(\theta(w'_1) \dots \theta(w'_k)))(\phi(x)) \\
&= \beta(\theta(w'))(\phi(x))
\end{aligned}$$

This shows that $\beta(\theta(w)) = \beta(\theta(w'))$, hence that $\theta(w) = \theta(w')$. A similar argument shows that if $\theta(w) = \theta(w')$ then necessarily $w = w'$. This proves that θ lifts to a monoid isomorphism witnessing that α and β are conjugate. ❀

Proposition 3.4.2 *Conjugate* AMCs *are (strongly) computationally equivalent.*

Proof. Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be conjugate abstract models of computation. We write $\theta : \mathbb{M}(I) \to \mathbb{M}(J)$ the monoid morphism and $\phi : \mathbf{X} \to \mathbf{Y}$ the space isomorphism such that

$$\phi(g \cdot x) = \theta(g) \cdot \phi(x). \tag{3.1}$$

It should be clear that β simulates α: for any $i \in I$ and $x \in \mathbf{X}$, we have

$$\phi \circ \alpha(i)(x) = \phi(\alpha(i)(x)) = \beta(\theta(i))(\phi(x)) = \beta(\theta(i)) \circ \phi(x).$$

Conversely, α simulates β: one can consider the map θ^{-1} restricted to J and the map $\phi^{-1} : \mathbf{Y} \to \mathbf{X}$; it then satisfies, for all $j \in J$ and $y \in \mathbf{Y}$:

$$\begin{aligned}\phi^{-1} \circ \beta(j)(y) &= \phi^{-1}(\beta(\theta(\theta^{-1}(j)))(\phi(\phi^{-1}(y))) \\ &= \phi^{-1}(\phi(\alpha(\theta^{-1}(j))(\phi^{-1}(y)))) \\ &= \alpha(\theta^{-1}(j))(\phi^{-1}(y)),\end{aligned}$$

where we used Equation 3.1 for the third equality. Moreover, this is a strong simulation: ϕ is an isomorphism and $\theta(I)$ generates $\mathbb{M}(J)$ (since θ is an isomorphism). ❀

Orbit equivalence. Orbit equivalence is an equivalence weaker than conjugation. We will see in a later section (chapter 12) how it may be related to computational complexity. But this requires additional material; we will only state the definition for the time being.

Definition 3.4.2 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be abstract models of computation. We say that α is* orbit included *in β when there exists an embedding of spaces $\Psi : \mathbf{X} \hookrightarrow \mathbf{Y}$ mapping orbits to orbits, i.e. $\forall x \in \mathbf{X}$,*

$$\Psi(\mathbb{M}(I) \cdot x) \subseteq \mathbb{M}(J) \cdot \Psi(x).$$

When Ψ is an isomorphism and the latter inclusion is an equality, we say that α is orbit equivalent *to β.*

Proposition 3.4.3 *Simulation implies orbit inclusion. Strong simulation implies orbit equivalence.*

Proof. Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ such that β simulates α, as witnessed by the embedding $\phi : \mathbf{X} \hookrightarrow \mathbf{Y}$ and $\theta : I \to \mathbb{M}(J)$. Now, consider the orbit of $x \in \mathbf{X}$ under the action of $\mathbb{M}(I)$, and consider $y = \phi(x)$. For every $i \in \mathbb{M}(I)$, we have that $\phi(i \cdot x)) = \theta(i) \cdot y$, implying that $\mathbb{M}(I) \cdot x \subset \mathbb{M}(J) \cdot \phi(x)$. This shows the first result.

When the simulation is strong, we moreover have that $\theta(I)$ is a generating set for $\mathbb{M}(J)$ and ϕ is an isomorphism. We then only need to show the

converse inclusion. Consider some $j \in \mathbb{M}(J)$. By assumption, j can be written as a $\theta(i_0) \in \mathbb{M}(I)$, implying that $\phi(i_0 \cdot x)) = \theta(i_0) \cdot y = j \cdot y$. This proves the second result. ❀

It is important to notice that, in general, computationally equivalent AMCs are not orbit equivalent. But we will define later on a notion of *weak equivalence* that is finer than both computational and orbit equivalence.

Compilation equivalence The following definition makes reference to the notion of *α-computable* partition of the space $\mathbf{X}$, where $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ is an AMC. This definition requires notions from the next chapter: a *α-computable* partition of the space $\mathbf{X}$ is a family of *α-computable* subspaces X_i of $\mathbf{X}$, where a subset $X_i \subset \mathbf{X}$ is *α-computable* if there exists an α-program P such that[17]

$$\mathbf{Gr}(P) = \{(x, x) \mid x \in X_i\}.$$

17: Here $\mathbf{Gr}(P)$ (Definition 4.3.3) is the partial function $\mathbf{X} \to \mathbf{X}$ defined by the program P.

Definition 3.4.3 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be abstract models of computation. We say that α* compiles *β when there exists an embedding $\phi : \mathbf{Y} \hookrightarrow \mathbf{X}$ such that $\forall \iota \in J$, there exists a partition $Y_1, \dots, Y_k, \dots$ of $\mathbf{Y}$ and elements $\iota_1, \dots, \iota_k, \dots$ in $\mathbb{M}(I)$ such that*

$$\forall y \in Y_i, \phi \circ \beta(\iota)(y) = \alpha(\iota_i)(\phi(y)).$$

If for each $\iota \in J$ the corresponding partition is finite (resp. α-computable), we say that α finitely (resp. computably) compiles β.

The following proposition is a direct consequence of the definitions.

Proposition 3.4.4 *If $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ simulates $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$, α compiles β.*

We will show in the next chapter that computable compilation implies program-wise intentional simulation (Proposition 4.4.4).

3.5. Relating with previous work: conditional abstract models of computation

The above definition of model of computation seems the most adequate, as it does not distinguish in an ad-hoc way between conditions – i.e. verifying that the input satisfies a given property (which should be computable – and computations. However, since some previous work used the condition version, we state a second definition in which instructions are distinguished from conditions. This definition will also be useful to draw the connection with the notions introduced in earlier papers.

Definition 3.5.1 (Conditional AMC) *A* conditional abstract model of computation *(CAMC) is a pair of a monoid action $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and a family of subspaces $\mathbf{cond} = \{\mathbf{X}_c \mid c \in C\}$ where:*

1. *$\mathbf{X}$ is a space of configurations, together with a notion of morphisms;*
2. *I is a set of instructions, generating the monoid action α;*
3. *C is a set of conditions.*

The first fact to notice is that a CAMC $(\alpha, \mathbf{cond})$ defines an AMC in a natural way: writing $\mathbf{cond} = \{\mathbf{X}_c \mid c \in C\}$, we define an abstract model of compuation $\text{AMC}(\alpha, \mathbf{cond}) : \mathbb{M}(I + C) \curvearrowright \mathbf{X}$ defined as α extended with the set C of projections $\{\pi_c \mid c \in \mathbf{cond}\}$ where π_c is defined by $\pi_c(x) = x$ if $x \in \mathbf{X}_c$ and $\pi_c(x)$ undefined otherwise. It should be clear though that not all AMC is of the form $\text{AMC}(\alpha, \mathbf{cond})$ for a CAMC $(\alpha, \mathbf{cond})$. But a slightly weaker statement can be proven, an adjunction of sorts.

Proposition 3.5.1 *Any AMC β there exists a CAMC $(\alpha, \mathbf{cond})$ such that $\beta \cong \text{AMC}(\alpha, \mathbf{cond})$.*

Example 3.5.1 Turing machines can also be defined as a CAMC. Using the notations defined in the previous example, we consider the action on $\mathbf{X}$ induced by **moveR**, `left`, and $\texttt{write}_\star$ for $* \in \{0, 1, \star\}$, together with the conditions $\{\mathbf{X}_* \mid * = 0, 1, \star\}$ where $\mathbf{X}_* = \{(s_i)_{i \in \mathbf{Z}} \mid s_0 = *\}$. One can in fact check that the AMC defined in the previous example is the AMC $\bar{\alpha}$ for the CAMC α thus defined.

Abstract machines and programs | 4.

4.1. Abstract machines as graphings.

We now have our notion of abstract models of computation. As we have seen in the examples, elements of the generating set should be thought of as instructions. The next step is to define abstract machines, i.e. some finite arrangement of such basic instructions that should be followed by a machine. One important notion here will be that of control state, which is part of the program and not of the model of computation. The reason is historical (it was used to define additive connectives in linear realisability models – see chapter 10) and one could argue that control states could be represented as part of the configuration of the machine. This is for instance what is done in Gurevich's abstract state machines [74], but it has two major drawbacks. The first is that it does not allow to compose any two machines when they use the same part of configurations to represent control states. E.g. if a machine M with at least two control states is composed with itself, but ends its computation in a different state than the initial state, composing M with itself naively will not represent the expected behaviour; composing machines therefore requires some careful renaming (which, though not impossible, seems less natural than our approach). The second, and more important aspect is that of keeping state handling out of the model of computation. This is important in view of the proposed invariant-based approach to computational complexity, but also has a philosophical importance. Indeed, while the model of computation should be thought as an idealised model of a physical device, the restriction to some particular forms of control states handling (e.g. straight-line programs, circuits) witnesses a programming discipline rather than a physical constraint.

[74]: Gurevich (1995), *Specification and Validation Methods*

The notion of abstract machine is based on the notion of *graphing* introduced by Adams [19] and later studied by Gaboriau [21, 85] in ergodic theory. Note that while the formal definition is almost identical (modulo the addition of control states), our use of graphings differs greatly from their original use. Graphings are of interests to researchers in ergodic theory as generators of a given Borel equivalence relation [86]. Here consider them as specifying dynamical processes. The connection with Borel equivalence relations will nonetheless appear in a later section (chapter 12) and their use in ergodic theory is part of the foundational backbone of our invariant-based approach to complexity. Before defining α-programs, I introduce *graphing representatives*.

[19]: Adams (1990), *Trees and amenable equivalence relations*

[21]: Gaboriau (2000), *Coût des relations d'équivalence et des groupes*

[85]: Gaboriau (2002), *Invariants ℓ^2 de relations d'équivalence et de groupes*

[86]: Gaboriau (2010), *Orbit equivalence and measured group theory*

Definition 4.1.1 *A* graphing representative *G for an* AMC *$\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ is a family of elements of $\mathbb{M}(I)$, i.e. it is defined as*

$$G = \{m_e \mid m_e \in \mathbb{M}(I), e \in E^G\},$$

where E^G is an indexing set. Elements of E^G are called edges*.*

Let $(\Omega, \times)$ be a monoid. An Ω-weighted graphing *H is a family of pairs of an*

element of $\mathbb{M}(I)$ *and an element in* Ω*, i.e.*

$$H = \{(m_e, \omega_e) \mid m_e \in \mathbb{M}(I), \omega_e \in \Omega, e \in E^H\}.$$

If all elements m_e *belong to* I*, we say that* G *is* atomic.

A graphing is then defined as an equivalence of graphing representatives with respect to some notion of equivalence (Definition 4.1.4).

We will here consider weighted graphing representatives over an extension of the AMC that will be used to account for control states. We call those specific graphing representatives *abstract machine representatives* to distinguish them from standard graphing representatives. For the moment, we fix an arbitrary monoid of weights Ω.

Definition 4.1.2 *Let* α *be an* AMC*, with* $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ *and* S^P *a space of* control states*. An* (α, Ω)*-machine representative* P *with control states* S^P *is defined as an* Ω*-weighted* $\tilde{\alpha}$*-graphing representative where* $\tilde{\alpha} = \alpha \times End_*(S_P)$ *is the extension of the action* α *to* $\mathbf{X} \times S_P$ *by the monoid of partial endomorphisms on* S_P*. That is,* P *is defined as a countable collection:*

$$\{(m_e \times \sigma_e, \omega_e) \mid e \in E^P, m_e \in \mathbb{M}(I), \sigma_e \in \mathfrak{G}_{S_P}, \omega_e \in \Omega\}$$

Note that in general the space of control states is a finite discrete space[1]*. If this is the case, one can – at the cost of multiplying the number of edges – write abstract program representatives as tuples* $(m_e, \omega_e, i_e \to f_e)$ *where* $i_e, f_e \in S^P$*.*

1: Although some work considered the case of having the interval $[0, 1]$ as control space [30].

The set of (α, Ω)*-machines representatives is written* $\mathrm{Programs}_\Omega(\alpha;)$*.*

It is important to comment on the set E^P here. The graphing generalises (or rather, concretises) the transition function of, e.g. Turing, machines. Standard notions of programs would therefore require the constraint that E^P should be finite, or at least countable. There is no need for this constraint in theory, and the existence of graphings with infinite indexing set can turn out to be useful in some cases [30].

[30]: Seiller (2016), *Interaction Graphs: Full Linear Logic*

Remark 4.1.1 In some cases, it will be relevant to restrict the set of functions allowed on the set of states. Typically, to represent algebraic computation trees, it is convenient to consider the set of states to be endowed with an order, and to restrict to the action by monotone functions on this set. We however keep these somehow ad-hoc restrictions out of the general theory; indeed algebraic computation trees are just specific types of algebraic computation "graphs". Constraints on the control structures are here considered as part of the choice of a *programming language* (Definition 4.2.2).

In some cases, one may want to identify graphing representatives (and therefore abstract programs). For instance, the following graphing representatives may be considered equivalent:

- the graphing representative with a single edge corresponding to `right`;
- the graphing representative with three edges corresponding to $\texttt{right} \circ \texttt{read}_0$, $\texttt{right} \circ \texttt{read}_1$, and $\texttt{right} \circ \texttt{read}_\star$.

Indeed, these are intuitively corresponding to the same program: in one case the program moves to the right, in the other it moves to the right (1) if reading a 0, (2) if reading a 1, or (3) if reading a blank tape symbol. However, these programs are not equivalent in practice. While the Turing machine model and associated notion of complexity considers reading as a costless operation, memory access is the main source of complexity in real-world computers. One could therefore very well consider those two programs non-equivalent: while both will in the end perform the same operation (i.e. access the same state x' from a given state x), the second one does so while performing non-necessary memory access, implying a larger time complexity.

One may anyway want to introduce some equivalences in specific models. This can be done through the notion of *equational theory*.

Definition 4.1.3 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be an abstract model of computation. An* equational theory *$\mathcal{E}$ for α is a set of pairs of families $(a_i)_{i=0}^n, (b_j)_{j=0}^m$ such that $\sum_{i=0}^n \alpha(a_i) = \sum_{j=0}^m \alpha(b_j)$. We will write this as $(a_i)_{i=0}^n =_{\mathcal{E}} (b_j)_{j=0}^m$.*

The full equational theory *for α is the set of all pairs of families $(a_n), (b_m)$ such that $\sum a_n$ and $\sum b_m$ are equal elements of $\mathbf{X} \rightharpoonup \mathbf{X}$. We denote abusively $\sum a_n =_\alpha \sum b_m$ that the pair $(a_n), (b_m)$ belongs to the full equational theory of α.*

An equational theory $\mathcal{E}$ for α induces a notion of equivalence between (α, Ω)-machines. It is defined as the transitive closure of the family of symmetric relations that follows, which is indexed by pairs $(a_i)_{i=0}^n =_{\mathcal{E}} (b_j)_{j=0}^m$:

$$M \cup \{(a_i \times \sigma_i, \omega) \mid i \in [0, n]\} \sim_E M \cup \{(b_i \times \sigma_i, \omega) \mid i \in [0, m]\}.$$

Another equivalence should be considered on abstract machines, namely that of equality up to renaming the control states. We say that M is equivalent to N up to the renaming of control states if there exists an injective map $\theta : S^M \to S^N$ such that for all $e \in E^M$, there exists $f \in E^N$ such that

$$(m_e \times \sigma_e, \omega_e) = (m_f \times \theta^{-1} \circ \sigma_f \circ \theta, \omega_f).$$

Combining those, we can define the notion of (α, Ω)-machine (w.r.t. an equational theory $\mathcal{E}$).

Definition 4.1.4 *Let α be an* AMC *with $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$, $\mathcal{E}$ an equational theory for α, and S^P a space of* control states*. An abstract $(\alpha, \mathcal{E}, \Omega)$-machine P with control states S^P is an equivalence class of (α, Ω)-machine representatives with control states S^P w.r.t. the equational theory $\mathcal{E}$ and w.r.t. the renaming of set of control states.*

The set of (α, Ω)-machines w.r.t. the equational theory E will be denoted by $\mathrm{Programs}_\Omega(\alpha; \mathcal{E})$.

An (α, Ω)-graphing is an equivalence class of (α, Ω)-machine representatives w.r.t. the full equational theory for α. The set of all (α, Ω)-graphings will be denoted by $\mathrm{Graphings}_\Omega(\alpha)$

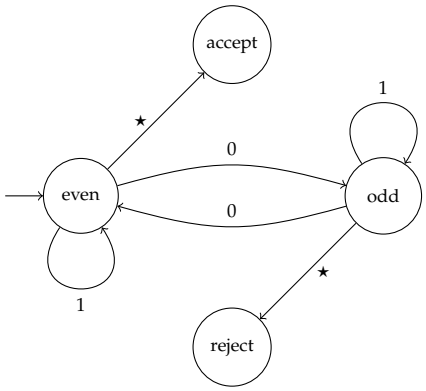


Figure 4.1.: Automata representation of the Turing machine of Example 4.1.1

Example 4.1.1 We now detail an example of Turing machine represented as an abstract machine. The Turing machine considered is an automata (for simplicity), shown in Figure 4.1, and has four states $\{\text{odd}, \text{even}, \text{end}_0, \text{end}_1\}$, as well as the following transition function:

$$\begin{array}{rcl}
(0, \text{even}) & \to & (\text{right}, \text{odd}) \\
(1, \text{even}) & \to & (\text{right}, \text{odd}) \\
(\star, \text{even}) & \to & (\text{right}, \text{end}_0) \\
(0, \text{odd}) & \to & (\text{right}, \text{even}) \\
(1, \text{odd}) & \to & (\text{right}, \text{even}) \\
(\star, \text{odd}) & \to & (\text{right}, \text{end}_1) \\
(_, \text{end}_0) & \to & (\text{right}, \text{end}_0) \\
(_, \text{end}_1) & \to & (\text{right}, \text{end}_1)
\end{array}$$

This machine can be represented in a straightforward manner as the following α_{TM} machine with state set $\{\text{odd}, \text{even}, \text{end}_0, \text{end}_1\}$:

$$\left\{ \begin{array}{l}
(\texttt{right} \circ \texttt{read}_0, \text{even} \to \text{odd}), \\
(\texttt{right} \circ \texttt{read}_0, \text{odd} \to \text{even}), \\
(\texttt{right} \circ \texttt{read}_1, \text{even} \to \text{odd}), \\
(\texttt{right} \circ \texttt{read}_1, \text{odd} \to \text{even}), \\
(\texttt{right} \circ \texttt{read}_\star, \text{even} \to \text{end}_0), \\
(\texttt{right} \circ \texttt{read}_\star, \text{odd} \to \text{end}_1), \\
(\texttt{right}, \text{end}_0 \to \text{end}_0), \\
(\texttt{right}, \text{end}_1 \to \text{end}_1)
\end{array} \right\}$$

This representation combines the discrete nature of the transition graph and the complex and intricate dynamics induced by the action of instructions on the underlying space. In Figure 4.2 we try to illustrate the situation: while the abstract machine can be thought of as a finite object, the target subspace of a given instruction ι depends on the dynamics of $\alpha(\iota)$. We can thus see how complex the induced orbits (which will correspond to *computations*, c.f. section 4.3) can become.

We will be using this example further in the next section to explain and motivate the notion of *program*.

Remark 4.1.2 The consideration of non-trivial equational theories may be problematic once one considers the notion of *cost* which generalises that of complexity. In fact, if one considers that reading the current symbol is

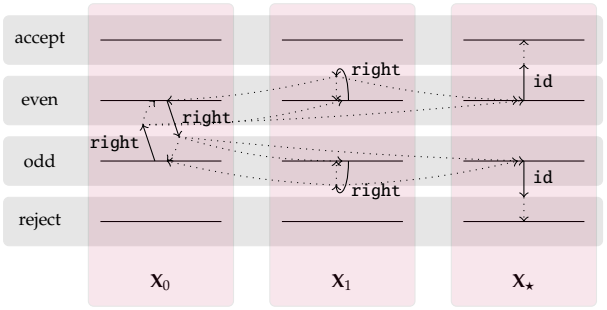


Figure 4.2.: Illustration of an abstract program representing the automata shown in Figure 4.1

costless then an equivalence between having three edges

$$(\texttt{right} \circ \texttt{read}_0, i \to o), (\texttt{right} \circ \texttt{read}_1, i \to o), (\texttt{right} \circ \texttt{read}_\star, i \to o)$$

and a single edge

$$(\texttt{right}, i \to o)$$

would make sense. If one however considers that accessing the value has an associated cost, one would not wish to consider those interchangeable. Once a notion of cost is considered attached to the model (section 5.4 and section 5.5), a natural choice of equivalence (which may be trivial) thus becomes to consider equivalent two sets of edges that (1) perform the same action (i.e. they represent the same partial function), *and* (2) the associated cost is equal.

Now that these definitions are formally written, we will consider that some values of Ω and $\mathcal{E}$ were chosen and simply write Programs(α) for the set of abstract programs. The values of Ω and $\mathcal{E}$ will only be discussed when relevant, i.e. very rarely. In fact, the information provided by Ω could very well be considered as part of the AMC by using the natural action of Ω on itself induced by the monoid structure: more formally one can show a correspondence between (α, Ω)-programs and $(\alpha \times \Omega, \{1\})$-programs ($\{1\}$ being the one-element monoid). But the consideration of elements of Ω as weights of transitions stays closer to the intuitions and will be kept throughout this document.

4.2. Programs

One important aspect of the theory and the practice of computer science is the notion of programming language. We will here provide a completely abstract definition that will be discussed and modulated later on.

In particular, we will distinguish between *machines* and *programs*: while a machine is simply a device with a defined time evolution, the notion of program introduces additional constraints to understand this evolution. Let us take as example the machine that changes states depending on the parity of the number of symbols 0 seen up to a given point (this is the machine in Example 4.1.1). This machine does not compute a function; in fact it does compute many different functions, depending in particular on the control state one is starting from, but also depending on when/how the computation stops. Here are several functions computed by this exact machine:

- The function which accepts if the number of symbols 0 in the initial ⋆-free segment of the tape is odd;
- The function which decides if the number of symbols 0 in the initial ⋆-free segment of the tape is even;
- The function which decides if there is at least one 0 in the initial ⋆-free segment of the tape.

These functions are computed by looking at specific computations made by the machine, that is orbits with constraints on the initial and final control states. A *program* will in fact introduce initial and final control states, to limit the computations considered, and allow for the definition of *the (partial) function computed by the program*.

Definition 4.2.1 *An α-program is defined as a machine $P \in \mathrm{Machines}(\alpha)$ with set of control states S, together with an initial state* $\mathbf{initial}(P) \in S$ *and a terminal state* $\mathbf{terminal}(P)$ *distinct from* $\mathbf{initial}(P)$*, and such that*[2] *for all $e \in E^P$, $i_e \neq \mathbf{terminal}(P)$.*

The set of α-programs is written $\mathrm{Programs}(\alpha)$.

2: An equivalent requirement, which will be sometimes considered, is that there exists a unique edge in P taking its source at $\mathbf{terminal}(P)$, namely $(1, \mathbf{terminal}(P), \mathbf{terminal}(P))$.

We can now define the notion of programming language. It is interesting here to comment first on the computer science practice. How does one define a programming language? This is done in two steps. First, one defines a syntax, usually using inductive grammars (but not exclusively). But a syntax alone is not enough. So the second step is to define the semantics. This can be done in two ways: by giving term rewriting rules (which is more usual in the functional programming community), or by explicit operational semantics: i.e. providing a way to associate to each term a program in a predefined model of computation. Both situations are captured by the following definition.

Definition 4.2.2 *A* programming langage *over the alphabet Σ is a langage $\mathcal{P}$ in the formal sense, i.e. $\mathcal{P} \subset \Sigma^*$, together with an* operational semantics*, i.e. an* AMC *$\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ together with a map $[\![\cdot]\!] : \mathcal{P} \to \mathrm{Programs}(\alpha)$.*

Elements of $\mathcal{P}$ will be called source codes*.*

One important aspect of the theory of programming languages is that the map $[\![\cdot]\!]$ is in general neither surjective (to avoid pathological machines) nor injective (to allow for defining a given program in several different ways).

We note that the distinction between *programs* – i.e. machines described in a programming language – and abstract machines is sometimes blurred. In fact, the set of abstract programs is in itself a programming language. More generally, a universal program (section 5.7) defines a programming language with a surjective mapping into programs.

An important note to make relative to programming languages is that the view above may seem quite limited while the theory of programming languages is rich and complex. In most cases, a programming language will have some structure, i.e. the set P is not simply a language but is defined by induction and/or has an additional algebraic structure. We however believe that further investigations on how the present approach can account for these aspects are outside of the scope of this document.

Restrictions on the control structures or the graphing. Some computational frameworks impose constraints on the control structure of programs or on the graphings considered. For instance, circuits impose the absence of loops: control states can be totally ordered in such a way that transitions are strictly increasing. Another restriction, at the level of graphings, is the constraint of determinism. We view these constraints as part of a choice of a programming language, i.e. restricting to deterministic abstract programs does not correspond to a new notion of machine, but rather to disallowing some machines.

Among these restrictions, we will define and discuss the following restrictions:

- loop-free programs, or *circuits*;
- stateless programs;
- deterministic programs;
- probabilistic programs.

Definition 4.2.3 *A* loop-free *program, or* circuit, *is an abstract program G with set of states S^G such that there exists a partial order $<$ on S^G satisfying:*

- *for all edge (m_e, i_e, o_e) in G, we have $i_e < o_e$;*
- **initial**(G) *is the smallest element of S^G;*
- **terminal**(G) *is the largest element of S^G.*

As a particular case of loop-free programs, we distinguish straightline *programs, for which the order $<$ is total.*

Examples of loop-free programs will be given in chapter 8: algebraic computation trees, or algebraic circuits, are natural occurrences of loop-free programs.

Definition 4.2.4 *A* stateless *program P is an abstract program with $S^P = \{\textbf{initial}(P), \textbf{terminal}(P)\}$.*

Important examples of stateless programs arise from rewriting systems. Indeed, in those examples the programs do not possess control states. As a consequence, the action of a lambda-term t on Λ corresponds to the stateless program $[\![t]\!] = \{(t, \textbf{initial}(t), \textbf{terminal}(t))\}$. This representation however hides that the action of t on Λ is itself defined as a dynamical process at the lower level: computing $[\![t]\!](u)$ requires computing the orbit of $(t)u$ in the underlying dynamical system $\to_\beta \curvearrowright \Lambda$.

Similarly, cellular automata and automata networks give rise to stateless programs. As an example, let us take the cellular automata example. As explained above, one can define the corresponding AMC as $(\Sigma^V \to \Sigma)^* \curvearrowright \Sigma^{\mathbf{X}_{c.a.}}$, where an element of $\Sigma^V \to \Sigma$ represents a set of *rules*. A cellular automata is traditionally given by a single set of rules R, and therefore corresponds to the program $\{(R, \textbf{initial}(R), \textbf{initial}(R))\}$. Note that here the terminal state is never reached. From the abstract model, one could consider alternative models in which different sets of rules are applied at different steps. But since each set of rules defines a total function, this implies that any finite deterministic program will correspond to a finite sequence of rules applied one after the other (and starting back at the first after the last set of rules is applied): but this is equivalent to a single set of rules (usually on a larger alphabet: a sequence of k rules

defined on a neighbourhood of size n corresponds to a single rule on a neighbourhood of size n^k).

Definition 4.2.5 *A* deterministic *program is an abstract program G such that for all*[3] $x \in \mathbf{X}$ *and* $s \in S^G$*, there exists at most one* $e \in E^G$ *with* $(x, s) \in \mathrm{Dom}(\alpha(m_e))$.

3: In the case of measurable spaces, this property may only be satisfied for almost all $x \in \mathbf{X}$.

Proposition 4.2.1 *A deterministic α-graphing defines a partial dynamical systems f on $\mathbf{X} \times [k]$ (k being the cardinal of the set of control states) whose graph is included in the set:*

$$\{(x, y) \mid x \in \mathbf{X} \times [k], \exists (m, \sigma) \in \mathbb{M}(I) \times End_*([k]), \alpha(m) \times \sigma(x) = y\}.$$

Proof. We give the proof for the case of graphing with a singleton set as set of states. The general result follows naturally.

We first explain how to associate a deterministic α-graphing to a partial dynamical system. Consider $G = \{(m_e) \mid e \in E^G\}$ an α-graphing representative. Since it is deterministic, the following relation defines a partial function:

$$f_G : x \mapsto \sum_{e \in E^G} \alpha(m_e)(x)$$

Now, the fact that the graph of f_G is included in

$$\{(x, y) \mid x \in \mathbf{X}, \exists m \in \mathbb{M}(I), \alpha(m) \times \sigma(x) = y\}$$

is a direct consequence of the definition: any $(x, f_G(x))$ is by definition of the form $(x, \alpha(m_e)(x))$ for a chosen $m_e \in \mathbb{M}(I)$. ❀

Remark 4.2.1 The converse is not true, mainly because a partial dynamical system whose graph is included in the required set does not necessarily 'decompose' as a set of edges. In particular, one may consider the subspaces $\mathbf{X}_m = \{x \in \mathbf{X} \mid f(x) = \alpha(m)(x)\}$ which are natural candidates for the sources of such edges, but nothing ensures that the projection onto $\mathbf{X}_m$ belongs to the AMC.

Remark 4.2.2 This property leads to the proof that the graph of a deterministic program P, i.e. the set of pairs (x, y) where y is the output configuration of P when given x as initial configuration, is included in a preorder defined only from α (Equation 4.1).

Definition 4.2.6 *A* subprobabilistic *program is an abstract* $(]0, 1], \times)$*-weighted program G such that for all*[4] $x \in \mathbf{X}$ *and* $s \in S^G$*, we have:*

$$\sum_{e \in E^G, (x,s) \in \mathrm{Dom}(\alpha(m_e))} \omega_e \leq 1.$$

4: In the case of measurable spaces, this property may only be satisfied for almost all $x \in \mathbf{X}$.

The program G is probabilistic *when for all $x \in \mathbf{X}$ and $s \in S^G$ the expression above is an equality.*

We will now show how the probabilistic graphings on measurable spaces define specific sub-Markov kernels called *discrete-image*.

Definition 4.2.7 *Let* $\mathbf{X}$, $\mathbf{Y}$ *be measured spaces. A sub-Markov kernel on* $\mathbf{X} \times \mathbf{Y}$ *is a measurable map* $\kappa : X \times \mathcal{Y} \to [0,1]$ *such that* $\forall x \in X$ *and* $\forall B \in \mathcal{Y}$, $\kappa(x, \_)$ *is a subprobability measure on* X *and* $\kappa(\_, B)$ *is a measurable function.*

If $\kappa(x, \_)$ *is a probability measure,* κ *is a Markov kernel.*

Definition 4.2.8 *A* discrete-image kernel *is a sub-Markov kernel* κ *on* $\mathbf{X} \times \mathbf{Y}$ *such that for all* $x \in \mathbf{X}$, $\kappa(x, \_)$ *is a discrete probability distribution.*

Proposition 4.2.2 *Consider an* AMC α *on a measurable space* $\mathbf{X}$. *A sub-probabilistic* α*-graphing defines a sub-Markov discrete-image kernel* κ *on* $(\mathbf{X} \times [k]) \times (\mathbf{X} \times [k])$ *(*k *being the cardinal of the set of control states) whose graph is included in the set:*

$$\{(x, y) \mid x \in \mathbf{X} \times [k], \exists (m, \sigma) \in \mathbb{M}(I) \times End_*([k]), \alpha(m) \times \sigma(x) = y\}.$$

Proof. Here again, we prove the result for graphings without control states, which is enough to deduce the general statement.

One defines from a subprobabilistic graphing $G = \{m_e, \omega_e \mid e \in E^G\}$ the kernel:

$$\kappa_G : X \times \mathcal{X} \to [0,1]; (x, y) \mapsto \sum_{e \in E^G, \alpha(m_e)(x) = y} \omega_e.$$

The fact that is is a discrete-image sub-Markov kernel is clear.

Now, we define the graph of κ_G as the set of all (x, y) such that $\kappa_G(x, y) > 0$. Then by definition there exists some $e \in E^G$ such that $\alpha(m_e)(x) = y$, and $\omega_e > 0$. This implies the result. ❀

4.3. Computation

Before defining notions of equivalences between programs, we need to establish some notations and define what a *computation* is. The definition is quite easy: since a program is a (generalised) dynamical system, a computation should correspond to an orbit.

Definition 4.3.1 *Let* α *be an* AMC *and* M *be an abstract* α*-machine with set of states* S^M. *An* abstract computation *of* M *on* $(x, s) \in \mathbf{X} \times S^M$ *is defined as an orbit*

$$(x, s) = (x_0, s_0) \to (x_1, s_1) \to \dots$$

such that for all (x_i, s_i) *there exists* $e \in E^M$ *such that* $\alpha(m_e)(x_i, s_i) = (x_{i+1}, s_{i+1})$.

Given a machine M *and an integer* n, *we denote by* $\mathrm{Orbits}^n(M; (x_0, s_0))$ *the set of orbits of length* n *starting at* (x_0, s_0), *i.e.* $\mathrm{Orbits}^n(M; (x_0, s_0))$ *is the set*

$$\{(x_i, s_i)_{i=1}^n \mid \forall i \in [0, n-1], \exists e \in E^M, \alpha(m_e)(x_i, s_i) = (x_{i+1}, y_{i+1})\}.$$

Note that we do not provide explicitly the elements of $\mathbb{M}(I)$ used to define the orbit. This implies that an abstract computation is not bound to an AMC, but is only a sequence in $\mathbf{X} \times S^M$. We now introduce a number of notations.

Definition 4.3.2 *Let α be an* AMC *and M be an abstract α-machine with set of states S^M. For all $x \in \mathbf{X}$ and state s, we define the orbit sphere of size n as:*

$$M^{(n)}(x,s) = \{(y,s') \mid \exists \mathcal{O} \in \mathrm{Orbits}^n(M;(x,s)), (y,s') = \mathcal{O}_n.$$

This leads to the definition of the orbit ball of size n as follows:

$$[M]^n \cdot (x,s) = \bigcup_{i=0}^{n} M^{(i)}(x,s).$$

We will also write $[M]^\omega \cdot (x,s)$ the set $\cup_{i=0}^{\infty} M^{(i)}(x,s)$.

We will now consider the input/output behaviour of programs. Indeed, while a program is represented as a dynamical system, one can reduce it to a relation on the space of configurations as follows.

Definition 4.3.3 *Given an* AMC *α and a program P in* $\mathrm{Programs}(\alpha)$*, we define the relation* $\mathbf{Gr}(P)$ *on* $\mathbf{X} \times \mathbf{X}$ *as:*

$$\{(x,y) \mid x \in \mathbf{X}, (y, \mathbf{terminal}(P)) \in [P]^\omega \cdot (x, \mathbf{initial}(P))\}.$$

Remark 4.3.1 Note that if P is deterministic, the orbit sphere of size n consists in a singleton and the relation $\mathbf{Gr}(P)$ is a functional relation.

We will introduce a last notation for the set of *terminating orbits*, i.e. orbits containing exactly one terminating state.

Definition 4.3.4 *Let α be an* AMC *and P be an abstract α-program with set of states S^P. For all $x \in \mathbf{X}$, we define the set of terminating orbits* $\mathrm{Orbits}_T(P;x)$ *as the set:*

$$\{\mathcal{O} \in \mathrm{Orbits}^\omega(P;(x, \mathbf{initial}(P))) \mid \exists! x' \in \mathbf{X}, (x', \mathbf{terminal}(P)) \in \mathcal{O}\}.$$

Note that this implies, by definition of programs, that the terminal state appears in the last configuration of the orbit.

We say that P is a terminating program when

$$\mathrm{Orbits}_T(P;x) = \mathrm{Orbits}^\omega(P;(x, \mathbf{initial}(P))).$$

We write $\mathrm{Programs}_T(\alpha)$ *the set of terminating programs in* $\mathrm{Programs}(\alpha)$*.*

Remark 4.3.2 While a deterministic program always has an associated (partial) function, it is in general a partial function from $\mathbf{X}$ to $\mathbf{X}$. To represent functions, say, from integers to integers, one has to introduce data types and consider programs restricted to those data types.

We will now notice the following important fact: the choice of the AMC α corresponds to putting constraints on the possible (finite) computations by α-programs. Let us first discuss the deterministic case. A computation is here described as a sequence of instructions applied to a chosen $x \in \mathbf{X}$, ending at a point $y \in \mathbf{X}$. As a consequence, we can deduce that if a given computation is possible then the pair (x,y) belongs to the following set:

$$\mathcal{P}(\alpha) = \{(x,y) \in \mathbf{X} \times \mathbf{X} \mid \exists m \in \mathbb{M}(I), \alpha(m)(x) = y\}. \tag{4.1}$$

Conversely, any pair $(x, y) \in \mathcal{P}(\alpha)$ is *realised* as an orbit. By definition there exists $m = i_1 \dots i_n \in \mathbb{M}(I)$ such that $\alpha(m)(x) = y$. Then one can consider the α-program with $\deg(m) = n$ control states defined as

$$\{(i_k, k \to k+1) \mid k = 1, \dots, n-1\} \cup \{(1, n \to n)\}.$$

The set $\mathcal{P}(\alpha)$ is in general a pre-order. But in the specific case where α is a *measure-preserving group action*, it is what is commonly known as a Borel equivalence relation [87].

[87]: Kechris (2012), *Classical Descriptive Set Theory*

Definition 4.3.5 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be abstract models of computation. We say that α is* weakly reducible *to β when there exists an embedding $\phi : \mathbf{X} \hookrightarrow \mathbf{Y}$ such that $\phi(\mathcal{P}(\alpha)) \subseteq \mathcal{P}(\beta)$, where*

$$\phi(\mathcal{P}(\alpha)) = \{(\phi(x), \phi(y)) \in \mathbf{Y} \times \mathbf{Y} \mid (x, y) \in \mathcal{P}(\alpha)\}.$$

Lemma 4.3.1 *Let α be an* AMC *and P a program in* Programs(α)*. Then we have $\mathbf{Gr}(P) \subset \mathcal{P}(\alpha)$.*

Proof. Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be an AMC and $P \in$ Programs(α). By definition, a pair (a, b) belongs to $\mathbf{Gr}(P)$ if and only if $(b, \mathbf{terminal}(P)) \in [P]^\omega \cdot (a, \mathbf{initial}(P))$, that is if there exists a computation

$$(a, \mathbf{initial}(P)) \xrightarrow{m_1} (x_1, s_1) \to \cdots \to (x_{n-1}, s_{n-1}) \xrightarrow{m_n} (b, \mathbf{terminal}(P)).$$

This implies that $\alpha(\prod_{i=1}^{n} m_i)(x) = y$, hence $(a, b) \in \mathcal{P}(\alpha)$. ❀

Lemma 4.3.2 *Given an* AMC *α, we have*

$$\cup_{P \in \text{Programs}(\alpha)} \mathbf{Gr}(P) = \mathcal{P}(\alpha).$$

Proof. Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be an AMC. From the previous lemma, we have that

$$\cup_{P \in \text{Programs}(\alpha)} \mathbf{Gr}(P) \subseteq \mathcal{P}(\alpha).$$

For the converse inclusion, we consider a pair $(a, b) \in \mathcal{P}(\alpha)$ and prove that $(a, b) \in \mathbf{Gr}(P)$ for some program $P \in$ Programs(α). This is straightforward: by definition the fact that $(a, b) \in \mathcal{P}(\alpha)$ implies the existence of an element $m \in \mathbb{M}(I)$ such that $\alpha(m)(a) = b$. Now, one can consider the program P with exactly two states $\mathbf{initial}(P)$ and $\mathbf{terminal}(P)$ and edges $(m, \mathbf{initial}(P) \to \mathbf{terminal}(P))$ and $(1, \mathbf{terminal}(P) \to \mathbf{terminal}(P))$. By construction, $\mathbf{Gr}(P) = \{(x, y) \mid x \in \mathbf{X}, y = \alpha(m)(x)\}$, hence $(a, b) \in \mathbf{Gr}(P)$. ❀

Theorem 4.3.3 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be simulated by $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$. Then α is weakly reducible to β.*

Proof. Suppose that $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ is simulated by $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$. This means there exists $\phi : \mathbf{X} \to \mathbf{Y}$ and $\theta : I \to \mathbb{M}(J)$ such that:

$$\forall x \in \mathbf{X}, \forall \iota \in I, \phi \circ \alpha(\iota) = \beta(\theta(\iota)) \circ \phi.$$

Consider now $(x, y) \in \mathcal{P}(\alpha)$. This means there exists $\iota \in \mathbb{M}(I)$ such that $y = \alpha(\iota)(x)$. To prove the result, we need to show that the pair

$(\phi(x), \phi(\alpha(\iota))(x))$ belongs to $\mathcal{P}(\beta)$. But using the above property, we can deduce that $\phi(\alpha(\iota))(x)) = \beta(\theta(\iota))(\phi(x))$, i.e. writing $y = \phi(x)$, this pair is equal to $(y, \beta(\mu)(y))$ for a well-chosen $\mu \in \mathbb{M}(J)$, implying that it belongs to $\mathcal{P}(\beta)$. ❀

4.4. Program-level equivalences

Intensional equivalence

We can now define a number of machine-level equivalences. These equivalences state that each machine in a given AMC α is simulated by a machine in an AMC β. However, the simulation is not necessarily as structured as a computational simulation (Definition 3.3.3).

Definition 4.4.1 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be AMCs, and ϕ be an embedding $\mathbf{X} \hookrightarrow \mathbf{Y}$. A program $P \in \mathrm{Programs}(\alpha)$* is intentionally simulated by *a program $Q \in \mathrm{Programs}(\beta)$ w.r.t. ϕ if for all $x \in \mathbf{X}$ and orbit $\mathcal{O} \in \mathrm{Orbits}^{\omega}(P; (x, \mathbf{initial}(P)))$, there exists an orbit $\phi\{\mathcal{O}\} \in \mathrm{Orbits}^{\omega}(Q; (\phi(x), \mathbf{initial}(Q)))$ and an injection $\iota : \mathbf{N} \to \mathbf{N}$ such that:*

$$\forall k \in \mathbf{N}, \left(\phi(x_k) = x'_k\right) \wedge \left(s_k = \mathbf{terminal}(P) \Rightarrow s'_k = \mathbf{terminal}(Q)\right),$$

where we write $\mathcal{O}_k = (x_k, s_k)$ and $\phi\{\mathcal{O}\}_{\iota(k)} = (x'_k, s'_k)$.

We write $P \prec^{\mathrm{int}}_{\mathrm{pw}} Q$ when P is program-wise intentionally simulated by Q.

Definition 4.4.2 *An AMC $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ is* program-wise intentionally *simulated by an AMC $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ when there exists a space embedding $\phi : \mathbf{X} \hookrightarrow \mathbf{Y}$ such that for all $M \in \mathrm{Programs}(\alpha)$ there exists $N \in \mathrm{Programs}(\beta)$ such that $M \prec^{\mathrm{int}}_{\mathrm{pw}} N$ w.r.t. ϕ. This is written as $\alpha \prec^{\mathrm{int}}_{\mathrm{pw}} \beta$.*

Proposition 4.4.1 *If $\alpha \prec^{\mathrm{int}}_{\mathrm{pw}} \beta$, and $\beta \prec^{\mathrm{int}}_{\mathrm{pw}} \gamma$, then $\alpha \prec^{\mathrm{int}}_{\mathrm{pw}} \gamma$.*

Proof. The proof is pretty straightforward. There exists a space embedding $\phi : \mathbf{X} \hookrightarrow \mathbf{Y}$ such that for all $M \in \mathrm{Programs}(\alpha)$ there exists $N \in \mathrm{Programs}(\beta)$ such that $M \prec^{\mathrm{int}}_{\mathrm{pw}} N$ w.r.t. ϕ, and there exists a space embedding $\psi : \mathbf{Y} \hookrightarrow \mathbf{Z}$ such that for all $N \in \mathrm{Programs}(\beta)$ there exists $P \in \mathrm{Programs}(\gamma)$ such that $N \prec^{\mathrm{int}}_{\mathrm{pw}} P$ w.r.t. ψ. It should be clear that the space embedding $\psi \circ \phi : \mathbf{X} \hookrightarrow \mathbf{Z}$ is such that for all $M \in \mathrm{Programs}(\alpha)$ there exists $P \in \mathrm{Programs}(\gamma)$ such that $M \prec^{\mathrm{int}}_{\mathrm{pw}} P$ w.r.t. $\psi \circ \phi$. ❀

Definition 4.4.3 *An AMC $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ is* intentionally equivalent *to an AMC $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ when α program-wise intentionally simulates β and β program-wise intentionally simulates α.*

Lemma 4.4.2 *If α simulates β, then α program-wise intentionally simulates β.*

Proof. Suppose that $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ simulates $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ through the map $\phi : \mathbf{Y} \to \mathbf{X}$. Then for each instruction $\iota \in J$, there exists $\theta(\iota) \in \mathbb{M}(I)$ such that $\phi \circ \alpha(\iota) = \beta(\theta(\iota)) \circ \phi$. We extend θ to a monoid morphism $\mathbb{M}(J) \to \mathbb{M}(I)$ which we abusively denote by θ.

Now, consider a program $M \in \text{Programs}(\beta)$ with set of states S^M defined as

$$(m_e \times \sigma_e, \omega_e) \mid e \in E^M).$$

We define $\tilde{M}$ as the α-program with set of states S^M and edges

$$(\theta(m_e) \times \sigma_e, \omega_e) \mid e \in E^M),$$

letting $\mathbf{initial}(\tilde{M}) = \mathbf{initial}(M)$ and $\mathbf{terminal}(\tilde{M}) = \mathbf{terminal}(M)$.

We will show that for all $k \in \mathbf{N}$ and for all $(y, s) \in \mathbf{Y} \times S^M$, $\phi(M^{(k)} \cdot (x, s)) = \tilde{M}^k \cdot (\phi(x), s)$. But this is a simple induction using the fact that $\phi(\alpha(\iota)(y)) = \beta(\theta(\iota))(\phi(y))$ for all $\iota \in J$ and $y \in \mathbf{Y}$. ❀

A direct corollary of this is the following result.

Proposition 4.4.3 *Computationally equivalent* AMCs *are intentionally equivalent.*

Before going through an example, let us state and prove another implication between different notions of simulations.

Proposition 4.4.4 *If $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ computably compiles $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$, then α program-wise extensionally simulates β.*

Proof. Suppose that $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ computably compiles $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$, and take a program $P \in \text{Programs}(\beta)$ through $\phi : \mathbf{Y} \to \mathbf{X}$. We will explain how to defined from an edge e in P several edges $\{\bar{e}_1, \dots, \bar{e}_k, \dots\}$ in the AMC α that jointly program-wise intentionally simulate e. This construction then easily provides a construction of a program $\bar{P} \in \text{Programs}(\alpha)$ that program-wise intentionally simulates P. Let then $e = (m_e, s_0 \to s_1)$ be an edge. By assumption there exists an α-computable partition $(Y_i)_{i \in \mathbf{N}}$ and elements $(\iota_i)_{i \in \mathbf{N}}$ in $\mathbb{M}(I)$ such that

$$\forall i \in \mathbf{N}, \forall y \in Y_i, \phi \circ \beta(\iota)(y) = \alpha(\iota_i)(\phi(y)). \tag{4.2}$$

This means there exists programs $(P_i)_{i \in \mathbf{N}}$ in $\text{Programs}(\alpha)$ such that for all $i \in \mathbf{N}$, P_i computes the projection onto Y_i. We can assume that all those programs have disjoint sets of control states. We can then build an α-program Q by taking the union of all those programs P_i, identifying states $\mathbf{initial}(Q) = \mathbf{initial}(P_0) = \mathbf{initial}(P_1) = \dots$ and appending additional edges $(\theta(\iota_i), \mathbf{terminal}(P)_i, \mathbf{terminal}(Q))$ for all i. The fact that the program Q program-wise intentionally simulates the edge e should be clear from the assumption: an orbit starting at x and reaching $\mathbf{terminal}(Q)$ needs to go through one of the states $\mathbf{terminal}(P)_i$, and this implies that x belongs to Y_i, and the result then follows from Equation 4.2. ❀

Example 4.4.1 As an example of an AMC which is not computationally simulated by α_{TM} but program-wise simulated, we consider the AMC $\alpha_{k\text{TM}}$ of multi-tape Turing machines. This is a simple adaptation of the AMC α_{TM} of Turing machines over $\{0, 1\}$. We consider the space

$$\mathbf{X}_{k\text{TM}} = \prod_{i=1}^{k} \left(\{0, 1, *\}_*^{\mathbf{Z}} \times \mathbf{Z} \right),$$

and we define the following instruction acting on $\mathbf{X}_{k\text{TM}}$:

- (left) for all $\rho \in \{1, \dots, k\}$,

$$\alpha_{k\text{TM}}(\texttt{left}^{(\rho)}) : ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto ((s_i^j)_{i\in\mathbf{Z}}, \bar{h}^j)_{j\in\{1,\dots,k\}},$$

  where $\bar{h}^j = h^j$ if and only if $j \neq \rho$, and $\bar{h}^\rho = h^\rho - 1$;
- (right) for all $\rho \in \{1, \dots, k\}$,

$$\alpha_{k\text{TM}}(\texttt{right}^{(\rho)}) : ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto ((s_i^j)_{i\in\mathbf{Z}}, \bar{h}^j)_{j\in\{1,\dots,k\}},$$

  where $\bar{h}^j = h^j$ if and only if $j \neq \rho$, and $\bar{h}^\rho = h^\rho + 1$;
- (write) for all $\star \in \{0,1\}^k$ and $\rho \in \{1, \dots, k\}$,

$$\alpha_{k\text{TM}}(\texttt{write}_*) : ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto ((\bar{s}_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}},$$

  where $\bar{s}_i^j = s_i^j$ if and only if $j \neq \rho$ or $i \neq h^\rho$, and $\bar{s}_{h^\rho}^\rho = *$;
- (read) for all $\star \in \{0,1\}^k$ and $\rho \in \{1, \dots, k\}$,

$$\alpha_{k\text{TM}}(\texttt{read}_*) : ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}},$$

  when $s_{h^\rho}^\rho = *$ and undefined otherwise.

Proposition 4.4.5 *The AMC $\tilde{\alpha}_{TM}$ program-wise simulates the AMC α_{kTM}.*

Proof. We define

$$\min = \min(\min_j\{h^j\}, \min_i\{\exists j, s_i^j \in \{0,1\}\})$$
$$\max = \max(\max_j\{h^j\}, \max_i\{\exists j, s_i^j \in \{0,1\}\}).$$

The embedding $\mathbf{X}_{k\text{TM}} \hookrightarrow \tilde{\mathbf{X}}_{\text{TM}}$ is defined by:

$$((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto ((t_i)_{i\in\mathbf{Z}}, h),$$

where

- $h = 2h^1k + 1$ and
- for all $j \in \{1, \dots, k\}$ and $i \in [\min, \max]$:

$$t_{3ik+2k+j-1} = s_i^j$$
$$t_{3ik+2(j-1)}t_{3ik+2j-1)} = 00 \text{ if } h_j > i$$
$$t_{3ik+2(j-1)}t_{3ik+2j-1)} = 01 \text{ if } h_j < i$$
$$t_{3ik+2(j-1)}t_{3ik+2j-1)} = 11 \text{ if } h_j = i$$

- $t_n = \star$ if $n < 3\min$ or $n \geq 3k + \max$.

Note that the single tape is thus split into words of length $3k$ in which the first $2k$ symbols are split into pairs indicating whether the i-th head can be found on the left (01) or the right (00) or at this position (11), and the last k symbols encode the values of the k different tapes at the given position.

The simulation is then defined by simulating each instruction by a small $\tilde{\alpha}_{\text{TM}}$ program and using these programs to interpret individual edges. More precisely, one can define the interpretation of a program $P \in \text{Programs}(\alpha_{k\text{TM}})$ with set of control states S^P as the program $\tilde{P}$ with

set of control states $S^P \times \{i, s_r, s_l, u, b, b_r, b_l\}$ and defined by translating each edge $(\mathtt{inst}, s \to s')$ as a set of edges as follows[5]:

5: We suppose P is atomic to ease the proof, but extending to non atomic instructions does not involve any additional difficulties.

$$(\mathtt{left}^\rho, s \to s') \mapsto \left\{ \begin{array}{l} (\mathtt{right}^{2(\rho-1)}, (s,i) \to (s,s_r)), \\ (\mathtt{right}^{3k} \circ \mathtt{read}_0, (s,s_r) \to (s,s_r)), \\ (\mathtt{left}^{3k} \circ \mathtt{read}_1, (s,s_r) \to (s,s_l)), \\ (\mathtt{read}_\star, \{(s,s_r),(s,s_r)\} \to (s,u)), \\ (\mathtt{left}^{2(\rho-1)} \circ \mathtt{write}_\star \circ \mathtt{left}^{3k} \circ \mathtt{write}_1, (s,u) \to (s,b)), \\ (\mathtt{right}^{3k} \circ \mathtt{read}_0, \{(s,b),(s,b_r)\} \to (s,b_r)), \\ (\mathtt{left}^{3k} \circ \mathtt{read}_1, \{(s,b),(s,b_l)\} \to (s,b_l)), \\ (\mathtt{read}_\star, \{(s,b),(s,b_r),(s,b_l)\} \to (s',i)) \end{array} \right\}$$

$$(\mathtt{right}^\rho, s \to s') \mapsto \left\{ \begin{array}{l} (\mathtt{right}^{2(\rho-1)}, (s,i) \to (s,s_r)), \\ (\mathtt{right}^{3k} \circ \mathtt{read}_0, (s,s_r) \to (s,s_r)), \\ (\mathtt{left}^{3k} \circ \mathtt{read}_1, (s,s_r) \to (s,s_l)), \\ (\mathtt{read}_\star, \{(s,s_r),(s,s_r)\} \to (s,u)), \\ (\mathtt{left}^{2(\rho-1)} \circ \mathtt{write}_\star \circ \mathtt{right}^{3k} \circ \mathtt{write}_0, (s,u) \to (s,b)), \\ (\mathtt{right}^{3k} \circ \mathtt{read}_0, \{(s,b),(s,b_r)\} \to (s,b_r)), \\ (\mathtt{left}^{3k} \circ \mathtt{read}_1, \{(s,b),(s,b_l)\} \to (s,b_l)), \\ (\mathtt{read}_\star, \{(s,b),(s,b_r),(s,b_l)\} \to (s',i)) \end{array} \right\}$$

$$(\mathtt{read}_*^\rho, s \to s') \mapsto \left\{ \begin{array}{l} (\mathtt{right}^{2(\rho-1)}, (s,i) \to (s,s_r)), \\ (\mathtt{right}^{3k} \circ \mathtt{read}_0, (s,s_r) \to (s,s_r)), \\ (\mathtt{left}^{3k} \circ \mathtt{read}_1, (s,s_r) \to (s,s_l)), \\ (\mathtt{read}_\star, \{(s,s_r),(s,s_r)\} \to (s,u)), \\ (\mathtt{left}^{2k+\rho-1} \circ \mathtt{read}_* \circ \mathtt{right}^{2k-\rho}, (s,u) \to (s,b)), \\ (\mathtt{right}^{3k} \circ \mathtt{read}_0, \{(s,b),(s,b_r)\} \to (s,b_r)), \\ (\mathtt{left}^{3k} \circ \mathtt{read}_1, \{(s,b),(s,b_l)\} \to (s,b_l)), \\ (\mathtt{read}_\star, \{(s,b),(s,b_r),(s,b_l)\} \to (s',i)) \end{array} \right\}$$

$$(\mathtt{write}_*^\rho, s \to s') \mapsto \left\{ \begin{array}{l} (\mathtt{right}^{2(\rho-1)}, (s,i) \to (s,s_r)), \\ (\mathtt{right}^{3k} \circ \mathtt{read}_0, (s,s_r) \to (s,s_r)), \\ (\mathtt{left}^{3k} \circ \mathtt{read}_1, (s,s_r) \to (s,s_l)), \\ (\mathtt{read}_\star, \{(s,s_r),(s,s_r)\} \to (s,u)), \\ (\mathtt{left}^{2k+\rho-1} \circ \mathtt{write}_* \circ \mathtt{right}^{2k-\rho}, (s,u) \to (s,b)), \\ (\mathtt{right}^{3k} \circ \mathtt{read}_0, \{(s,b),(s,b_r)\} \to (s,b_r)), \\ (\mathtt{left}^{3k} \circ \mathtt{read}_1, \{(s,b),(s,b_l)\} \to (s,b_l)), \\ (\mathtt{read}_\star, \{(s,b),(s,b_r),(s,b_l)\} \to (s',i)) \end{array} \right\}$$

It is then easy to verify that $\tilde{P}$ simulates P. Suppose given $x \in \mathbf{X}_{\mathrm{TM}(2^k)}$ and an edge $(\mathtt{inst}, s \to s')$ in P. The computation $(x,s) \overset{\mathtt{inst}}{\to} (x',s')$ is simulated by the computation:

$$(\phi(x), s, i) \to \cdots \to (\phi(x'), s, i)$$

whose length depends on $h^\rho - h^1$ (where ρ denotes the tape on which the instruction $\mathtt{inst}$ acts). ❀

Extensional equivalence

In the previous section we introduce program-wise intentional equivalence, i.e. in some way an *algorithmically complete* notion of simulation: if a program exists in the AMC α, then some program in the AMC β simulates it step by step. We now weaken this notion to only consider *extensional* simulation: we only focus on the computed function (on $\mathbf{X}$) computed by α-programs, dismissing *how* those are computed.

Definition 4.4.4 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be AMCs, and ϕ be an embedding $\mathbf{X} \hookrightarrow \mathbf{Y}$. A program $M \in \mathrm{Programs}(\alpha)$* is extensionally simulated by *a program $N \in \mathrm{Programs}(\beta)$ w.r.t. ϕ if $\phi(\mathbf{Gr}(M)) \subseteq \mathbf{Gr}(N)$.*

We denote note this by $M \prec^{\mathrm{ext}}_{\mathrm{pw}} N$.

Definition 4.4.5 *An AMC $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ is* program-wise extensionally *simulated by an AMC $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ when there exists a space embedding $\mathbf{X} \hookrightarrow \mathbf{Y}$ such that all α-programs $M \in \mathrm{Programs}(\alpha)$ are extensionally simulated by a β-program. This is denoted by $\alpha \prec^{\mathrm{ext}}_{\mathrm{pw}} \beta$.*

The following result is straightforward and can be proved in a similar way as Proposition 4.4.1

Proposition 4.4.6 *If $\alpha \prec^{\mathrm{ext}}_{\mathrm{pw}} \beta$ and $\beta \prec^{\mathrm{ext}}_{\mathrm{pw}} \gamma$, then $\alpha \prec^{\mathrm{ext}}_{\mathrm{pw}} \gamma$.*

Definition 4.4.6 *An AMC $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ is* extensionally equivalent *to an AMC $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ when α program-wise extensionally simulates β and β program-wise extensionally simulates α.*

Lemma 4.4.7 *If α program-wise intentionally simulates β, then α program-wise extensionally simulates β.*

Proof. This result is a corollary of the fact that if a program $P \in \mathrm{Programs}(\alpha)$ intentionally simulates a program $Q \in \mathrm{Programs}(\beta)$, then P extensionally simulates β. This is what we will prove now. Suppose P intentionally simulates Q. Then there exists an embedding $\phi : \mathbf{Y} \hookrightarrow \mathbf{X}$ such that each computation of P is mapped to a computation of Q. Suppose now that $(x, x') \in \mathbf{Gr}(P)$; by definition this means that $(x', \mathbf{terminal}(P)) \in [P]^{\omega} \cdot ((x, \mathbf{initial}(P)))$, i.e. there exists $n \in \mathbf{N}$ and $\mathcal{O} \in \mathrm{Orbits}^{\omega}(P; (x, \mathbf{initial}(P)))$ such that $\mathcal{O}_k = (x', \mathbf{terminal}(P))$. But since P is intentionally simulated by Q, there exists an orbit $\phi\{\mathcal{O}\} \in \mathrm{Orbits}^{\omega}(Q; (x, \mathbf{initial}(P)))$ and an injection $\iota : \mathbf{N} \to \mathbf{N}$ such that $\phi\{\mathcal{O}\}_{\iota(n)} = (\phi(x'), \mathbf{terminal}(Q))$. This implies that $(\phi(x), \phi(x')) \in \mathbf{Gr}(Q)$. ❀

Proposition 4.4.8 *If α is program-wise extensionally simulated by β, then α is weakly reducible to β.*

Proof. Suppose that α is program-wise extensionally simulated by β. I.e. there exists an embedding $\phi : \mathbf{X} \hookrightarrow \mathbf{Y}$ such that for all all $M \in \mathrm{Programs}(\alpha)$, there exists $\tilde{M} \in \mathrm{Programs}(\beta)$ with $\phi(\mathbf{Gr}(M)) \subseteq \mathbf{Gr}(N)$. But by Lemma 4.3.2, we have that

$$\cup_{M \in \mathrm{Programs}(\alpha)} \mathbf{Gr}(M) = \mathcal{P}(\alpha).$$

Now, given $(x, y) \in \mathcal{P}(\alpha)$, this implies that there exists $M \in \text{Programs}(\alpha)$ such that $(x, y) \in \mathbf{Gr}(M)$. Now, this means that $(\phi(x), \phi(y)) \in \mathbf{Gr}(\tilde{M})$. But by Lemma 4.3.1, $\mathbf{Gr}(\tilde{M}) \subset \mathcal{P}(\beta)$, hence $(\phi(x), \phi(y)) \in \mathcal{P}(\beta)$. Since this is true for any $(x, y) \in \mathcal{P}(\alpha)$, we have proved that $\phi(\mathcal{P}(\alpha)) \subset \mathcal{P}(\beta)$, i.e. that α is weakly reducible to β. ❀

Abstract data structures and complexity

5.

5.1. Abstract data structures

One important aspect of computation, essential in the criticism of the Church-Turing thesis[1] is that most of the time the models are compared over some data structure. We continue studying in this section additional notions of equivalences between models of computation. The weakest of those, namely *extensional* `nat`*-equivalence*, expresses the Church-Turing thesis. The question of data representation will also be essential in the definition of (specified) algorithms.

1: I here understand the original Church-Turing thesis, as most probably understood originally. The question of if, and rather when, the Church-Turing thesis started being understood for other types (e.g. higher functionals) is unclear to me at this point. At least Kleene makes the distinction in a paper [88].

One can already find a notion of "abstract data *type*" in the literature. However, this notion is not operational enough for our purposes, and we will rather work with an abstraction of data *structures*: i.e. a set of *values*, together with a set of allowed operations on those values. Underlying this notion, one can identify a notion of *data representation*. We will therefore define simultaneously two notions: that of data domain and that of data structure, which subsumes a choice of data domain.

Definition 5.1.1 *An* abstract data domain $\mathcal{D}$ *is a set* D*. An* abstract data structure $\mathtt{D}$ *over an abstract data domain* $\mathcal{D}$ *is a set of allowed operations* $S_{\mathtt{D}}$ *– called* structural maps *– in* $\sum_{k,k'} \mathcal{D}^k \rightharpoonup \mathcal{D}^{k'}$*.*

Any element $s \in S_{\mathtt{D}}$ *belongs to* $\mathcal{D}^k \rightharpoonup \mathcal{D}^{k'}$ *for fixed values of* k *and* k'*: we will write* $\mathrm{dom}(s) = k$ *and* $\mathrm{im}(s) = k'$*.*

The maximal arity *of the abstract data structure* $\mathtt{D}$ *is defined as (by definition, we consider that* $\max S = \infty$ *when the set* S *is unbounded):*

$$\max\{\mathrm{dom}(s), \mathrm{im}(s) \mid s \in S_{\mathtt{D}}\}.$$

We note that once again, we do not impose restrictions on the cardinality of abstract data domains or on the cardinality of the set of allowed maps. A particularly interesting case is when the latter is defined by induction (Example 5.1.1). Similarly, in order to define a proper, usable, abstract data structure one would require some maps relating the representations of $\mathcal{D}^k$ for various values of k. But on principle, we will not impose such restrictions.

This notion is defined in a way which is quite similar to the definition of abstract model of computation: we use partial maps to allow for reading. For instance, a notion of integers with a test to zero could be represented by the set $\mathbf{N}$ together with maps from $\mathbf{N}$ to $\{0, 1\}$ but this would imply the introduction of another data type (booleans), or a particular (ad-hoc) choice of encoding of booleans as specific elements of the set $\mathbf{N}$. Partial functions avoid those by allowing to project on a subset; this can then be used to implement any of these other approaches. But this choice allows for self-contained definitions of data structures.

We will now give examples. We note that agreeing on a specific axiomatisation for a given data structure is a complex task, and in fact depends on the expected use. As a consequence, the examples below are proposed

axiomatisations but could very well be replaced by others. We are not trying to establish those as *standard* definitions, but rather illustrate how one could manipulate the notion.

Definition 5.1.2 (Abstract boolean) *One can define an abstract data structure representing booleans as follows: the underlying data domain is* $\mathcal{B} = \{0, 1\}$, *with the structural maps* $\texttt{read}_0$, $\texttt{read}_1$, `and`, `and`, *and* `not` *defined as:*

$$\begin{aligned}
\texttt{read}_0 &: n \mapsto n \text{ if and only if } n = 0\\
\texttt{read}_1 &: n \mapsto n \text{ if and only if } n = 1\\
\texttt{and} &: (m, n) \mapsto m \times n\\
\texttt{or} &: (m, n) \mapsto m + n - m \times n\\
\texttt{not} &: n \mapsto 1 - n.
\end{aligned}$$

The following definition captures the standard unary representation of natural numbers, which was used in the early work on computability.

Definition 5.1.3 (Natural numbers) *One can define an abstract data structure representing natural numbers as follows: the underlying abstract data domain is* $\mathcal{N} = \mathbf{N}$, *with the structural maps* $\texttt{read}_0$, $\texttt{read}_S$, `succ` *defined by:*

$$\begin{aligned}
\texttt{read}_0 &: n \mapsto n \text{ if and only if } n = 0\\
\texttt{read}_S &: n \mapsto n \text{ if and only if } n \neq 0\\
\texttt{succ} &: n \mapsto n + 1.
\end{aligned}$$

This data structure can be extended by other operations, such as predecessor, addition and multiplication defined as:

$$\begin{aligned}
\texttt{pred} &: n \mapsto n - 1 \text{ if and only if } n \neq 0\\
\texttt{add} &: (n, m) \mapsto n + m\\
\texttt{mult} &: (n, m) \mapsto n \times m.
\end{aligned}$$

The standard approach is however to represent integers in base 2. The ability of representing integers in binary is particularly important when considering computational complexity: due to the logarithmic factor between an integer and the size of its binary representation (or its representation in any base ≥ 2), the notion of *polynomially bounded running time* do not correspond. This is due to the fact that operations do not have the same cost: multiplying an integer n by 2 in unary can only be performed in time $O(n)$ while the same operation is performed in time $O(\log n)$ in binary (or any other base ≥ 2).

Definition 5.1.4 (Binary lists, simply linked) *One can define an abstract data domain representing binary lists as the set* $\mathcal{L}_2 = \{0, 1\}^{|\mathbf{N}|}$. *We will write* ϵ *the empty list.*

This domain can be endowed with several data structures. One can initially consider the set of structural maps $\mathrm{R} = \{\texttt{read}_0, \texttt{read}_1, \texttt{pop}\}$ *where:*

- $\texttt{read}_0 : \mathcal{L}_2 \to \mathcal{L}_2, \alpha \mapsto \alpha$ *only if* $\alpha_0 = 0$;
- $\texttt{read}_1 : \mathcal{L}_2 \to \mathcal{L}_2, \alpha \mapsto \alpha$ *only if* $\alpha_0 = 1$;
- $\texttt{pop} : \mathcal{L}_2 \to \mathcal{L}_2, \alpha \mapsto \alpha'$ *only if* $\alpha = a \cdot \alpha'$;

The data structure thus defined is that of simply linked read-only binary lists: one can read each bit and go through the list from left to right, but no structural map allows for accessing the previous bits.

One can modify the above set of structural maps by adding L = {`right`, `left`} to get a data structure representing read-only[2] doubly chained binary lists:

2: In fact, a proper read-only variant would disallow the `pop` instruction; here the list can be modified by removing elements, but adding new elements or modifying values is impossible.

Definition 5.1.5 (Binary lists, doubly linked) *We define the datadomain as* $\mathcal{L}_2 \times \mathcal{L}_2$. *The instructions are then:*

- $\texttt{read}_0 : \mathcal{L}_2 \times \mathcal{L}_2 \to \mathcal{L}_2 \times \mathcal{L}_2, (\alpha, b \cdot \beta) \mapsto (\alpha, b \cdot \beta)$ *only if* $b = 0$;
- $\texttt{read}_1 : \mathcal{L}_2 \times \mathcal{L}_2 \to \mathcal{L}_2 \times \mathcal{L}_2, (\alpha, b \cdot \beta) \mapsto (\alpha, b \cdot \beta)$ *only if* $b = 1$;
- $\texttt{left} : \mathcal{L}_2 \times \mathcal{L}_2 \to \mathcal{L}_2 \times \mathcal{L}_2, (\alpha, a \cdot \beta) \mapsto (a \cdot \alpha', \beta)$;
- $\texttt{right} : \mathcal{L}_2 \times \mathcal{L}_2 \to \mathcal{L}_2 \times \mathcal{L}_2, (a \cdot \alpha, \beta) \mapsto (\alpha', a \cdot \beta)$.
- $\texttt{pop} : \mathcal{L}_2 \times \mathcal{L}_2 \to \mathcal{L}_2 \times \mathcal{L}_2, (\alpha, b \cdot \beta) \mapsto (\alpha, \beta')$;

One can also extend the set of structural maps with $\mathrm{W} = \{\texttt{write}_0, \texttt{write}_1\}$ *to allow for modifying the values of the list:*

- $\texttt{write}_0 : \mathcal{L}_2 \times \mathcal{L}_2 \to \mathcal{L}_2 \times \mathcal{L}_2, (\alpha, b \cdot \beta \mapsto (\alpha, 0 \cdot \beta)$;
- $\texttt{write}_1 : \mathcal{L}_2 \times \mathcal{L}_2 \to \mathcal{L}_2 \times \mathcal{L}_2, (\alpha, b \cdot \beta) \mapsto (\alpha, 1 \cdot \beta)$.

These different data structures arise naturally in different models of computation: read-only simply chained lists quite naturally capture the operations a standard automata can perform, while read-only doubly-linked lists would be natural to consider for two-way automata as well as in the geometry of interaction interpretation of the usual representation of binary lists in lambda-calculus – corresponding to the logical formula (or a translation in linear logic such as the one considered in subsection 11.2):

$$\forall X, (X \Rightarrow X) \Rightarrow (X \Rightarrow X) \Rightarrow (X \Rightarrow X).$$

We leave it to the reader to convince themselves that other abstract structures quite standard in computability and complexity, such as trees or graphs, give rise to abstract data structures in a natural way. We however detail one essential example, that of recursive functions.

The notion of recursive function is sometimes considered as a model of computation. My understanding is that it is not, because the definition of recursive functions is abstracted from computation; it does not determine computation, it only specifies *what* could be computed. As such, I considered for a while that it could be an example of a programming language. But once again it is not a programming language because there are no standard operational semantics associated to recursive functions; one may implement the computation of a recursive function in very different ways. This leads to thinking of the notion as something of a more algorithmic nature, separated from consideration on implementations. It finally occurred to me that it was an example of an abstract data structure.

Example 5.1.1 (Recursive functions) Primitive and general recursive functions are in our sense definitions of data structures. Indeed, one can take as data domain $D = \mathbf{N}$ together with a set of structural maps Rec. One can then define Rec inductively as follows:

- Rec contains constant functions $c_i : \mathbf{N} \to \mathbf{N}$ for $i \in \mathbf{N}$;
- Rec contains successor function $S : \mathbf{N} \to \mathbf{N}$;

- Rec contains projections $\Pi^i_k : \mathbf{N}^k \to \mathbf{N}$;
- Rec is closed under composition operators: for all $i \in \mathbf{N}$ and $(f, g) \in \text{Rec}$ such that $\text{dom}(f) = k \leqslant i$ and $\text{dom}(g) = k'$, there exists $g \circ_i f \in \text{Rec}$ such that:

$$g \circ_i f(x_1, \ldots, x_{i-1}, y_1, \ldots, y_{k'}, x_{i+1}, \ldots, x_k) \\ = f(x_1, \ldots, x_{i-1}, g(y_1, \ldots, y_{k'}), x_{i+1}, \ldots, x_{k'});$$

- Rec is closed under primitive recursion: for all $(f, g) \in \text{Rec}$ such that $\text{dom}(f) = \text{dom}(g) + 2$, there exists $\rho(f, g) \in \text{Rec}$ such that $\rho(f, g)(z, x_1, \ldots, x_{\text{dom}(f)})$ is equal to

$$\begin{cases} f(x_1, \ldots, x_{\text{dom}(f)}) & \text{if } z = 0 \\ g(z', \rho(f, g)(z', x_1, \ldots, x_{\text{dom}(f)}), x_1, \ldots, x_{\text{dom}(f)}) & \text{if } z = S(z') \end{cases}$$

- (Only for general recursive) Rec is closed under minimisation: for all $f \in \text{Rec}$ such that $\text{dom}(f) > 1$, there exists $\mu(f) \in \text{Rec}$ such that

$$\mu(f)(x_2, \ldots, x_{\text{dom}(f)}) = z \\ \Leftrightarrow \\ \Big(f(z, x_2, \ldots, x_{\text{dom}(f)}) = 0\Big) \wedge \Big(\forall x_1 < z, f(x_1, \ldots, x_{\text{dom}(f)}) \neq 0\Big).$$

Remark 5.1.1 There are several consequences of this. First, it means that Turing-completeness can be defined as the possibility to implement this data-structure. Second, it provides a new point of view on algebraic characterisation of complexity classes from Implicit complexity (ICC), such as the Bellantoni and Cook result [36]. Those can now be understood as defining abstract data structures that capture time- or space-bounded computations, in the same way linear logic based techniques in ICC captures those same classes through types.

[36]: Bellantoni et al. (1992), *A new recursion-theoretic characterization of the polytime functions*

Obviously, this notion of data structure is independent of a chosen machine model. To compute on data structures, one needs to implement them. To be precise, one would have to account for the complexity of the implementation, which could lead to overhead in time or space when simulating programs from a model of computation into another. This will be discussed later in this section.

Definition 5.1.6 *Suppose given a model of computation $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$, and an abstract data domain $\mathcal{D}$ with maximal arity* $\mathbf{a}$*. An interpretation of a data domain $\mathcal{D}$ is a map*

$$\Delta : \cup_{k \leq \mathbf{a}} D^k \to \mathbf{X}.$$

Notations 5.1.1. We establish some notations that will be useful in the remaining parts of the chapter. For each sequence $s \in \{0, 1, \star\}^n$, we write $\underline{\star} s \underline{\star}$ the configuration $(t_i)_{i \in \mathbf{Z}}$ defined as $t_i = s_i$ for $i = 0, 1, \ldots, n-1$. Given two sequences s_1, s_2, we also write $s_1 \star s_2$ the concatenation of s_1 and s_2. Lastly, for each natural number n we write $\overline{n}^2$ the sequence in $\{0, 1\}$ corresponding to its binary representation and 1^n the sequence consisting of n occurrences of the symbol 1.

We also write $\text{shift}^k(x)$ the sequence x shifted of k positions to the left: $(\text{shift}^k(x))_i = x_{i+k}$.

Example 5.1.2 (Interpretations of Booleans) We consider the Boolean data structure defined above (Definition 5.1.2). Note that it is has maximal arity 2. We can define the interpretation as follows:

$$\begin{aligned} \Delta_{\mathcal{B}} : \qquad a \in \mathcal{B} &\mapsto \underline{\star} a \underline{\star} \\ (a,b) \in \mathcal{B}^2 &\mapsto \underline{\star} a \star b \underline{\star} \end{aligned}$$

Note that the data structure as defined there does not allow for composing functions. One could add a rule for defining new structural maps by allowing for composition, i.e. defining the set of structural maps inductively by saying that if $f : \mathcal{D}^n \to \mathcal{D}$ and $g : \mathcal{D}^m \to \mathcal{D}$ are structural maps, then $g \circ_i f : \mathcal{D}^{m+n-1} \to \mathcal{D}$ defined as

$$\begin{aligned} &g \circ_i f(x_1, \ldots, x_{i-1}, y_1, \ldots, y_m, x_{i+1}, \ldots, x_n) \\ &= g(x_1, \ldots, x_{i-1}, f(y_1, \ldots, y_m), x_i, \ldots, x_n), \end{aligned}$$

is also a structural map. An interpretation of the underlying domain in the Turing machines AMC α_{TM} for this extended data structure is given as the following:

$$\Delta_{\mathcal{B}} : (a_0, a_1, \ldots, a_n) \underline{\star} a_0 \star a_1 \star \cdots \star a_n \underline{\star}$$

An interpretation of the abstract domain of booleans in the lambda-calculus AMC $\alpha_\lambda(S)$ for this extended data structure is given by

$$\begin{aligned} \Delta_{\mathcal{B}} : 0 &\mapsto \lambda x.\lambda y.y, \\ \Delta_{\mathcal{B}} : 1 &\mapsto \lambda x.\lambda y.x, \\ (a_0, a_1, \ldots, a_n) &\mapsto \lambda f.(\ldots(f)\Delta_{\mathcal{B}}(a_0))\Delta_{\mathcal{B}}(a_1))\ldots\Delta_{\mathcal{B}}(a_n). \end{aligned}$$

Example 5.1.3 (Interpretations of natural numbers) One can define two interpretations of the abstract domain of natural numbers in the Turing machines AMC α_{TM}. given as the following:

$$\begin{aligned} \Delta_{\mathcal{N}} &: (a_0, a_1, \ldots, a_n) \mapsto \underline{\star} 1^{a_0} \star 1^{a_1} \star \cdots \star 1^{a_n} \underline{\star} \\ \Delta^{(2)}_{\mathcal{N}} &: (a_0, a_1, \ldots, a_n) \mapsto \underline{\star} \overline{a_0}^2 \star \overline{a_1}^2 \star \cdots \star \overline{a_n}^2 \underline{\star} \end{aligned}$$

An interpretation of the abstract domain of booleans in the lambda-calculus AMC $\alpha_\lambda(S)$ is given by

$$\begin{aligned} \Delta_{\mathcal{B}} : 0 &\mapsto \lambda x.\lambda y.y, \\ \Delta_{\mathcal{B}} : 1 &\mapsto \lambda x.\lambda y.x, \\ (a_0, a_1, \ldots, a_n) &\mapsto \lambda f.(\ldots(f)\Delta_{\mathcal{B}}(a_0))\Delta_{\mathcal{B}}(a_1))\ldots)\Delta_{\mathcal{B}}(a_n). \end{aligned}$$

Example 5.1.4 (Interpretations of binary lists) An interpretation of the abstract domain of binary lists in the Turing machines AMC α_{TM} is given as the following:

$$\begin{aligned} s = a_0, \ldots, a_n &\mapsto \underline{\star} a_0 a_1 \ldots a_n \underline{\star} \\ (s_1, s_2) = (a_0, \ldots, a_n, b_0, \ldots, b_m) &\mapsto \text{shift}^{n+1}(\underline{\star} a_0 a_1 \ldots a_n b_0 b_1 \ldots b_m \underline{\star}) \end{aligned}$$

Definition 5.1.7 *Suppose given a model of computation* $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$*, and an abstract data structure* $\mathtt{D} = (\mathcal{D}, S_D)$*. An implementation of* $\mathtt{D}$ *in* α *is an interpretation* Δ *of the underlying data domain* $\mathcal{D}$ *together with a map* $\delta : S_D \to$ Programs(α) *such that for all* $d \in D^k$*,* $\exists n, \delta(f)^n(\Delta(d)) = \Delta(f(d))$*.*

Example 5.1.5 (Implementations of Booleans) An implementation of the boolean data structure in the Turing machines AMC α_{TM} is given by:

$$\text{and} \mapsto \left\{ \begin{array}{c} (\mathtt{write}_1 \cdot \mathtt{read}_0, \mathbf{initial}(P) \to \mathbf{terminal}(P)), \\ (\mathtt{write}_0 \cdot \mathtt{read}_1, \mathbf{initial}(P) \to \mathbf{terminal}(P)) \end{array} \right\},$$

$$\text{and} \mapsto \left\{ \begin{array}{c} (\mathtt{write}_\star \cdot \mathtt{read}_0, \mathbf{initial}(P) \to e), \\ (\mathtt{right}^2 \cdot \mathtt{write}_\star \cdot \mathtt{read}_1, \mathbf{initial}(P) \to \mathbf{terminal}(P) \\ (\mathtt{write}_0 \cdot \mathtt{left}^2, e \to \mathbf{terminal}(P))) \end{array} \right\},$$

$$\text{or} \mapsto \left\{ \begin{array}{c} (\mathtt{write}_\star \cdot \mathtt{read}_1, \mathbf{initial}(P) \to e), \\ (\mathtt{right}^2 \cdot \mathtt{write}_\star \cdot \mathtt{read}_0, \mathbf{initial}(P) \to \mathbf{terminal}(P) \\ (\mathtt{write}_1 \cdot \mathtt{left}^2, e \to \mathbf{terminal}(P))) \end{array} \right\}.$$

An implementation of booleans in lambda-calculus (Example 3.2.13) is given by the following. We use the notation $\text{fst}(a) = (a)\lambda x.\lambda y.x$ and $\text{scd}(a) = (a)\lambda x.\lambda y.y$.

$$\begin{aligned} \text{and} &\mapsto \lambda a.((\text{fst}a)\text{scd}(a))\lambda x.\lambda y.y, \\ \text{or} &\mapsto \lambda a.(\text{fst}a)\text{scd}(a), \\ \text{not} &\mapsto \lambda a.\lambda x.\lambda y.((a)y)x. \end{aligned}$$

It is a standard exercise to check that these terms interpret the structural maps considered in Definition 5.1.2..

We leave it as an exercise to define implementations of integers in the AMC α_{TM} of Turing machines (w.r.t. the two interpretations described in), as well as in the AMC of lambda-calculus. This allows us to skip those and directly detail implementations of binary lists.

Example 5.1.6 (Implementations of simply linked binary lists) We detail the implementation of simply linked binary lists in the AMC α_{FA} of finite automata () w.r.t. the interpretation of the data domain shown in Definition 5.1.4.

$$\begin{aligned} \mathtt{read}_0 &\mapsto \left\{ \ (\mathtt{read}_0, \mathbf{initial}(P) \to \mathbf{terminal}(P)) \ \right\}, \\ \mathtt{read}_1 &\mapsto \left\{ \ (\mathtt{read}_1, \mathbf{initial}(P) \to \mathbf{terminal}(P)) \ \right\}, \\ \mathtt{pop} &\mapsto \left\{ \ (\mathtt{right}, \mathbf{initial}(P) \to \mathbf{terminal}(P)) \ \right\}, \end{aligned}$$

Example 5.1.7 (Implementations of doubly linked binary lists) We detail the implementation of binary lists in the AMC α_{TM} w.r.t. the interpretation

of the data domain shown in Example 5.1.4.

$$
\begin{aligned}
\texttt{read}_0 &\mapsto \left\{ \ (\texttt{read}_0, \textbf{initial}(P) \to \textbf{terminal}(P)) \ \right\},\\
\texttt{read}_1 &\mapsto \left\{ \ (\texttt{read}_1, \textbf{initial}(P) \to \textbf{terminal}(P)) \ \right\},\\
\texttt{left} &\mapsto \left\{ \ (\texttt{left}, \textbf{initial}(P) \to \textbf{terminal}(P)) \ \right\},\\
\texttt{right} &\mapsto \left\{ \ (\texttt{right}, \textbf{initial}(P) \to \textbf{terminal}(P)) \ \right\},\\
\texttt{write}_0 &\mapsto \left\{ \ (\texttt{write}_0, \textbf{initial}(P) \to \textbf{terminal}(P)) \ \right\},\\
\texttt{write}_1 &\mapsto \left\{ \ (\texttt{write}_1, \textbf{initial}(P) \to \textbf{terminal}(P)) \ \right\},\\
\texttt{pop} &\mapsto \left\{ \begin{array}{c}
(\texttt{right} \cdot \texttt{write}_\star, \textbf{initial}(P) \to r),\\
(\texttt{right} \cdot \texttt{read}_0, r \to r),\\
(\texttt{right} \cdot \texttt{read}_1, r \to r),\\
(\texttt{left} \cdot \texttt{read}_{star}, r \to l),\\
(\texttt{left} \cdot \texttt{write}_\star \cdot \texttt{read}_1, l \to l1),\\
(\texttt{left} \cdot \texttt{write}_\star \cdot \texttt{read}_0, l \to l0),\\
(\texttt{left} \cdot \texttt{write}_1 \cdot \texttt{read}_1, l1 \to l1),\\
(\texttt{left} \cdot \texttt{write}_1 \cdot \texttt{read}_0, l1 \to l0),\\
(\texttt{left} \cdot \texttt{write}_0 \cdot \texttt{read}_1, l0 \to l1),\\
(\texttt{left} \cdot \texttt{write}_0 \cdot \texttt{read}_0, l0 \to l0),\\
(\texttt{write}_0 \cdot \texttt{read}_\star, l0 \to \textbf{terminal}(P)),\\
(\texttt{write}_1 \cdot \texttt{read}_\star, l1 \to \textbf{terminal}(P))
\end{array} \right\}.
\end{aligned}
$$

5.2. Data-constrained Equivalences

Definition 5.2.1 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be an abstract model of computation, and $\mathcal{D}_1, \mathcal{D}_2$ be abstract data domain respectively interpreted by Δ_1 and Δ_2 in α. A program $P \in \text{Programs}(\alpha)$ is a $\Delta_1 \to \Delta_2$-program when for all $d_1 \in \mathcal{D}_1$,*

$$(d, \textbf{terminal}(P)) \in [P]^\omega \cdot ((\Delta_1(d_1), \textbf{initial}(P))) \Rightarrow d \in \Delta_2(\mathtt{D}_2).$$

We write $\text{Programs}^{\Delta_1 \to \Delta_2}(\alpha)$ *the set of all $\Delta_1 \to \Delta_2$-programs in* $\text{Programs}(\alpha)$, *and* $\text{Programs}_T^{\Delta_1 \to \Delta_2}(\alpha)$ *the set of all terminating programs in* $\text{Programs}^{\Delta_1 \to \Delta_2}(\alpha)$.

Definition 5.2.2 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be abstract models of computation, $\mathcal{D}$ and $\mathcal{D}'$ be abstract data domains, and Δ and Δ' (resp. Θ and Θ') be interpretations of $\mathcal{D}$ and $\mathcal{D}'$ in α (resp. β). We say that α program-wise $\mathcal{D} \to \mathcal{D}'$-intentionally simulates β (w.r.t. $\Delta, \Delta', \Theta, \Theta'$) when there exists an embedding $\phi : \mathbf{Y} \hookrightarrow \mathbf{X}$ such that for all $Q \in \text{Programs}^{\Theta \to \Theta'}(\beta)$ there exists $P \in \text{Programs}^{\Delta \to \Delta'}(\alpha)$ that extensionally simulates Q.*

Proposition 5.2.1 *Suppose an AMC α program-wise intentionally simulates an AMC β. For all interpretations Θ, Θ' of the data domains $\mathcal{D}, \mathcal{D}'$ in $\mathbf{Y}$, there exists interpretations Δ, Δ' of $\mathcal{D}, \mathcal{D}'$ in $\mathbf{X}$ such that α program-wise $\mathcal{D} \to \mathcal{D}'$-intentionally simulates β (w.r.t. $\Delta, \Delta', \Theta, \Theta'$).*

Proof. The key element here is that if α program-wise intentionally simulates β through a map $\phi : \mathbf{Y} \to \mathbf{X}$, then interpretations Θ, Θ' of the data domains $\mathcal{D}$ and $\mathcal{D}'$ yield interpretations of $\mathcal{D}$ and $\mathcal{D}'$ in $\mathbf{X}$ through $\Delta = \phi \circ \Theta$ and $\Delta' = \phi \circ \Theta$. Showing that α program-wise $\mathcal{D} \to \mathcal{D}'$-intentionally simulates β is then straightforward since all $Q \in \text{Programs}(\beta)$ is intentionally simulated by a $P \in \text{Programs}(\alpha)$, and

this implies that if $(\Theta'(d'), \mathbf{terminal}(Q)) \in [Q]^\omega \cdot ((\Theta(d), \mathbf{initial}(Q)))$, then $(\phi \circ \Theta'(d'), \mathbf{terminal}(P)) \in [P]^\omega \cdot ((\phi \circ \Theta(d), \mathbf{initial}(P)))$. That is, if $Q \in \text{Programs}^{\Theta \to \Theta'}(\beta)$ then $P \in \text{Programs}^{\Delta \to \Delta'}(\alpha)$. ❀

We can obviously define an equivalent extensional notion and prove similar results.

Definition 5.2.3 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ be abstract models of computation, $\mathcal{D}$ and $\mathcal{D}'$ be abstract data domains, and Δ and Δ' (resp. Θ and Θ') be interpretations of $\mathcal{D}$ and $\mathcal{D}'$ in α (resp. β). We say that α program-wise $\mathcal{D} \to \mathcal{D}'$-intentionally simulates β (w.r.t. $\Delta, \Delta', \Theta, \Theta'$) when there exists an embedding $\phi : \mathbf{Y} \hookrightarrow \mathbf{X}$ such that for all $Q \in \text{Programs}^{\Theta \to \Theta'}(\beta)$ there exists $P \in \text{Programs}^{\Delta \to \Delta'}(\alpha)$ that extensionally simulates Q.*

Proposition 5.2.2 *Suppose an AMC α program-wise extensionally simulates an AMC β. For all interpretations Θ, Θ' of the data domains $\mathcal{D}, \mathcal{D}'$ in $\mathbf{Y}$, there exists interpretations Δ, Δ' of $\mathcal{D}, \mathcal{D}'$ in $\mathbf{X}$ such that α program-wise $\mathcal{D} \to \mathcal{D}'$-extensionally simulates β (w.r.t. $\Delta, \Delta', \Theta, \Theta'$).*

Proof. The proof is similar to that of Proposition 5.2.1. Writing $\phi : \mathbf{Y} \hookrightarrow \mathbf{X}$ the embedding supporting the program-wise extensional simulation of β by α, we can define $\Delta = \phi \circ \Theta$ and $\Delta' = \phi \circ \Theta'$. We then have that if $Q \in \text{Programs}(\beta)$ is intentionally simulated by $P \in \text{Programs}(\alpha)$ and $Q \in \text{Programs}^{\Theta \to \Theta'}(\beta)$, then $P \in \text{Programs}^{\Delta \to \Delta'}(\alpha)$. ❀

Proposition 5.2.3 *For fixed data domains and interpretations, program-wise $\mathcal{D} \to \mathcal{D}'$-intentional simulation implies program-wise $\mathcal{D} \to \mathcal{D}'$-extensional simulation.*

Proof. Suppose $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ program-wise $\mathcal{D} \to \mathcal{D}'$-intentionally simulates $\beta : \mathbb{M}(J) \curvearrowright \mathbf{Y}$ w.r.t. $\Delta, \Delta', \Theta, \Theta'$ and $\phi : \mathbf{Y} \to \mathbf{X}$. Then for all program $Q \in \text{Programs}^{\Theta \to \Theta'}(\beta)$, there exists $P \in \text{Programs}^{\Delta \to \Delta'}(\alpha)$ that intentionally simulates it. But this implies that P extensionally simulates Q by Definition 4.4.5. ❀

5.3. A bit of history

Recall that the Church-Turing thesis states that the set of *effectively computable functions*[3] (which is an informal notion) is equal to the set of functions computable by Turing machines (or equivalently, are computable in lambda-calculus). A forceful argument for the Church-Turing thesis is that the following sets of functions (on natural numbers) are equal:

- functions computed by Turing machines;
- general recursive functions;
- lambda-definable functions.

3: Here I believe the original statements considered only functions from natural numbers to natural numbers, although some extensions were considered later on [88].

It is interesting to dive into the research work of the time to understand which results are proven exactly. It turns out that the above equivalence is, from a purely formal point of view, not properly established. We are obviously not putting the result into question, but if we reformulate the exact statements of theorem proven in these papers, we obtain the following:

- In his 1937 paper (Computability and λ-Definability), Turing shows[4] that if f is a lambda-definable function (on Church numerals), then there exists a Turing Machine whose output is the sequence

$$\underbrace{11\ldots1}_{f(0)\text{ symbols}}\ 0\ \underbrace{11\ldots1}_{f(1)\text{ symbols}}\ 0\ldots0\ \underbrace{11\ldots1}_{f(n)\text{ symbols}}\ 0\ldots$$

- In the same paper, Turing also shows that if there exists a Turing Machine whose output is the sequence above, then f is general recursive. This is done by means of an encoding of configurations as integers.
- In his 1936 paper (λ-definability and recursiveness), Kleene shows that every general recursive function is λ-definable using a *shifted version of Church numerals*, in which 0 is represented as 1, 2 represented as 3, etc.

The representation of computable functions by Turing shows that the equivalence considered is based on *unary* representations of integers. Moreover, the choice in the definition of what a computable function is leaves open the question of the representation of partial functions. Indeed, the result applies only to total functions[5]. Lastly, one should notice that Kleene uses a different interpretation of (unary) natural numbers.

We will not detail here how the following results can be proved. However, these results can be restated as[6]:

- Lambda-calculus interprets the data structure of partial recursive functions (and can interpret it for both a unary and binary representation of the underlying data domain);
- Turing machines interpret the data structure of partial recursive functions (and can interpret it for both a unary and binary representation of the underlying data domain);
- One can encode the elements of $\mathbf{X}_{\text{TM}}$ as natural numbers and show that, based on this encoding, each instruction is interpreted as a partial recursive function;
- One can encode the elements of Λ as natural numbers and show that, based on this encoding, that a step of β-reduction is interpreted as a partial recursive function.

Based on the encoding of configurations, one can obtain a simulation of lambda-calculus by Turing machines (and conversely). This simulation is a data-constrained program-wise simulation, which means that the above results could be used as an algorithmic Church-Turing thesis, namely that[7] *the set of effective methods to compute functions is equal to the set of Turing machines (computing functions)*.

We note however that this alternative statement does not account for complexity. Indeed, the quantitative aspects of the simulations above

4: Although he writes "No attempt is being made to give a formal proof that this machine has the properties claimed for it. Such a formal proof is not possible unless the ideas of substitution and so forth occurring in the definition of conversion are formally defined, and the best form of such a definition is possibly in terms of machines".

5: This shows that, even though Turing knows that Turing machines compute partial functions, his notion of *computable function* presupposes totality.

6: Although binary representations are not considered in these papers, it is easily obtained in a similar manner

7: Here we restrict the discussion to functions from natural numbers to natural numbers for simplicity.

imply that time and space constrained (especially for 'small' classes) effective methods do not coincide.

5.4. Configuration complexity

We will now discuss how to formalise space and time complexity, although the notion of space complexity will in fact not necessarily be related to space but rather abstractly captures any notion of size of configurations. This will be simply represented as a map from the space of configuration to the natural numbers[8]. For this reason, we chose the terminology *configuration complexity* that seems more adequate as it does not refer to a notion of 'space'.

8: This restriction is somehow ad-hoc, and could be replaced with any totally ordered semi-ring, but we will not consider any examples in which another codomain is needed.

Definition 5.4.1 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be an* AMC. *A* configuration cost model *is a map $c : \mathbf{X} \to \mathbf{N}$.*

Suppose given an abstract machine M, a pair $(x, s) \in \mathbf{X} \times S^M$, and an orbit $O = (x_i, s_i)_{i=1}^n \in \mathrm{Orbits}^n(M; (x, s))$. One defines the configuration complexity *of the orbit O as the quantity $c(O) = \max_{(x,s)\in O} c(x)$.*

Example 5.4.1 One can define the following size measures on the model of Turing machines. The first one simply counts the number of symbols equal to 0 and 1, while the second measures the length between the leftmost and the rightmost symbols different from $\star$.

$$s_0 : (s_i)_{i\in\mathbf{Z}} \mapsto \mathrm{Card}(\{i \mid s_i \neq \star\}),$$
$$s_1 : (s_i)_{i\in\mathbf{Z}} \mapsto \min\{\mathrm{len}(w) \mid w \in \{0, 1, \star\}^*, (s_i) = \star^\omega w \star^\omega\}.$$

Note that $s_0 \leqslant s_1$, but there is no constant C such that $s_1 \leqslant C \times s_0$. In fact, the two size measures are not equivalent, and while s_1 corresponds to the usual notion of space complexity, s_0 allows for computing non-regular langages in constant space.

We are now able to define configuration complexity classes in a natural way. For this, we introduce the following notation, where c is a configuration cost function on an AMC α, P is a program in $\mathrm{Programs}(\alpha)$, and f is a function $\mathbf{N} \to \mathbf{N}$:

$$\mathrm{BdC}(x, P, c, f) \Leftrightarrow \forall \mathcal{O} \in \mathrm{Orbits}^\omega(P; x, \mathbf{initial}(P)), c(\mathcal{O}) = O(f(c(x))).$$

Definition 5.4.2 *Let α be an* AMC, *c a configuration cost function, and let $f : \mathbf{N} \to \mathbf{N}$ be any function. We define the following sets:*

$$\mathrm{Conf}_{\alpha,c}(f) = \{P \in \mathrm{Programs}(\alpha) \mid \forall x \in \mathbf{X}, \mathrm{BdC}(x, P, c, f)\}$$

We also define the following set for all abstract domains $\mathcal{D}$ and $\mathcal{D}'$, respectively interpreted by Δ and Δ' in α:

$$\mathrm{Conf}_{\alpha,c}^{\Delta\to\Delta'}(f) = \{P \in \mathrm{Programs}^{\Delta\to\Delta'}(\alpha) \mid \forall d \in \mathcal{D}, \mathrm{BdC}(\Delta(d), P, c, f)\}.$$

This leads to the definition of the corresponding data configuration cost class:

$$\textsc{Conf}_{\alpha,c}^{\Delta\to\Delta'}(f) = \{(\Delta \times \Delta')^{-1}(\mathbf{Gr}(P)) \mid P \in \mathrm{Conf}_{\alpha,c}^{\Delta\to\Delta'}(f)\}.$$

The above definition induces

Remark 5.4.1 Given a configuration cost function c and an interpretation Δ of an abstract data domain $\mathcal{D}$, one can define a measure $c^{\mathcal{D}}$ on $\mathcal{D}$ by defining $c^{\mathcal{D}}(d) = c(\Delta(d))$. If one considers the measure s_1 defined above, the induced measure on binary lists corresponds to the usual measure considered on integers (represented as binary lists) when studying complexity in the setting of Turing machines.

Example 5.4.2 To show this with a simple example, we consider a variant of the AMC α_{TM} in which the input (resp. output) is given on a separate read-only (resp. write only) tape:

$$\mathbf{X}_{\text{TM}}^{\text{sublin}} = \mathbf{X}_{\text{TM}} \times \mathbf{X}_{\text{TM}} \times \mathbf{X}_{\text{TM}}.$$

We write elements of this space as triples (s_i, s_w, s_o) for the input, work, and output tapes respectively. The instructions are given as:

- $\texttt{right}^i(s_i, s_w, s_o) = (\alpha_{\text{TM}}(\texttt{right})(s_i), s_w, s_o)$;
- $\texttt{right}^w(s_i, s_w, s_o) = (s_i, \alpha_{\text{TM}}(\texttt{right})(s_w), s_o)$;
- $\texttt{right}^o(s_i, s_w, s_o) = (s_i, s_w, \alpha_{\text{TM}}(\texttt{right})(s_o))$;
- $\texttt{left}^i(s_i, s_w, s_o) = (\alpha_{\text{TM}}(\texttt{left})(s_i), s_w, s_o)$;
- $\texttt{left}^w(s_i, s_w, s_o) = (s_i, \alpha_{\text{TM}}(\texttt{left})(s_w), s_o)$;
- $\texttt{left}^o(s_i, s_w, s_o) = (s_i, s_w, \alpha_{\text{TM}}(\texttt{left})(s_o))$;
- $\texttt{read}_*^i(s_i, s_w, s_o) = (\alpha_{\text{TM}}(\texttt{read}_*)(s_i), s_w, s_o)$;
- $\texttt{read}_*^w(s_i, s_w, s_o) = (s_i, \alpha_{\text{TM}}(\texttt{read}_*)(s_w), s_o)$;
- $\texttt{write}_*^w(s_i, s_w, s_o) = (s_i, \alpha_{\text{TM}}(\texttt{write}_*)(s_w), s_o)$;
- $\texttt{write}_*^o(s_i, s_w, s_o) = (s_i, s_w, \alpha_{\text{TM}}(\texttt{write}_*)(s_o))$.

We then define the configuration cost models $\bar{s}_0$ and $\bar{s}_1$ defined as:

$$\bar{s}_0(s_i, s_w, s_o) = s_0(s_w) \bar{s}_1(s_i, s_w, s_o) = s_1(s_w).$$

We leave it to the reader to convince themselves that the AMC $\alpha_{\text{TM}}^{\text{sublin}}$ together with the configuration cost model $\bar{s}_1$ captures the usual notion of space complexity for Turing machines with one working tape. In particular, the notion of Logspace computation is captured by those machines whose orbits have $\bar{s}_1$ measure at most logarithmic in the size of the input (a more formal definition is given in the next section).

But now, one can consider the following program that decides whether the input is a palindrome. We will consider that the program moves the pointers for the input and working tape simultaneously (hence "moving one step to the right" means applying both $\texttt{right}^i$ and $\texttt{right}^w$). The program goes as follows:

1. The program looks at the symbol b on the working tape: if it is 0 it accepts (it writes 1 on the output state and reaches the terminal state), otherwise it reads the symbol a on the input tape, remembers its value (as part of the control state), and write 1 on the working tape (at the position of a);
2. The program moves to the right until it reads a 0 on the working tape or a $\star$ on the input tape, if it read a 0 on the working tape, it writes a $\star$ to replace it and moves back one step to the left, if it read a $\star$ on the input tape it simply goes back one step to the left

3. It reads the symbol on the working tape: if it is a 1 the program accepts, otherwise it reads the symbol on the input tape; if this symbol is different from a, the program rejects (i.e. writes 0 on the output tape and reaches the terminal state), otherwise it writes a 0 on the working tape;
4. The program moves left until it reads a 1 on the working tape, writes a $*$ in its place and moves one step to the right, and goes back to step 1.

One can verify that the $\bar{s}_0$ function on configurations the program goes through during computation is always at most 2: there is at most one symbol 1 and one symbol 0 which denote the position of the last bits of the input checked. Hence the program works in constant configuration cost! (On the other hand, the configuration cost for $\bar{s}_1$ of the orbits is equal, at some point, to the length of the input.) But this is incoherent with what standard complexity tells us: since the palindrome language is non-regular, it cannot be computed by a machine working in $o(\log \log n)$ [89].

[89]: Lewis et al. (1965), *Memory bounds for recognition of context-free and context-sensitive languages*

5.5. Transition complexity

The notion of time complexity is also abstracted in a way that will allow for other interpretations, such as energy consumption. As a consequence, I chose to use the terminology *transition complexity*. The natural idea is that the orbit length corresponds to time complexity. However, one may want to consider non-trivial measures, for which an instruction may have a non-unital time complexity, or a complexity that depends on the configuration.

The natural definition is then to fix a cost for each instruction in I and then deduce the cost of arbitrary sequences. The definition is somehow straightforward: if m, n are two instructions, then the cost $t(mn, x)$ of performing mn on configuration x – i.e. performing the instruction n on x and then applying m to the result – should be equal to the sum of the cost of performing n on x, i.e. $t(n, x)$, and the cost of performing m on $n \cdot x$, i.e. $t(m, n \cdot x)$. This is formalised as a so-called *logarithmic chain rule* whose name is explained below.

Definition 5.5.1 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be an AMC. A* transition cost model *for α is a function $t : I \times \mathbf{X} \to \mathbf{N}$. This is extended to $\mathbb{M}(I)$ by a logarithmic chain rule:*

$$t(mn, x) = t(m, n \cdot x) + t(n, x).$$

Let now M be an abstract machine and x a configuration. The transition complexity *of the length k orbit*

$$\mathcal{O} = (x_0, s_0) \overset{m_1 \times \sigma_1}{\to} (x_1, s_1) \to \dots \overset{m_k \times \sigma_k}{\to} (x_k, s_k) \in \mathrm{Orbits}^k(M; (x_0, s_0))$$

is defined as $t(\mathcal{O}) = \sum_{i=1}^{k} t(m_{i-1}, x_i)$.

We notice that cost models do act like logarithms of differential operators. Indeed, writing $D_x = \exp(t(\_, x))$ for a cost model t and $f = \alpha(n)$,

$g = \alpha(m)$, the logarithmic chain rule we have written becomes:

$$D_x(g \circ f) = D_{f(x)}(g) \times D_x(f).$$

Whether this remark can be used further remains to be seen. Let us simply notice that some complexity measures defined to prove lower bounds on algebraic models of computation are based on partial derivatives [90, 91].

[90]: Baur et al. (1983), *The complexity of partial derivatives*
[91]: Chen et al. (2011), *Partial Derivatives in Arithmetic Complexity and Beyond*

As for the case of configuration cost functions, we can define classes of programs and functions whose transition cost is bounded. For this definition to make sense, we will however require both a transition cost function and a configuration cost function. Given c and t configuration and transition costs functions on an AMC α, P a program in Programs(α), and f any function $\mathbf{N} \to \mathbf{N}$, we define:

$$\mathrm{BdT}(x, P, c, t, f) \Leftrightarrow \forall \mathcal{O} \in \mathrm{Orbits}_T(P; \omega)(x, \mathbf{initial}(P)), t(\mathcal{O}) = O(f(c(x))).$$

Definition 5.5.2 *Let α be an AMC, c a configuration cost function, t a transition cost function, and let $f : \mathbf{N} \to \mathbf{N}$ be any function. We define the following sets:*

$$\mathrm{Trans}_{\alpha,c,t}(f) = \{P \in \mathrm{Programs}_T(\alpha) \mid \forall x \in \mathbf{X}, \mathrm{BdT}(x, P, c, t, f)\}$$

We also define the following set for all abstract domains $\mathcal{D}$ and $\mathcal{D}'$, respectively interpreted by Δ and Δ' in α:

$$\mathrm{Trans}^{\Delta \to \Delta'}_{\alpha,c,t}(f) = \{P \in \mathrm{Programs}^{\Delta \to \Delta'}_T(\alpha) \mid \forall d \in \mathcal{D}, \mathrm{BdT}(\Delta(d), P, c, t, f)\}.$$

This leads to the definition of the corresponding data configuration cost function class:

$$\textsc{Trans}^{\Delta \to \Delta'}_{\alpha,c,t}(f) = \{(\Delta \times \Delta')^{-1}(\mathbf{Gr}(P)) \mid P \in \mathrm{Conf}^{\Delta \to \Delta'}_{\alpha,c,t}(f)\}.$$

In the Turing machine models, the standard transition cost model t_1 – corresponding to the usual time complexity measure – is defined by

$$\begin{aligned} t_1(\mathtt{read}_0) = t_1(\mathtt{read}_1) = t_1(\mathtt{read}_\star) &= 0 \\ t_1(\mathtt{write}_0) = t_1(\mathtt{write}_1) = t_1(\mathtt{write}_\star) &= 1 \\ t_1(\mathtt{left}) = t_1(\mathtt{right}) &= 1. \end{aligned}$$

Together with the standard configuration cost s_1 (discussed in the previous section), this allows to define standard complexity classes using the interpretations of integers and booleans (Example 2 and Example 5.1.2), e.g.

$$\begin{aligned} \textsc{Ptime} &= \cup_k \textsc{Trans}^{\Delta^{(2)}_{\mathcal{N}} \to \Delta_{\mathcal{B}}}_{\alpha_{\mathrm{TM}}, s_1, t_1}(x \mapsto x^k) \\ \textsc{FPtime} &= \cup_k \textsc{Trans}^{\Delta^{(2)}_{\mathcal{N}} \to \Delta^{(2)}_{\mathcal{N}}}_{\alpha_{\mathrm{TM}}, s_1, t_1}(x \mapsto x^k) \end{aligned}$$

Remark 5.5.1 The two notions of configuration and transition cost models are in fact instances of a single notion of *cost model*, where a cost model is a function c from $I \times \mathbf{X}$ to some monoid $(\Omega, +)$ and satisfying the logarithmic chain rule (w.r.t. the monoid operation). Taking $(\Omega, +)$ yields the notion of transition cost model introduced above, while configuration

costs models are recovered by taking $(\Omega, \max)$ and considering that the function c does not depend on I.

5.6. Quantitative equivalences

A quantitative model of computation (QMC) is an abstract AMC together with both a configuration c^α and a transition cost model t^α. We can define the quantitative overheads of a simulation of a QMC β by a QMC α as follows.

Definition 5.6.1 *Suppose α and β are QMCs such that β simulates α as AMCs with respect to the space embedding $\phi : \mathbf{X} \hookrightarrow \mathbf{Y}$ and the map $\theta : I \mapsto \mathbb{M}(J)$, and let g be a function $\mathbf{N} \to \mathbf{N}$. We say that β simulates α in g-bounded configuration cost expansion when*

$$c^\beta(\phi(x)) \leqslant g(c^\alpha(x)).$$

Similarly, we say that β simulates α in g-bounded transition cost expansion when

$$t^\beta(\theta(\iota), \phi(x)) \leqslant g(t^\alpha(\iota, x)).$$

We will now prove some results explaining how complexity classes defined in different QMCs can be related when the simulation has bounded configuration ratio and/or transition ratio. It however appears that to be able to relate the notions in this way, it is required to have a common notion of cost for the data. For this, we consider two QMC α and β, where α is simulated by β, together with abstract data domains $\mathcal{D}$ and $\mathcal{D}'$ and respective interpretations $\Delta, \Delta', \Theta, \Theta'$. To relate the classes $\textsc{Conf}^{\Delta\to\Theta}_{\alpha,c^\alpha}(f)$ and $\textsc{Conf}^{\Delta'\to\Theta'}_{\beta,c^\beta}(f)$, we will moreover assume that:

$$\forall d \in \mathcal{D}, c^\alpha(\Delta(d)) = c^\beta(\Delta'(d)).$$

Proposition 5.6.1 *Suppose we are in the situation just described and that $g : \mathbf{N} \to \mathbf{N}$ is an increasing function, i.e. satisfying $g(\max S) = \max g(S)$. Then for all function $f : \mathbf{N} \to \mathbf{N}$, we have:*

$$\textsc{Conf}^{\Delta\to\Theta}_{\alpha,c^\alpha}(f) \subseteq \textsc{Conf}^{\Delta'\to\Theta'}_{\beta,c^\beta}(g \circ f).$$

Proof. Consider $P \in \mathrm{Conf}_{\alpha,c^\alpha}(f)$ and the program $\tilde{P}$ simulating P. This program is defined by replacing each instruction $\iota \in I$ by $\theta(\iota) \in J$. I.e. the image of the orbit $\mathcal{O} \in \mathrm{Orbits}^\omega(P; (x, \mathbf{initial}(P)))$ with

$$\mathcal{O} = (x_0, s_0) \xrightarrow{\iota_0} (x_1, s_1) \xrightarrow{\iota_1} \ldots \xrightarrow{\iota_{n-1}} (x_n, s_n) \xrightarrow{\iota_n} \ldots,$$

through this simulation is $\mathcal{O}'$ which consists in the sequence

$$(\phi(x_0), s_0) \xrightarrow{\theta(\iota_0)} (\phi(x_1), s_1) \xrightarrow{\theta(\iota_1)} \ldots \xrightarrow{\theta(\iota_{n-1})} (\phi(x_n), s_n) \xrightarrow{\theta(\iota_n)} \ldots.$$

We now compute the configuration cost of $\mathcal{O}'$:

$$\begin{aligned} c^\beta(\mathcal{O}') &= \max_i c^\beta(\phi(x_i)) \\ &\leqslant \max_i g(c^\alpha(x_i)) \\ &\leqslant g(\max_i c^\alpha(x_i)) \\ &\leqslant g(c^\alpha(\mathcal{O})). \end{aligned}$$

Now, since P satisfied that $c^\alpha(\mathcal{O}) = O(f(c^\alpha(x)))$, this gives $c^\beta(\mathcal{O}') = O(g(f(c^\alpha(x))))$. If one restricts to configurations $x = \Delta(d)$ for a $d \in \mathcal{D}$, then using the fact that $c^\alpha(\Delta(d)) = c^\beta(\Delta'(d))$, we deduce that:

$$c^\beta(\mathcal{O}') = O(g(f(c^\beta(\Delta'(d))))),$$

which means that $\tilde{P}$ belongs to $\mathrm{Conf}_{\beta,c^\beta}(g \circ f)$. ❀

Remark 5.6.1 The previous proof works if one considers that intermediate configurations are not considered when computing the configuration cost of an orbit in which some transitions are non-atomic. If one were to consider only atomic graphings (a more satisfactory way of dealing with the complexity of a simulation), then additional considerations should be taken into account. In particular, one could use the fact that an atomic instruction can increase the configuration cost of at most 1 (this is the case for Turing machines for instance), hence intermediate configurations appearing when applying $\theta(i)$ on configuration x have their configuration cost bounded by $x + \deg(\theta(i))$ (where $\deg(\cdot)$ denotes the algebraic degree of the simulation as defined in Definition 3.3.3).

Proposition 5.6.2 *Suppose we are in the situation of the previous proposition and that $g : \mathbf{N} \to \mathbf{N}$ is a convex function, i.e. satisfying $\sum_i g(s_i) \leqslant g(\sum_i s_i)$. Then for all function $f : \mathbf{N} \to \mathbf{N}$, we have:*

$$\textsc{Trans}^{\Delta\to\Theta}_{\alpha,c^\alpha,t^\alpha}(f) \subseteq \textsc{Conf}^{\Delta'\to\Theta'}_{\beta,c^\beta,t^\beta}(g \circ f).$$

Proof. Consider $P \in \mathrm{Conf}_{\alpha,c^\alpha}(f)$ and the program $\tilde{P}$ simulating P. As before, this program is defined by replacing each instruction $\iota \in I$ by $\theta(\iota) \in J$. I.e. the image of the orbit $\mathcal{O} \in \mathrm{Orbits}^\omega(P; (x, \mathbf{initial}(P)))$ with

$$\mathcal{O} = (x_0, s_0) \xrightarrow{\iota_0} (x_1, s_1) \xrightarrow{\iota_1} \dots \xrightarrow{\iota_{n-1}} (x_n, s_n) \xrightarrow{\iota_n} \dots,$$

through this simulation is $\mathcal{O}'$ which consists in the sequence

$$(\phi(x_0), s_0) \xrightarrow{\theta(\iota_0)} (\phi(x_1), s_1) \xrightarrow{\theta(\iota_1)} \dots \xrightarrow{\theta(\iota_{n-1})} (\phi(x_n), s_n) \xrightarrow{\theta(\iota_n)} \dots.$$

We now compute the transition cost of $\mathcal{O}'$:

$$\begin{aligned} t^\beta(\mathcal{O}') &= \sum_i t^\beta(\theta(\iota_i), \phi(x_i)) \\ &\leqslant \sum_i g(t^\alpha(\iota_i, x_i)) \\ &\leqslant g(\sum_i t^\alpha(\iota_i, x_i)) \\ &\leqslant g(t^\alpha(\mathcal{O})). \end{aligned}$$

Now, since P satisfied that $t^\alpha(\mathcal{O}) = O(f(c^\alpha(x)))$, this gives $t^\beta(\mathcal{O}') = O(g(f(c^\alpha(x))))$. If one restricts to configurations $x = \Delta(d)$ for a $d \in \mathcal{D}$, then using the fact that $c^\alpha(\Delta(d)) = c^\beta(\Delta'(d))$, we deduce that:

$$t^\beta(\mathcal{O}') = O(g(f(c^\beta(\Delta'(d))))),$$

which means that $\tilde{P}$ belongs to $\mathrm{Trans}_{\beta,c^\beta,t^\beta}(g \circ f)$. ❀

We can also consider quantitative variants of the notions of program-wise simulations (Definition 4.4.2 and Definition 4.4.5). Here again, while the usual notion is that of *overhead*, it is in fact more natural to work with *cost expansions*. I.e. we will here note that $t_k(\mathcal{O}') \leqslant f(t_k(\mathcal{O}))$ for $f(n) = 2n^2 + 6kn$. Note that the overhead is simply computed as $f(n) - n$. The following definition introduces this notion.

Definition 5.6.2 *Let $f, g : \mathbf{N} \to \mathbf{N}$ be functions. Suppose a program $P \in \mathrm{Programs}(\alpha)$ is intentionally simulated by a program $Q \in \mathrm{Programs}(\beta)$ through $\phi : \mathbf{X} \to \mathbf{Y}$. Then by definition, for all $x \in \mathbf{X}$ and all orbit $\mathcal{O} \in \mathrm{Orbits}^\omega(P; (x, \mathbf{initial}(P)))$, there exists an injection $\iota : \mathbf{N} \to \mathbf{N}$ and a corresponding orbit $\mathcal{O}' \in \mathrm{Orbits}^\omega(Q; (\phi(x), \mathbf{initial}(Q)))$. We say that P is intentionally simulated with $f(n)$ transition cost expansion and with $g(n)$ configuration cost expansion when for all such choices of x and $\mathcal{O}$, we have:*

$$t^\beta(\mathcal{O}'_{\iota(k)}) \leqslant f(t^\alpha(\mathcal{O}'_k)),$$

$$c^\beta(\mathcal{O}_{\iota(k)}) \leqslant g(c^\alpha(\mathcal{O}'_k)).$$

Definition 5.6.3 *Let $f, g : \mathbf{N} \to \mathbf{N}$ be functions and $(\alpha, c^\alpha, t^\alpha)$, $(\beta, c^\beta, t^\beta)$ be QMCs. We say that α is program-wise intentionally simulated by β* with $f(n)$ transition cost expansion and with $g(n)$ configuration cost expansion *when for all program $P \in \mathrm{Programs}(\alpha)$ there exists a program $Q \in \mathrm{Programs}(\beta)$ which intentionally simulates P with $f(n)$ transition cost expansion and with $g(n)$ configuration cost expansion.*

Example 5.6.1 We can consider the standard configuration and transition costs s_k, t_k for the AMC $\alpha_{k\mathrm{TM}}$ (Example 4.4.1) defined as:

$$s_k : ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto k \times |((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}}|$$

where $|((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}}|$ is equal to

$$\max_i\{(\forall j, i > h^j) \wedge (\exists j s_i^j \neq 0)\} - \min_i\{(\forall j, i < h^j) \wedge (\exists j s_i^j \neq 0)\},$$

and

$$t_k : \begin{cases} \mathtt{left}^{(\rho)}, ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto 1 \\ \mathtt{right}^{(\rho)}, ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto 1 \\ \mathtt{write}_*^{(\rho)}, ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto 1 \\ \mathtt{read}_*^{(\rho)}, ((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}} \mapsto 0 \end{cases}$$

As we have shown in Proposition 4.4.5, the AMC $\tilde{\alpha}_{\mathrm{TM}}$ program-wise simulates the AMC $\alpha_{k\mathrm{TM}}$. What about the quantitative aspects, i.e. if one considers the QMCs $(\tilde{\alpha}_{\mathrm{TM}}, s, t)$ and $(\alpha_{k\mathrm{TM}}, s_k, t_k)$? Suppose that $P \in$

Programs($\alpha_{k\text{TM}}$) is simulated by the program $\tilde{P} \in$ Programs($\tilde{\alpha}_{\text{TM}}$) defined in the proof of Proposition 4.4.5.

We first consider the configuration cost functions. Recall that $\phi : \mathbf{X}_{k\text{TM}} \to \tilde{\alpha}_{\text{TM}}$ maps a configuration $((s_i^j)_{i\in\mathbf{Z}}, h^j)_{j\in\{1,\dots,k\}}$ to a configuration split into blocks of $3k$ symbols corresponding to a position $j \in \mathbf{Z}$ such that at least one of the tape has a non-$\star$ symbol at position j. Hence $s(\phi(x)) \leqslant 3ks_k(x)$. This shows the simulation has a $g(n) = 3kn$ configuration cost expansion, i.e. a $g(n) - n = (3k-1)n$ linear overhead.

Now, given an orbit $\mathcal{O} \in \text{Orbits}^n(P; x)$, we have already explained that a computation $(x, s) \xrightarrow{\texttt{inst}} (x', s')$ is simulated by the computation:

$$(\phi(x), s, i) \to \cdots \to (\phi(x'), s, i).$$

We need to understand what is the cost of this orbit. The computation can be split into steps as follows:

$$\begin{aligned}(\phi(x), s, i) &\to \cdots \to (x_1, s, s_r)\\ &\to \cdots \to (x_1, s, u)\\ &\to \cdots \to (x_1, s, b)\\ &\to \cdots \to (x_1, s, i)\end{aligned}$$

The first step is of fixed length (depending on the tape ρ on which the instruction acts), and bounded by $2k$. The third step depends on the instruction but is always of length bounded by $4k$. The second step correspond to the search of the position of the ρ-th head from the position of the first head, while the fourth step corresponds to the search of the position of the first head from the position of the ρ-th head. Both are thus bounded in length by $|h^\rho - h^1|$, a value bounded by the configuration cost $c_k(x)$. We therefore have a bound on the cost of the orbit $\mathcal{O}' \in \text{Orbits}^{n'}(\tilde{P}; x)$ corresponding to $\mathcal{O}$:

$$t_k(\mathcal{O}') \leqslant t_k(\mathcal{O}) \times (6k + 2c_k(\mathcal{O})).$$

This shows that the simulation has $f(n) = 2n^2 + 6kn$ transition cost expansion. This can be turned into a notion of overhead (which is more standard in complexity theory) by considering $f(n) - n$. In this case, this gives a quadratic overhead. As a consequence, the simulation of P by $\tilde{P}$ has a constant $3k$ overhead, and a linear time overhead in the configuration cost.

Now, in that specific case one can also show that the configuration cost grows at most linearly in the transition cost. Indeed, for each instruction `inst` and configuration $x \in \mathbf{X}_{k\text{TM}}$, we have $c_k(\alpha_{k\text{TM}}(\texttt{inst})(x)) \leqslant c_k(x) + t_k(\texttt{inst})$. Using this, we obtain:

$$t_k(\mathcal{O}') \leqslant t_k(\mathcal{O}) \times (6k + 2t_k(\mathcal{O})),$$

that is a transition cost overhead of $t_k(\mathcal{O}) \times (6k + 2t_k(\mathcal{O})) - t_k(\mathcal{O})$, i.e. a quadratic overhead.

Remark 5.6.2 Let us note that these notions allow to consider models that are usually described as *unrealistic* with a cost model that turns them into realistic models! Consider for instance the BSS model over rational

numbers. With naive configuration cost model

$$c(f) = \min\{i \in \mathbf{Z} \mid \forall j > i, f(j) = 0\} - \max\{i \in \mathbf{Z} \mid \forall j < i, f(j) = 0\}$$

and naive transition cost model (counting a unitary cost to tests, and to multiplications/additions[9]), this is indeed a model that is not realistic.

I however argue that it is possible to associate a cost model to this model such that it can be simulated by a Turing machine with bounded (and even linearly bounded) configuration and transition cost expansions. Indeed, let us consider the following cost model, where $|\frac{p}{q}| = \log_2(p) + \log_2(q)$:

$$c'(f) = \sum_{i \in \mathbf{Z}} |f(i)|.$$

This should allow for such a simulation. Similarly, one could define a realistic transition cost model, although the definition would be very complicated[10].

9: I avoid writing down the definition, but this would coincide with defining the cost of applying some polynomial to be the number of vertices in the algebraic computation trees computing it.

10: It can be more easily defined in the algebraic model (Example 3.2.7)), in which we only compute simple operations (additions, multiplications); in this case, defining for instance $t'(\texttt{add}(n, m)) = t'(\texttt{mul}(n, m)) = |m| \times |n|$ should allow for a simulation with a linearly bounded transition cost expansion.

5.7. Universal programs and hierarchy theorems

In this section, we will define universal programs. Let us start with a definition. Note that this section is concerned only with deterministic programs.

Definition 5.7.1 *A universal program for an* AMC α *is an abstract* α*-program* U *and an embedding* ψ_U *of* $\text{Programs}(\alpha) \times \mathbf{X}$ *into* $\mathbf{X}$ *such that for all* $P \in \text{Programs}(\alpha)$, $U \cdot (\psi_U(P, \_))$ *– a program on* $\mathbf{X}$ *– intensionally simulates* P*.*

The configuration and transition cost expansions of U *are defined as the configuration and transition cost expansions of the program-wise intentional simulation, when well defined.*

More precisely, a universal program U is such that for all program $P \in \text{Programs}(\alpha)$, for all $x \in \mathbf{X}$, and for all orbit $\mathcal{O} \in \text{Orbits}^{\omega}(P; (x, \textbf{initial}(P)))$, there exists an orbit $\mathcal{O}' \in \text{Orbits}^{\omega}(U; (\psi_U(P, x), \textbf{initial}(P)))$ and an injection $\iota : \mathbf{N} \to \mathbf{N}$ satisfying:

$$\forall k \in \mathbf{N}, \mathcal{O}'_{\iota(k)} = \phi_U(M, \mathcal{O}_k).$$

We will now consider $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ and first notice that – if one restricts to finite sets of states:

$$\text{Card}(\text{Programs}(\alpha)) = \text{Card}(\mathbb{M}(I)) = \max\{\text{Card}(I), \omega\}.$$

We write $\kappa = \max\{\text{Card}(I), \omega\}$, and we fix a bijective map

$$\theta : \text{Programs}(\alpha) \to \kappa.$$

The goal is now to prove a hierarchy theorem w.r.t. some data structure $\mathtt{D}$ and an interpretation Δ. We will focus on programs computing functions from Δ to $\Delta_{\mathcal{B}}$. To obtain the theorem in a general abstract way but keep the statements tractable, we will require some additional hypotheses that we will introduce in a lazy way (as they are needed).

First, we suppose that there exists an implementation in α of the data structure $\mathtt{C}$ whose underlying domain is $\mathcal{C} = \mathcal{D}\times\mathbf{N}$ and with the structural maps $\mathtt{pred} : (d,n) \mapsto (d,n-1)$, $\mathtt{read}_0 : (d,n) \mapsto (d,n)$ defined only when $n = 0$, and $\mathtt{read}_S : (d,n) \mapsto (d,n)$ defined only when $n \neq 0$. We will write $b_p : \mathcal{D}\times\mathbf{N} \to \mathbf{N}$ a bound (we suppose it exists) on the transition cost of the program Pred implementing $\mathtt{pred}$, i.e. for all $(d,n) \in \mathcal{C}$,

$$((d,n-1), \mathbf{terminal}(\mathrm{Pred})) \in \mathrm{Orbits}^{b_p(d,n)}(\mathrm{Pred}; ((d,n), \mathbf{initial}(\mathrm{Pred}))).$$

Similarly, we write $b_r : \mathcal{D}\times\mathbf{N} \to \mathbf{N}$ a bound on the transition cost of the implementations of $\mathtt{read}_0$ and $\mathtt{read}_S$.

Second, we consider an α-computable function $f : \Delta(d) \to \mathbf{N}$, i.e. we will suppose that there exists a program $P_f \in \mathrm{Programs}(\alpha)$ whose graph satisfies

$$\{(d, f(d)) \mid d \in \mathcal{D}\} \subset \mathbf{Gr}(P_f).$$

We will write $b_f : \Delta(d) \to \mathbf{N}$ a bound (we suppose it exists) on the transition cost of the orbit $\mathcal{O}_d$ from $(d, \mathbf{initial}(P_f))$ to $((d, f(d)), \mathbf{terminal}(P_f))$, i.e. $t^\alpha(\mathcal{O}_d) \leqslant b_f(d)$.

Third, we suppose that the transition complexity of the instructions are not impacted by considering pairs. More precisely, we suppose that $t^\alpha(\iota, x) = t^\alpha(\iota, (x,n))$.

We can then prove the following essential lemma.

Lemma 5.7.1 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be an AMC, $P \in \mathrm{Programs}^{\Delta\to\Delta_\mathcal{B}}(\alpha)$ be a program and $f : \mathbf{X} \to \mathbf{N}$ an α-computable function.*

There exists a program $\tilde{P}_{\leq f} \in \mathrm{Trans}_\alpha^{\Delta\to\Delta_\mathcal{B}}()$ simulating P on x for at most $f(x)$ steps, and outputs 0 if P has not reached a terminating state in those $f(x)$ steps. Moreover, if P has transition complexity bounded by a function $g : \mathcal{D} \to \mathbf{N}$, the program $\tilde{P}_{\leq f}$ has transition complexity bounded by $g(d) + f(d) \times (b_p(f(d)) + b_r(f(d))) + b_f(d)$.

Proof. We construct P_f as follows:

- the initial state $\mathbf{initial}(\tilde{P}_{\leq f})$ is identified with $\mathbf{initial}(P_f)$, the program that computes $(d, f(d))$ from d;
- we then identify $\mathbf{terminal}(P_f)$ with $\mathbf{initial}(P')$ where P' is obtained by interleaving P with the program Pred: it is defined with states $S^P \times S^{\mathrm{Pred}}$ with the set of edges defined as follows:
  - for each edge $e \in E^P$, we have an edge
    $$(m_e \circ \mathtt{read}_S, (i_e, \mathbf{terminal}(\mathrm{Pred})) \to (o_e, \mathbf{initial}(\mathrm{Pred})));$$
  - for each edge $f \in E^{\mathrm{Pred}}$ and $s \in S^P$ we define the edge
    $$(m_f, (s, i_f) \to (s, o_f));$$
  - for all $s \in S^P \setminus \{\mathbf{terminal}(P)\}$, we add the edges
    $$(\mathtt{read}_0, (s, \mathbf{terminal}(\mathrm{Pred})) \to \mathbf{terminal}(\tilde{P}_{\leq f}));$$
  - we identify states $(\mathbf{terminal}(P), \mathbf{terminal}(\mathrm{Pred}))$ with the state $\mathbf{terminal}(\tilde{P}_{\leq f})$.

Now, one can check that for all $x \in \mathbf{X}$ there exists $y \in \mathbf{X}$ such that $(x, y) \in \mathbf{Gr}(P_f)_{f(\cdot)}$. Moreover, $\mathbf{Gr}(P)_{f(\cdot)} \subset \mathbf{Gr}(P_f)_{f(\cdot)}$.

The last thing to check is the transition cost of P_f supposing that P has transition complexity bounded by g. From the definition and the assumptions, an orbit of $\tilde{P}_{\leq f}$ starting at $(\Delta(d), \mathbf{initial}(\tilde{P}_{\leq f}))$ is composed by:

- an orbit from $(\Delta(d), \mathbf{initial}(\tilde{P}_{\leq f}))$ to $((\Delta(d), f(d)), \mathbf{initial}(P'))$, together with a use of $\mathtt{read}_0/\mathtt{read}_S$ are each step; the transition is therefore bounded by $b_f(d) + d \times b_r(f(d))$;
- an orbit from $((\Delta(d), f(d)), \mathbf{initial}(P'))$ to some $(y, \mathbf{terminal}(\tilde{P}_{\leq f}))$ which is obtained as an interleaving of an orbit of length at most $f(d)$ of P starting at $(\Delta(d), \mathbf{initial}(P))$ with transition complexity bounded by g, and orbits from $((x, k), \mathbf{initial}(\mathrm{Pred}))$ to $((x, k-1), \mathbf{terminal}(\mathrm{Pred}))$ in Pred, each of them with transition complexity bounded by $b_p(f(d))$.

This gives the required bound. ❀

Now, this lemma can be used to prove the following proposition. Here we suppose that b_p, b_r are constant and b_f is $O(g \circ h)$[11].

Proposition 5.7.2 *Let g, h be α-computable functions $\mathbf{N} \to \mathbf{N}$. If there exists a universal machine U for α with transition cost expansions bounded by g, then the following language*[12]*:*

$$\mathcal{L}^h = \{\phi_U(M, d) \in \mathbf{X} \mid M \cdot \Delta(d) \to_g 1\}$$

belongs to the class $\textsc{Trans}^{\phi_U \to \Delta_{\mathcal{B}}}_{\alpha, c^\alpha, t^\alpha}(g \circ h)$.

Proof. Note that ϕ_U can be understood as the interpretation of the abstract data domain $\kappa \times \mathcal{D}$.

To prove this statement, we define a program deciding whether a pair (M, d) belongs to $\mathcal{L}$. This program $\tilde{M}$ needs to run M on x for h steps. This is equivalent to running U on $\phi_U(M, x)$ for $g \circ h$ steps. From the previous lemma, this can be done in transition cost $g \circ h(d) + g \circ h(d) \times (b_p(g \circ h(d)) + b_r(g \circ h(d))) + b_f(d)$, hence $g \circ h(d)(1 + b_p + b_r) + b_f$. Since we assumed that b_p and b_r are constant, and that $b_f = O(g \circ h)$, we have that $\tilde{M}$ has transition cost in $O(g \circ h)$, i.e. it belongs to $\textsc{Trans}^{\phi_U \to \Delta_{\mathcal{B}}}_{\alpha, c^\alpha, t^\alpha}(g \circ h)$. ❀

Now that we have shown that $\mathcal{L}^h$ belongs to the class

$$\textsc{Trans}^{\phi_U \to \Delta_{\mathcal{B}}}_{\alpha, c^\alpha, t^\alpha}(g \circ h),$$

we will show that it is not in

$$\textsc{Trans}^{\phi_U \to \Delta_{\mathcal{B}}}_{\alpha, c^\alpha, t^\alpha}(h).$$

One assumption used in the following proof is that the `not` operation on booleans can be performed in[13] $O(h)$ transition complexity. Moreover, we will require that $\mathcal{D} = \kappa$ and that there exists a program Q which on input $(\Delta(d), \mathbf{initial}(Q))$ reaches $(\phi_U(\Delta(d), \Delta(d)), \mathbf{terminal}(Q))$ in transition complexity in[14] $O(h)$.

11: These assumptions are verified in the case of Turing machines. In particular b_p and b_r are constant when the counter is represented on a separated tape.

12: We write here $M \cdot \Delta(d) \to_g 1$ to denote that the orbit of M at $\Delta(d)$ reaches $(\Delta_{\mathcal{B}}(1), \mathbf{terminal}(P))$ with transition complexity in $O(g(c^\alpha(\Delta(d))))$.

13: In the case of Turing machines, this is done in constant transition complexity.

14: In the case of Turing machines, this is done in $O(\log^2(d))$.

Given a program P in $\mathrm{Trans}^{\phi_U \to \Delta_{\mathcal{B}}}_{\alpha, c^\alpha, t^\alpha}(h)$, we define its *language* as follows:

$$\mathrm{Lang}(P) = \{\phi_U(P, d) \mid (\Delta_{\mathcal{B}}(1), \textbf{terminal}(P)) \in \mathrm{Orbits}^\omega(P; \phi_U(P, d)).$$

Theorem 5.7.3 *There are no P in $\mathrm{Trans}^{\phi_U \to \Delta_{\mathcal{B}}}_{\alpha, c^\alpha, t^\alpha}(h)$ such that $\mathrm{Lang}(P) = \mathcal{L}^h$.*

Proof. This is the standard diagonalisation proof. Suppose there exists such a program P. Then consider the program $\tilde{P}$ obtained by pre-composing P with a program that outputs $\phi_U(\theta^{-1}(x), x)$ and post-composing it with a program implementing the `not` operation on the output. The resulting program simply applies P to $\phi_U(M, \langle M \rangle)$ and negates the result. By the assumption that both the program that produces $\phi_U(\theta^{-1}(x), x)$ from x and the implementation of `not` are in $O(h)$ transition complexity, $\tilde{P}$ belongs to $\mathrm{Trans}_{\alpha, c^\alpha, t^\alpha}(h)$ as well.

We can now ask whether $\langle \tilde{P} \rangle$ is accepted by $\tilde{P}$ or not. Since $\tilde{P}$ belongs to $\mathrm{Trans}_{\alpha, c^\alpha, t^\alpha}(h)$, it produces either 0 or 1 on $\langle \tilde{P} \rangle$. If it produces a $\Delta_{\mathcal{B}}(0)$, it means that $P(\tilde{P}, \langle \tilde{P} \rangle)$ was producing a $\Delta_{\mathcal{B}}(1)$, which means that $\tilde{P} \cdot \langle \tilde{P} \rangle \to_g 1$, a contradiction. If it produces a $\Delta_{\mathcal{B}}(1)$, it means that $P(\tilde{P}, \langle \tilde{P} \rangle)$ was producing a $\Delta_{\mathcal{B}}(0)$, meaning that $\tilde{P} \cdot \langle \tilde{P} \rangle$ did not reach $\Delta_{\mathcal{B}}(1)$ in $O(g(\Delta(d)))$. Since we assumed that $\tilde{P}$ belongs to $\mathrm{Trans}_{\alpha, c^\alpha, t^\alpha}(h)$, this means that $\tilde{P} \cdot \langle \tilde{P} \rangle$ has reached $(\Delta_{\mathcal{B}}(0), \textbf{terminal}(\tilde{P}))$, another contradiction.

This ends the proof, as we have shown that supposing that the existence of a program P in $\mathrm{Trans}_{\alpha, c^\alpha, t^\alpha}(h)$ such that $\mathrm{Lang}(P) = \mathcal{L}^h$ leads to a contradiction. ❀

While the proof here is contrived and requires lots of assumptions, it should be noted that it does not rely on any assumptions on the cost model considered. As a consequence, it allows to recover standard time hierarchy theorems but also hierarchy theorem for other notions of transition cost (e.g. accounting for energy consumption rather than time).

I expect that these abstract hierarchy theorem could clarify the structure of the standard time hierarchy theorems [92–94]. Indeed, by enforcing the consideration of every step and clarifying every bound, the role of every assumption and how they impact the end result is made explicit.

While I did not develop it in this document, a proper configuration cost hierarchy theorem can be obtained in a similar way, generalising the space hierarchy theorem.

[92]: Hartmanis et al. (1967), *On the Computational Complexity of Algorithms*
[93]: Sudborough et al. (1976), *On Families of Languages Defined by Time-Bounded Random Access Machines*
[94]: Jones (1993), *Constant time factors do matter*

Abstract specification and algorithms

6.

The terminology "algorithm" predates computers, and in fact does not appear in the first papers about computability. It was used to refer mostly to methods of resolution in mathematics (leading to the job title of "algoriste" [95]). The general use within computer science seems to have its origin within the Russian school (Markov [96], then Kolmogorov [97]), even though the terminology is used by Church already in 1936 [98]. The word quickly gained traction in computer science (even leading to naming a subfield of theoretical computer science: algorithmics). In recent years, a third use has eclipsed the two previous ones (at least in term of everyday use) by being used outside of the computer science community to AI systems (what we would call *models* trained for, e.g. recommandation), but also to some extent to all computer systems.

[95]: Lamassé (2013), *Relationships between French "practical arithmetics" and teaching?*

[96]: Markov (1954), *The theory of algorithms*

[97]: Kolmogorov et al. (1958), *On the definition of an algorithm*

[98]: Church (1936), *An Unsolvable Problem of Elementary Number Theory*

These three different usages have been identified by Airoldi in his book [99], in which he separates those into 'eras': the analog era, the digital era, and the platform era. While this reading is extremely useful and clairvoyant, I feel the term *era* hides the fact that today all three usage of the word co-exist: while an era ends to be succeeded by the next, this is not the case here. The word *algorithm* is thus used with those three different meanings depending on the context. Are those notions the same? My answer would be the following.

[99]: Airoldi (2022), *Machine Habitus: Toward a Sociology of Algorithms*

The mathematics (analog) and computer science (digital) notions are the same, and they are somehow captured within the definitions shown below (as an example, I explain in subsection 3 how Euclid's constructions can be represented in this way). There is still a difference between the two in that I believe the computer science notion to be a specific subset of the mathematical one, being related to physical constraints. One could say that digital algorithms are captured by digits, but my view is different: how could this point of view arguably be extended to analog computers, quantum computers, biological computing? Formally distinguishing what is a computer system and what is simply a mathematical, abstract, device, is difficult, maybe an impossible task. And ever evolving: we can now compute with adiabatic systems, but this was just a thought experiment a few years back.

Are those notions related in any way with the third, fashionable, sense? I also would say it is, although I believe there is at the origin of this use a crude misunderstanding of computer systems leading to a confusion between the three notions I have already spent many pages distinguishing properly: computation, programs, source code, and algorithms. At appears that in most cases, the word is meant to be used to refer to the program (or trained model), while indeed referring to properties and issues with *some* algorithm the latter implements. I stress the word 'some' here, as my understanding is that a given program never implements a unique algorithm but a multitude of algorithms. To illustrate this, we can read things such as

> The *Faceflicks* algorithm is a set of ranking signals powered by machine learning and artificial intelligence. It calculates

> which content is most likely to appeal to each user and then delivers them a personalized feed.

If we overlook some of the obvious issues with the sentence (like the use of the word "powered"), we see that the part of the quote is about the algorithm: it produces a personalised ranking. However, it also refers to the program: only a program calculates, while an algorithm will tell you how the calculation should be structured. So, overall, my point of view is that most of the discourse on *algorithms* as understood in the platform era of Airoldi are indeed just this: algorithms in the same historical sense, i.e. some algorithm that is implemented by the trained model (which, I recall, is a program). But no, this algorithm is not hosted on the company's private servers, and in fact it cannot be tweaked directly: only the model has a physical reality. The cultural aspects of trained models, notably for personalised recommendation, seem essential, and a complete understanding of the production and use of those is essential. We note in particular the beautiful idea put forth (with much more care than what follows) by Airoldi that recommandation models are a material embodiment of Bourdieu's notion of *habitus*, an idea supported by quotations from Bourdieu himself:

> "C'est une espèce de machine transformatrice qui fait que nous « reproduisons » les conditions sociales de notre propre production, mais d'une façon relativement imprévisible. [. . .] On peut le penser par analogie avec un programme d'ordinateur (analogie dangereuse, parce que mécaniste), mais un programme autocorrectible."
>
> Pierre Bourdieu
> « Le marché linguistique », in Questions de sociologie,
> Éditions de Minuit, 1980.

Let us note however that without access to the model itself, some of its properties can still be deduced. In some way, this corresponds to finding refined algorithms that the program implements. I.e. say a program is supposed to select applicants being interviewed for a job (a real example [100]). This program is a black box, and all we know of it is that it takes in some information (e.g. the candidate's curriculum vitae) and outputs some answer (whether the applicant will be interviewed, i.e. a yes or no answer). This is an algorithm implemented by the program, but a very crude one. What if we are now able to probe the program, i.e. provide it with different curriculum vitae and check the answer. What can be deduced from it? We may be able to see that it will exclude some applicants based on their diploma (e.g. answer "no" for applicants with a PhD, or without a MBA). Obviously, we cannot know for certain that it will do so, and can only compute statistics and say that with high probability the program does implement a more refined algorithm which takes in a CV, checks the diploma and then either directly rejects or perform more computations before answering. However, neither the program itself or the source code can be probed in this way: one cannot deduce which languages or data structures were used to write the source code, how it was compiled, it is not possible to know on which architecture the program is defined, etc. In such a blackbox situation, the algorithm is the one entity on which one can experiment. In fact, the only information about a program that one can expect to obtain concerns the algorithms it

[100]: O'neil (2017), *Weapons of math destruction: How big data increases inequality and threatens democracy*

implements. Is this sufficient to, e.g. ensure that the law is respected? The kind of information one can deduce seems limited. Moreover, it requires unlimited queries to the program, something that is more often than not, impossible in practice. These questions around how to put into place such black box testing relating to *auditing* lead to interesting technical ("how to perform a black box analysis?" [101, 102]) and juridic ("how to access the needed information?") issues, but I also expect theoretical investigations to emerge from this (maybe based on material from the following sections), notably answering questions such as "what exactly can be deduced about the program?".

[101]: Merrer et al. (2021), *Setting the Record Straighter on Shadow Banning*
[102]: Merrer et al. (2023), *Modeling rabbit-holes on YouTube*

This is clearly a pressing current societal issue, and even more when one considers the way the program is produced. Indeed, the literature in biases is already well supplied, and most of it has been focussed on how to prevent biases in trained models. However, one interesting and less understood aspect of it is the social condition of their production. The work of Jaton [103] explains well how not only the training data, but also the process of annotation, impacts the produced model. In fact, we have moved from a situation in which a team of developers worked together to produce precise computer code whose behaviour was (for the most part) understood, to a situation in which the obtained program is designed by independent workers from the choice of corpus and the way it is annotated, without these workers consciously knowing their work impacts the resulting program. Theoretically, there is an interesting point to be made here (oblivious to the social aspects): these trained models usually answer an underspecified question. I.e. deciding if a picture shows a dog or a cat can sometimes have no answer. As a consequence, the computation that is expected does not coincide with a function Pictures $\to \{0, 1\}$. In fact, if one were asked which function the model should compute, one could not answer. All we can do is give examples, i.e. all we know are sample points in the space Pictures for which the answer is known. This is easily seen in work coming from adversarial AI, where models are used to produce edge cases such as a picture of a dog that is also the picture of a cat. The choice of corpus correspond to those sample points. But annotations give additional information which is used in the training process. It may for instance indicate some part of the picture to indicate where the animal is: the form and placement of these annotations will impact the trained model. Lastly, the training method used (what AI researcher actually call the algorithm) will also impact the model. We therefore have three interdependent actions which, together, produce a specific program. However, how these actions, independently and combined, affect the resulting program remains mostly unclear for the time being.

[103]: Jaton (2021), *The Constitution of Algorithms: Ground-Truthing, Programming, Formulating*

6.1. Background

We will for the moment forget part of the questions listed above to focus on the notion of algorithm itself, without considering the conditions of their production and use, as well as their (undeniable) impact.

We will focus on the following question: can the notion of algorithm be formalised? As we have seen, some conceptual analysis is needed beforehand, as one could ask what exactly is meant by the word "algorithm"

here. As explained above, my point of view is that there exists a notion of algorithm underlying all three uses above, and which may have in fact, more reality than the notion of program.

The question of formalising algorithms has been discussed by authors in the past. Most notably, Moschovakis [104, 105] and Gurevich [106, 107] both came up with proposed definitions. My understanding is that their proposals are different but not unreconcilable. An idea that subsumes Gurevich approach is that of an algorithm as a specification. The proposal below shares this point of view, but proposes a notion of algorithm that we feel is more satisfactory in that:

- its relation with other notions (programs, data structures, etc.) is much clearer than with abstract state machines which somehow conflate programs and algorithms[1];
- abstract state machines are less natural mathematical notion: our formalisation uses standard mathematics, providing numerous tools and methods;
- in practice, it is not possible to compare two abstract state machines (for instance saying that one somehow subsumes the other), whereas this can be done with the definition given below.

[104]: Moschovakis (2001), *What is an Algorithm?*
[105]: Moschovakis (1998), *On Founding the Theory of Algorithms*
[106]: Gurevich (2012), *What Is an Algorithm?*
[107]: Blass et al. (2003), *Algorithms: A quest for absolute definitions*

1: This is in particular illustrated by the fact that one can define an AMC of abstract states machines (subsection 3.2).

One important criticism of Gurevich approach is that of Moschovakis, in particular with respect to complexity. Again, we will discuss later how complexity issues can be formalised within our proposal, and illustrate it with Moschovakis' favorite example: merge sort.

6.2. A new proposal

I will present the proposed mathematical definition of algorithms in two steps. This decomposition is interesting in itself, because the use of the term *algorithm* can refer to any of the proposed definitions below. The first notion is that of *syntactic algorithm*. A syntactic algorithm provides information on how many different actions are performed in the algorithm and how they are arranged. It is in fact an approximation of what the general consensus is on the definition of algorithm, since it does not provide any information about the *semantics*, i.e. it does not impose restrictions on what the different actions actually do. This puts into light an important aspect of algorithms: implicit assumptions. Let us illustrate that point.

Consider the following algorithm:

Input: x, y
While $y \neq 0$:
    While $x \geq y$:
        $x = x - y$;
    $z = x$;
    $x = y$;
    $y = z$;
Output x

Anyone with some mathematics and/or computer science education will recognise this as Euclid's algorithm to compute the gcd. But there is some hidden assumption: what does it means to compute $x - y$? If asked, most people will answer that x and y are integers and "$-$" refers to

the usual subtraction. But some others may answer that x and y are real numbers, or elements of a finite field. I have not conducted experiments, but I expect that none (or a negligible number of them) will answer that the symbol "$-$" is something other than some kind of subtraction. But if we agree that x and y can represent any objects taken in a fixed domain, how can one decide what makes $-$ a subtraction or not?

In most occurrences of the above algorithm, some implicit interpretation will be given by the context. But is the context part of the algorithm? And to what extent this context should be part of it? Leaving aside the fact that Euclid did not write the algorithm in this way, he actually writes it twice in the Elements [108–110]. In book 7, he exposes the algorithm for integers, while in book 10 he exposes it for lines and segments. Are those the same algorithms? In some sense they are, and this would be the general agreement. But subtracting an integer to another is not the same as subtracting a segment from another. The relationship between both objects is complex: the are uncountably many segments but countably many integers, so there is no hope to encode one algorithm into the other (or not without introducing many complex operations: quotients, etc.).

[108]: Heath (1956), *The Thirteen Books of Euclid's Elements*
[109]: Heath (1956), *The Thirteen Books of Euclid's Elements*
[110]: Heath (1956), *The Thirteen Books of Euclid's Elements*

Syntactic algorithms

Definition 6.2.1 *A syntactic algorithm is a finite labelled graph (with control states)* $A = (S, \mathbf{initial}(A), \mathbf{terminal}(A), E, s, t, L, \ell)$ *where* S *is a finite set of* states *containing the initial and terminal states* $\mathbf{initial}(A)$ *and* $\mathbf{terminal}(A)$, (S, E, s, t) *is a directed graph,* L *a finite set of* labels, *and* $\ell : E \to L$ *is a labelling function.*

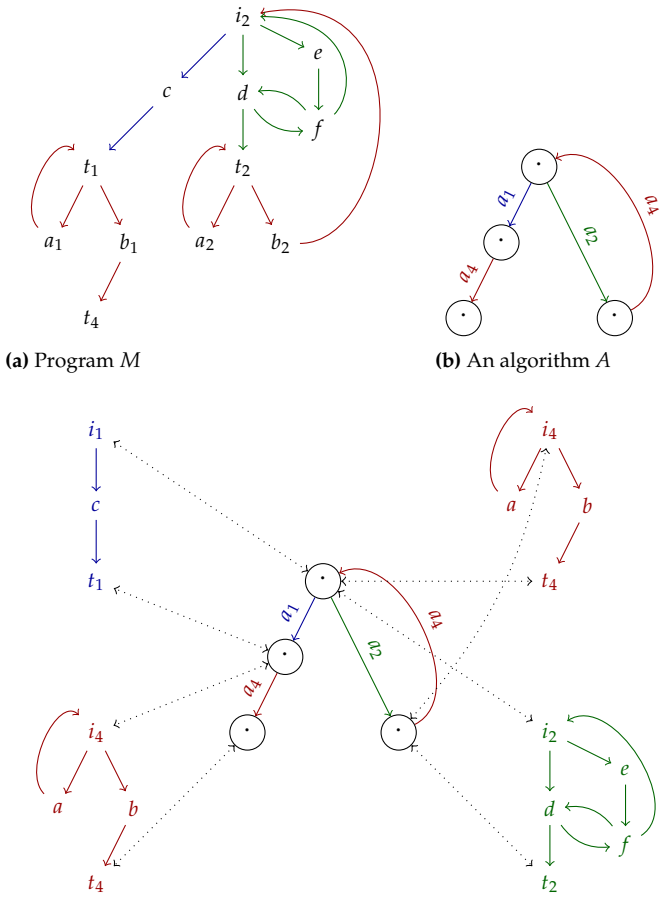
The set of all algorithms is written as $\mathrm{Algorithms}(*)$.

The guiding intuition behind this definition is that the graph describes a general control structure, while the labels corresponds to the name of operations used in the algorithm. For the moment, the algorithm is *syntactic* because the labels are just arbitrary names. The notions introduced in the next sections refine the definition by associating labels to more restricted classes of operations. Before going through those definitions, we can already define what it means for a program to *implement* a syntactic algorithm.

Definition 6.2.2 (Glueing of graphings along a labelled graph) *Suppose given a syntactic algorithm* $A = (V, S, \mathbf{initial}(A), \mathbf{terminal}(A), E, s, t, L, \ell)$, *and an* AMC $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$. *Suppose moreover given a map* $\phi : L \to \mathrm{Programs}(\alpha)$. *The pre-glueing of* ϕ *along* A *is the machine* $\mathcal{P}(A, \phi)$ *defined as the disjoint union* $\sum_{e \in E} \phi(\ell(e))$. *The* glueing of ϕ along A *is then defined as the program* $\mathcal{G}(A, \phi)$ *obtained by identifying for all* $v \in V$ *the states* $\{\mathbf{initial}(e, \phi(e)) \mid s(e) = v\} \cup \{\mathbf{terminal}(e, \phi(e)) \mid t(e) = v\}$, *and setting* $\mathbf{initial}(\mathcal{G}(A, \phi)) = \mathbf{initial}(A)$ *and* $\mathbf{terminal}(\mathcal{G}(A, \phi)) = \mathbf{terminal}(A)$.

We illustrate in Figure 6.1 how a program can be understood as a glueing along an algorithm.

Definition 6.2.3 *A program* G implements *the algorithm* A *when there exists* $\phi : L \to \mathrm{Programs}(\alpha)$ *such that* G *is the glueing of* A *along* ϕ.



(a) Program M

(b) An algorithm A

(c) M seen as a glueing along an algorithm A

Figure 6.1.: Example of glueing

6.3. Specified algorithms

The next step is to impose that labels correspond to some structure. In particular, we may want to indicate specific maps defined as part of data structures. In practice, these data structures are implicitly given when writing down algorithm, e.g. using pseudo-code.

We remark that if an algorithm uses two data structures, say integers and booleans, or simply manipulates several elements of a given data structures, the implicit assumption is that several copies of these data structures are simultaneously interpreted within the implementing model of computation. Formally, this is equivalent to considering products of data structures: if $\mathtt{D} = (\mathcal{D}, S)$ and $\mathtt{D}' = (\mathcal{D}', S')$ are data structures, then their product $\mathtt{D} \times \mathtt{D}'$ is defined as $(\mathcal{D} \times \mathcal{D}', S \bar{\times} S')$ where:

$$S \bar{\times} S' = \{s \times \mathrm{Id}_{\mathcal{D}'} \mid s \in S\} \cup \{\mathrm{Id}_{\mathcal{D}} \times s' \mid s' \in S'\}.$$

As a consequence, it is possible to define algorithms with respect to a single data structure.

Definition 6.3.1 *Let* $\mathtt{D} = (\mathcal{D}, S)$ *be a data structure. A* specified algorithm *w.r.t.* $\mathtt{D}$ *is a tuple* $A = (V, \mathbf{initial}(A), \mathbf{terminal}(A), E, s, t, L, \ell, [\![\cdot]\!])$ *where:*

- $(V, \mathbf{initial}(A), \mathbf{terminal}(A), E, s, t, L, \ell)$ *is a syntactical algorithm;*

- $[\![:]\!]L \to S$ *maps the labels to structural maps.*

The set of all specified algorithms w.r.t. $\mathtt{D}$ *is written* Algorithms($\mathtt{D}$).

Definition 6.3.2 (Coherent labelings) *Suppose given a specified algorithm* $A = (V, S, \mathbf{initial}(A), \mathbf{terminal}(A), E, s, t, L, \ell, [\![\cdot]\!])$ *for a data structure* $\mathtt{D} = (\mathcal{D}, S)$*, and an* AMC $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$*. A map* $\phi : L \to \text{Programs}(\alpha)$ *is* coherent *with* $[\![\cdot]\!]$ *with respect to an interpretation* Δ *of* $\mathcal{D}$ *when for all* $o \in L$*,* $\phi(o)$ *is an implementation of* $[\![o]\!]$ *w.r.t.* Δ*.*

Definition 6.3.3 *A program* G implements *the specified algorithm* A *when there exists* $\phi : L \to \text{Programs}(\alpha)$ *coherent with* $[\![\cdot]\!]$ *such that* G *is the glueing of* A *along* ϕ*.*

This notion of specified algorithm could probably be stronger than what one would expect. One important remark here is that the notion of data structure is set-theoretic and requires one to fix a concrete domain. As a consequence, the notion does not allow to identify euclidean division on integers and euclidean division on polynomials because the underlying abstract data domain is different. The following proposed definition should allow for a satisfactory solution to this problem.

Logically specified data structures

Definition 6.3.4 *A* logical data structure $\mathcal{D}$ *is defined as first-order logical theory over a first-order language* (Var, Fun, Rel)*.*

An abstract data structure $\mathtt{D}$ *is* a model of the logical data structure $\mathcal{D}$ *if it is a model of* $\mathcal{D}$*, i.e. if* $\mathtt{D} \vDash \mathcal{D}$*.*

Definition 6.3.5 *Let* $\mathcal{D}$ *be a logical data structure over a first-order language* (Var, Fun, Rel)*.* A logically-specified algorithm *w.r.t.* $\mathcal{D}$ *is a tuple* $A = (V, \mathbf{initial}(A), \mathbf{terminal}(A), E, s, t, L, \ell, [\![\cdot]\!])$ *where:*

- $(V, \mathbf{initial}(A), \mathbf{terminal}(A), E, s, t, L, \ell)$ *is a syntactical algorithm;*
- $[\![:]\!]L \to \text{Fun} \cup \text{Rel}$ *maps the labels to structural maps.*

The set of all specified algorithms w.r.t. $\mathcal{D}$ *is written* Algorithms($\mathcal{D}$)*.*

It should be clear that given a logically-specified algorithm A w.r.t. $\mathcal{D}$ and a model $\mathtt{D}$ of $\mathcal{D}$, one can deduce the unique induced specified algorithm $A_{\mathcal{D} \vDash \mathtt{D}}$. Now, implementing A boils down to implementing some $A_{\mathcal{D} \vDash \mathtt{D}}$ for a model of $\mathcal{D}$.

Definition 6.3.6 *A program* P *implements the logically specified algorithm* A *if there exists an abstract data structure* $\mathtt{D}$ *such that* P *implements the induced specified algorithm* $A_{\mathcal{D} \vDash \mathtt{D}}$*.*

Note that euclidean division on integers and on arbitrary euclidean ring are but different models of the same logical data structure. The corresponding notion of *logically specified algorithm*, intermediate between the syntactical and specified notions considered above, then allows to identify the euclidean division algorithms defined on different euclidean rings.

Remark 6.3.1 The reader familiar with Gurevich's *abstract state machines* [69] will probably be curious about potential connections with the notion we just developed since both are based on first order structures. While there may be a formal relation between the approaches, I note that Gurevich uses the first-order structure to define the states (i.e. the space underlying the AMC) while it is here used to describe instructions (or rather, more precisely, programs) that can be performed. This seems to be a fundamental difference. One consequence of this is that Gurevich modifies the values of the interpretations of functions and relation symbols, while here the interpretation is fixed once and for all.

6.4. Properties

Ordered structure

We can adapt the definition of *glueing* to define glueing of algorithms (instead of programs) along another algorithm. This leads to the definition of a preorder on the set of algorithms.

Definition 6.4.1 (Glueing of algorithms along a labelled graph) *Suppose given a syntactic algorithm* $A = (V, S, \mathbf{initial}(A), \mathbf{terminal}(A), E, s, t, L, \ell)$, *and an* AMC $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$. *Suppose moreover given a map* $\phi : L \to \mathrm{Algorithms}(*)$. *The pre-glueing of* ϕ *along* A *is defined as the disjoint union* $\mathcal{P}(A, \phi) = \sum_{e \in E} \phi(\ell(e))$. *The* glueing of ϕ along A *is then defined as the algorithm* $\mathcal{G}(A, \phi)$ *obtained by identifying for all* $v \in V$ *the vertices* $\{\mathbf{initial}(e, \phi(e)) \mid s(e) = v\} \cup \{\mathbf{terminal}(e, \phi(e)) \mid t(e) = v\}$, *and setting* $\mathbf{initial}(\mathcal{G}(A, \phi)) = \mathbf{initial}(A)$ *and* $\mathbf{terminal}(\mathcal{G}(A, \phi)) = \mathbf{terminal}(A)$.

If ϕ *maps* L *into* $\mathrm{Algorithms}(D)$ *(for* D *either an abstract data structure or a logical data structure), the resulting algorithm belongs to* $\mathrm{Algorithms}(D)$.

Proposition 6.4.1 *If* B *is obtained as the glueing of* A *along* ϕ, *and* P *is a program implementing* B, *then* P *also implements* A.

Proof. Here is a sketch of the proof, avoiding painful details. If P implements B, then P is a glueing of B along some $\psi : L^B \to \mathrm{Programs}(\alpha)$. Moreover, B is a glueing of A along some $\phi : L^A \to \mathrm{Algorithms}(*)$. Now, from both these maps one can deduce a $\theta : L^A \to \mathrm{Programs}(\alpha)$. It is not difficult to check that P is the glueing of A along this map θ. ❀

Remark 6.4.1 The notion of algorithm we obtain does not correspond to an equivalence relation on programs. In particular, one program implements many different algorithms. In some way, the notion is almost topological, and we could expect some separation axiom to be satisfied (probably the T_0 axiom: that if P and Q are different programs, there exists an algorithm A such that one of P and Q implement A but not the other).

Among the future research in that direction, one direction seems of particular importance: the definition of a notion of *distance*. A distance between algorithms, together with a notion of *implementation/glueing* up to some error ϵ. This could be used to talk about convergence of programs toward an algorithm for instance.

Complexity

We will now define cost models for abstract data structures. First, we will need to fix a notion of *size* on the abstract data domain. Then each operation in the data structure will have an associated cost.

Note that this reveals my point of view that an algorithm does not possess a unique associated complexity (in the same way an abstract program does not possess its own complexity). This is somehow coherent with the general use. If I define an algorithm using matrix multiplication, does its complexity change each time someone[2] finds a faster algorithm for matrix multiplication? Moreover, it would mean that either euclidean division has an associated complexity, whether it applies to natural numbers of polynomials, or that different instances of the algorithms on different abstract data structures should not be identified.

2: Most probably Virginia Vassilevska Williams [111, 112], although there are current limitations [113, 114].

Definition 6.4.2 *A* cost model *for an abstract data structure* $\mathtt{D} = (\mathcal{D}, S)$ *is defined as:*

- *a* size *function* $\mathcal{D} \longrightarrow \mathbf{N}$*;*
- *a* cost *function: for each* $s \in S$, $\mathrm{cost}(s) : \mathcal{D}^{\mathrm{ar}(s)} \longrightarrow \mathbf{N}$, *where* $\mathrm{ar}(s)$ *denotes the arity of* s*.*

Definition 6.4.3 *Let* $\mathtt{D} = (\mathcal{D}, S)$ *be an abstract data structure with a cost model* (s, c)*. A* QMC $(\alpha, c^\alpha, t^\alpha)$ quantitatively implements $\mathtt{D}$ *with cost model* (s, c) *if:*

- *the interpretation* $\Delta : \mathtt{D} \longrightarrow \mathbf{X}$ *satisfies* $c^\alpha(\Delta(d)) \leqslant \mathrm{size}(d)$*;*
- *for all* $s \in S$*, the program* P_s *implementing* s *is* $\mathrm{cost}(s)$*-bounded. Formally, this is expressed as the fact that*

$$P \in \mathrm{Trans}^{\Delta^{\mathrm{ar}(s)} \to \Delta^{\mathrm{dom}(s)}}_{\alpha, c^\alpha, t^\alpha}(\mathrm{cost}(s)),$$

where $\mathrm{dom}(s) = k$ *means that the codomain of* s *is equal to* D^k*.*

Here again, a thorough study of these notions would be required to establish a number of basic results. I leave this for future work.

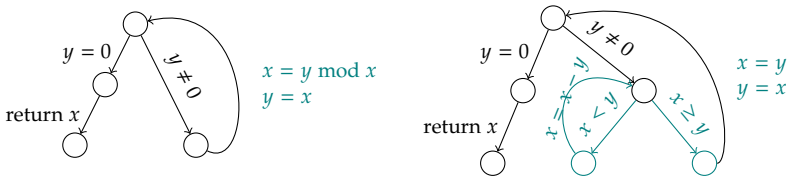
6.5. Examples

Variant of the GCD algorithm

We here discuss examples of algorithms and implementations. We chose to work with the computation of the GCD algorithm. We start by defining two syntactic algorithms, shown in Figure 6.2.

(a) Algorithm A **(b)** Algorithm B

Figure 6.2.: Two syntactical algorithms



(a) Algorithm A **(b)** Algorithm B

Figure 6.3.: Two specified algorithms

The algorithm A is implemented by the following two programs[3]:

```
def gcd1(x, y):
  while(y):
    z = x
    x = y
    y = z % y
  return x
```

```
def f(x, y):
  while(y):
    z = x
    x = y
    y = z - y
  return x
```

This illustrates the limitation of syntactical algorithms: they do not capture operational information about the programs used to interpret labels in the algorithms. Similarly, the following two programs implement the syntactic algorithm B:

```
def gcd2(x, y):
  while(y):
    while x>= y:
      x = x - y
    z = x
    x = y
    y = z
  return x
```

```
def g(x, y):
  while(y):
    while x>= y:
      x = x \ y
    z = x
    x = y
    y = z
  return x
```

As formally states by Proposition 6.4.1, and using the fact that B can be obtained as the glueing of algorithms along A, these two programs also implement the algorithm A. But now one can define specified version of the algorithms A and B. These are shown in Figure 6.3; we leave implicit the formal definition of the data structure, which should be clear (the only elements manipulated are integers here). Now, while `gcd1` still implements the *specified* algorithm A, this is not the case of `f`. Similarly, `gcd2` implements the *specified* algorithms B and A, but `g` does not.

Now, the algorithms in Figure 6.3 can also be considered as *logically specified*, where the data structure is defined as a model of a first order theory of euclidean rings.

Euclid's geometric algorithms

To represent ruler and compass constructions, we consider the space[4] $\mathbf{X} = (2^{\mathbb{R}^2})_0^{\mathbb{N}} \times (\mathbb{R}^2)_0^{\mathbf{N}} \times \{0,1\}^\omega$, i.e. a space of triples $(S_i)_{i=1}^m :: (x_j)_{j=1}^n :: \iota$ consisting of a finite sequence of subsets $(S_i)_{i=1}^m$ of the plane, a finite sequence of points $(x_j)_{j=1}^n$ of the plane, and an infinite input sequence (representing choices made by the environment).

3: We write those in the syntax of Python for simplicity (which defines an abstract program in an adequate AMC), but we note that any notion of abstract program could be used here.

4: Note that instead of $\mathbf{R}$, one could consider any euclidean field here.

We will use the infinite sequence to represent choices. Indeed, at some points in the constructions, a choice needs to be made. For instance. in the construction of an isosceles triangle, one starts from two points and constructs the third vertex of the triangle by intersecting two circles. But two such points exist, and a choice must be made. It seems however that such choices need only be made when the set of options is finite[5]. We will therefore consider chosen a map $\text{choose}(\iota_0 \dots \iota_k)(S)$ which is a function that takes a finite set S and a sequence of elements of $\{0,1\}$ and outputs one elements of S. Moreover, we suppose that the probability of choosing an element a is exactly $\frac{1}{\text{Card}(S)}$. This can be ensured by constructing choose as the function outputted by an automata defined as a binary tree of depth k the smallest integer such that $2^k \geqslant \text{Card}(S)$, in which the last $2^k - \text{Card}(S)$ leaves are identified with the root. One can easily check that, given a sequence of $0,1$ generated by independent equidistributed Bernoulli trials, the probability of reaching one of the remaining $\text{Card}(S)$ leaves is equal to[6] $\frac{1}{\text{Card}(S)}$.

5: This was pointed out to me by Alberto Naibo.

6: This is a result I established in a different context, namely that of proving that Agafonov's theorem holds for (rational) probabilistic selectors [53].

The monoid action is generated by the following maps:

- `exchange`(σ, τ) maps $(S_i)_{i=1}^m :: (x_j)_{j=1}^n :: \iota$ to
$$(S_{\sigma(i)})_{i=1}^m :: (x_{\tau(j)})_{j=1}^n :: \iota$$
- `circle` : maps $(S_i)_{i=1}^m :: (x_j)_{j=1}^n :: \iota$ to
$$(S_i)_{i=1}^m \cdot \{x \in \mathbb{R}^2 \mid d(x, x_n) = d(x_{n-1}, x_n)\} :: (x_j)_{j=1}^n :: \iota$$
- `intersection` : maps $(S_i)_{i=1}^m :: (x_j)_{j=1}^n :: \iota$ to
$$\mapsto (S_i)_{i=1}^m \cdot S_{m-1} \cap S_m :: (x_j)_{j=1}^n :: \iota$$
- `pick` : maps $(S_i)_{i=1}^m :: (x_j)_{j=1}^n :: \iota_0 \dots \iota_k \cdot \iota$ to
$$(S_i)_{i=1}^m :: (x_j)_{j=1}^n \cdot \text{choose}(\iota_0 \dots \iota_k)(S_m) :: \tilde{\iota},$$
which is defined when S_m is finite, and in which k is chosen so that $\text{choose}(\iota_0 \dots \iota_k)(S)$ is well defined.

Differently from what happens with ASM, here we remain faithful to Euclid's algorithmic procedure, as we explicitly have an action corresponding to the construction of the circle. This is possible because actions are made not on a single point, but they are instead *global* ones (made on a whole subspace).

Example 6.5.1 We now explain how this AMC allows to consider an abstract program that constructs the third vertex of an isosceles triangle starting from two points. Denoting τ the permutation over $\{1,2\}$ that exchanges both values, the (straight-line) program Iso is defined as:

$$\left\{ \begin{array}{l} (\texttt{circle}, \textbf{initial}(\text{Iso}) \longrightarrow s_1), \\ (\texttt{exchange}(\text{Id}, \tau), s_1 \longrightarrow s_2), \\ (\texttt{circle}, s_2 \longrightarrow s_3), \\ (\texttt{intersection}, s_3 \longrightarrow s_4), \\ (\texttt{pick}, s_4 \longrightarrow \textbf{terminal}(\text{Iso})) \end{array} \right\}$$

Note that the last instruction will pick one of the two intersection points of the two constructed circles. This choice is determined by the environment, represented as the advice string.

The merge sort algorithm

The merge sort algorithm, and more generally the recursive algorithms, are examples of algorithms which are simultaneously defined with the data structure. I.e. we define the merge sort algorithm as a specified algorithm with respect to a data structure that contains a sorting algorithm.

More precisely, suppose one wants to define the merge sort algorithm. The first thing is to define the data structure. We will require here a structure that allows for working with several lists, and compare elements (say integers). Defining a recursive algorithm then corresponds to the following: one adds to the data structure the corresponding structural map `sort` : $\Delta_{\mathcal{L}} \to \Delta_{\mathcal{L}}$. To ease the argument, we will suppose that there exists a structural map `split` : $\Delta_{\mathcal{L}} \to \Delta_{\mathcal{L}} \times \Delta_{\mathcal{L}}$ defined as $s_0, s_1, \dots, s_k \mapsto (s_0, s_2, s_4, \dots) \times (s_1, s_3, s_5, \dots)$. It should be clear that if such a map is not given as part of the structure, one can define a corresponding algorithme using more elementary operations on list. We can then define the sorting algorithm shown in Figure 6.4. Note that in the figures, we indicate the initial state by an incoming edge and the terminal state by a double circle.

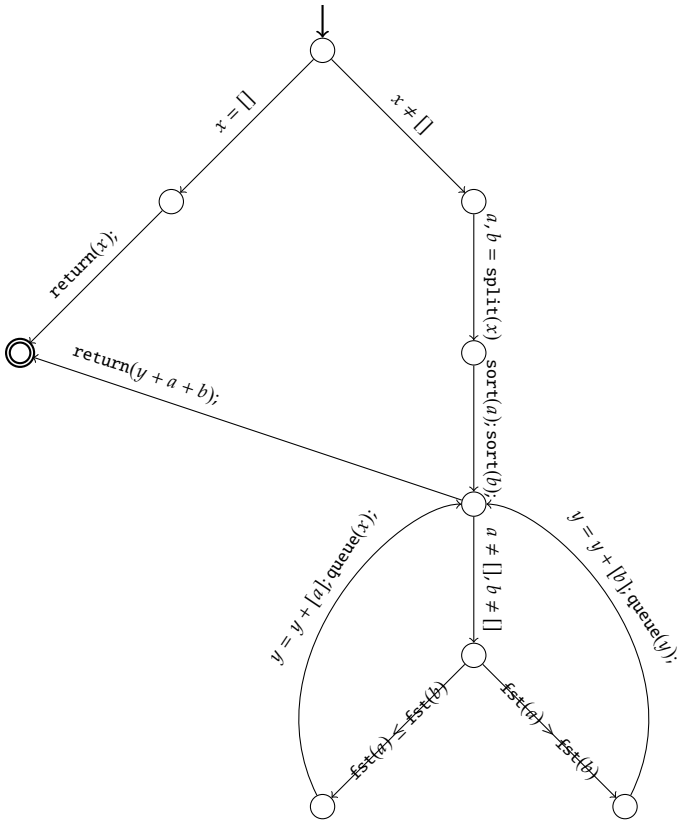


Figure 6.4.: The mergesort algorithm

Now if the `sort` label is interpreted by a program that sorts a list (whichever method is used), a program implementing this algorithm

is itself a program that sorts a list. The recursive definition consists in defining the overall algorithm R obtained by glueing A along the label `sort` within A. This is computing a limit: we start with A, and then obtain $A[\texttt{sort} \leftarrow A]$, in which we then glue A along the label `sort` to obtain $A[\texttt{sort} \leftarrow A][\texttt{sort} \leftarrow A]$, etc.

The recursive program R is then defined formally as the limit of this process, which can be represented as an infinite graph that no longer contains the label `sort`. If one tries to express in more details the recursive structure, we obtain that the obtained algorithm is defined as an implementation of the algorithm shown in Figure 6.5 which put forth the recursive structure, which is close to the recursive definition considered by Moschovakis [104]:

[104]: Moschovakis (2001), *What is an Algorithm?*

$$\text{sort}(x) = \begin{cases} [] & \text{if } x = [] \\ \text{merge}(\text{sort}(\text{split}(x)_1), \text{sort}(\text{split}(x)_2)) & \text{otherwise.} \end{cases}$$

Here the merge algorithm, shown in Figure 6.6, can also be given a recursive definition:

$$\text{merge}(a, b) = \begin{cases} b & \text{if } a = [] \\ a & \text{if } b = [] \\ \text{fst}(a) + \text{merge}(\text{queue}(a), b) & \text{if } \text{fst}(a) \leq \text{fst}(b) \\ \text{fst}(b) + \text{merge}(a, \text{queue}(b)) & \text{otherwise} \end{cases}$$

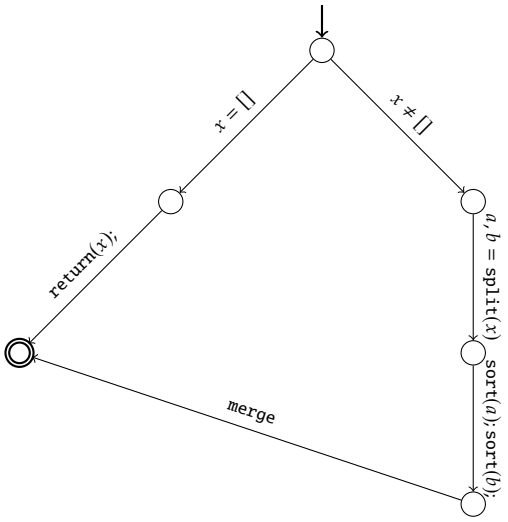


Figure 6.5.: Outer structure of the merge-sort algorithm

Finally, let us note that in the proposed algorithm, no order is enforced on the two subroutines `sort`(a) and `sort`(b): the label simply imposes that both operation are performed in this part of the algorithm. As a consequence, an implementation may choose to sort a before b, or b before a, or even sort both in parallel. Variants of this algorithm in which an order is imposed can be considered by replacing the arrow labelled with both sort operations by two consecutive arrows, one sorting a (for instance), the other sorting b. Another variant in which both operations are performed in parallel may be represented by a single arrow labelled with `sort`(a) | `sort`(b) where | indicates explicitly the parallel execution. These algorithms are refinements of the original one, and any implementation of those is an implementation of the algorithm in Figure 6.4. However they are more specific, and any implementation

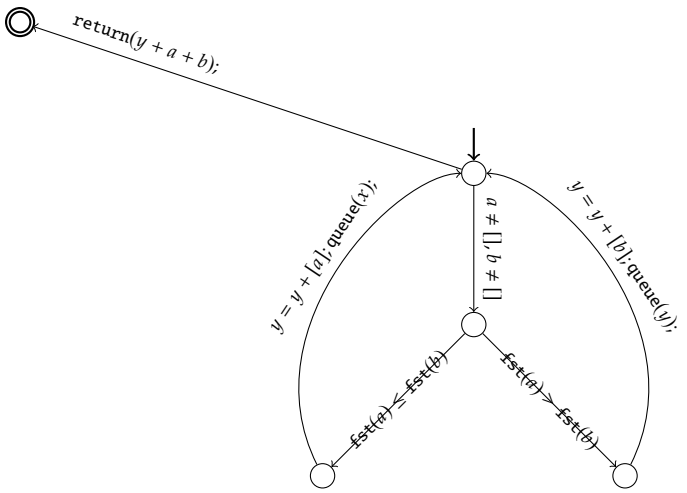


Figure 6.6.: Structure of the merge algorithm

of the original algorithm may not implement one of those, depending on how the subroutine are ordered.

Complexity. What about complexity then? Well, if one defines a cost model for the underlying data structure, we end up writing down an equation. Let us write down the details. We will consider that testing whether a list is empty or not has cost 1. Similarly, concatenation will have cost 1[7] as well the operation of taking the queue of the list (i.e. popping the first element). We are then left with comparisons, splitting, and sorting. Let suppose that splitting has a linear cost in the length of the list and comparison also has cost 1, and let us write the cost of sorting a list of length n as $f(n)$. Now, what is the cost of the overall algorithm?

7: While this is not correct in the Turing machine model, the use of chained list implemented with indirect references (pointers) makes this assumption realistic.

The worst case execution goes through the loop n times, where n is the length of the input list. The algorithm then has its transition complexity bounded by the map $n \mapsto 1 + n + 2 \times f(n/2) + n$. Now, since we identify the overall algorithm with the label `sort`, this imposes the following equation:

$$f(n) \leqslant 1 + 2n + 2.f(\frac{n}{2})$$

This expands as

$$f(n) \leqslant 1 + 2n + 2(1 + 2\frac{n}{2} + 2f(\frac{n}{4})) = 1 + 2n + 2 + 2n + 4f(\frac{n}{4}))$$

then

$$\begin{aligned} f(n) &\leqslant 1 + 2n + 2 + 2n + 4(1 + 2\frac{n}{4} + 2f(\frac{n}{8}))) \\ &\leqslant 1 + 2n + 2 + 2n + 4 + 2n + 4.f(\frac{n}{8}), \end{aligned}$$

and so on.

In the end we get

$$\begin{aligned} f(n) &\leqslant (1 + 2 + \cdots + 2^{\log_2(n)}) + (2n + 2n + \cdots + 2n) \\ &\leqslant 2^{\log_2(n)+1} + 2n \log_2(n) \\ &\leqslant 2n(\log_2(n) + 1). \end{aligned}$$

Part II.

Applications

Discretisation and static analyses of programs

7.

7.1. Flow analyses

The different results presented in this chapter take their origin in the Jones and Kristiansen MWP-flow analysis [2] which computes a polynomial bound – if it exists – on the sizes (of the values) of variables in an imperative `while` programming language, extended with a `loop` operator. This is done by computing for each variable a vector that tracks how it depends on other variables – and the program itself gets assigned a matrix collecting those vectors.

[2]: Jones et al. (2009), *A Flow Calculus of Mwp-bounds for Complexity Analysis*

While this does not ensure termination, it provides a certificate guaranteeing that the program uses throughout its execution at most a polynomial amount of space, and as a consequence that *if* it terminates, it will do so in polynomial time.

Despite this origin, we will here expose how the point of view developed in the first part can be used to recover the dependency analysis used in two of these work (initially obtained as a simplification of the MWP flow approach), through the *discretisation* of abstract programs. Before exposing this (new) material, I start by defining the programming language I will be working with.

A simple While imperative language (with parallel capacities)

I now introduce a simple imperative **`while`** language, with semantics similar to `C`, extended with a parallel command, similar to *e.g.* OpenMP's directives [115], allowing to execute its arguments in parallel.This language supports arrays but not pointers, and we let **`for`** and **`do`**`...`**`while`** loops be represented using **`while`** loops. It is easy to map to fragments of `C`, Java, or any other imperative programming language with parallel support.

[115]: Klemm et al. (2021), *OpenMP Application Programming Interface Specification Version 5.2*

Later sections will consider fragments of this langage. Most will consider only the non-parallel fragment: parallel instructions will only appear in the output of the automatic parallelisation presented in section 7.4, but the analysed programs will all be sequential.

The grammar of this langage is given in Figure 7.1. A variable represents either an undetermined 'primitive' datatype, *e.g.* not a reference variable, or an array, whose indices are given by an expression. We generally use `s`

var ::=	`i` | `j` | ... | `s` | `t` | ... | $\mathtt{x}_1$ | $\mathtt{x}_2$ | ... | $\mathtt{z}_n$ | *var*[*exp*]	(Variables)
exp ::=	*var* | *val* | *op*(*exp*, ..., *exp*)	(Expression)
com ::=	*var* ← *exp* | **`if`** *exp* `then` *com* **`else`** *com* | **`while`** *exp* **`do`** *com* | `use`(*var*, ..., *var*) | `skip` | *com*;*com* | `parallel`{*com*}{*com*} ··· {*com*}	(Command)

Figure 7.1.: A simple imperative **`while`** language

C	out(C)	in(C)	Occ(C) = out(C) ∪ in(C)
x = e	x	Occ(e)	x ∪ Occ(e)
t[e₁] = e₂	t	Occ(e₁) ∪ Occ(e₂)	t ∪ Occ(e₁) ∪ Occ(e₂)
if e then C₁ else C₂	out(C₁) ∪ out(C₂)	Occ(e) ∪ in(C₁) ∪ in(C₂)	Occ(e) ∪ Occ(C₁) ∪ Occ(C₂)
while e do C	out(C)	Occ(e) ∪ in(C)	Occ(e) ∪ Occ(C)
use(x₁, ..., xₙ)	f	{x₁, ..., xₙ}	{x₁, ..., xₙ, f}
skip	∅	∅	∅
C₁; C₂	out(C₁) ∪ out(C₂)	in(C₁) ∪ in(C₂)	Occ(C₁) ∪ Occ(C₂)

Table 7.1.: Definition of out, in and Occ for commands

and t for arrays. An expression is either a variable, a value (*e.g.* integer literal) or the application to expressions of some operator *op*, which can be *e.g.* relational (==, <, *etc.*) or arithmetic (+, -, *etc.*). We let V (resp. e, C) ranges over variables (resp. expression, command) and W range over **while** loops. We also use combined assignment operators and write *e.g.* x++ for x += 1. We assume commands to be correct, *e.g.* with operators correctly applied to expressions, no out-of-bounds errors, *etc.*

A program is thus a sequence of statements, each statement being either an *assignment*, a *conditional*, a *while* loop, a *function call*[1] or a *skip*. *Statements* are abstracted into *commands*, which can be a statement, a sequence of commands, or multiple commands to be run in parallel.

1: The use command represents any command which does not modify its variables but use them and should not be moved around carelessly (*e.g.* a printf). In practice, we currently treat all function calls as use, even if the function is pure.

The semantics of parallel is the following: variables appearing in the arguments are considered local, and the value of a given variable x after execution of the parallel command is the value of the last modified local variable x. This implies possible race conditions, but our transformation (detailed in section 7.4) is robust to those: it assumes given parallel-free programs, and introduces parallel commands that either uniformly update the (copy of the) variables across commands, or update them in only one command.

For convenience we define the following sets of variables which are summarised in Table 7.1. These are only defined for the non-parallel fragment.

Definition 7.1.1 *Given an expression* e, *we define the variables occurring in* e *by:*

$$\mathrm{Occ}(\mathtt{x}) = \mathtt{x} \tag{7.1}$$

$$\mathrm{Occ}(\mathtt{t[e]}) = \mathtt{t} \cup \mathrm{Occ}(\mathtt{e}) \tag{7.2}$$

$$\mathrm{Occ}(\mathrm{val}) = \emptyset \tag{7.3}$$

$$\mathrm{Occ}(\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n)) = \mathrm{Occ}(\mathtt{e}_1) \cup \cdots \cup \mathrm{Occ}(\mathtt{e}_n) \tag{7.4}$$

Definition 7.1.2 *Let* C *be a command, we let* out(C) *(resp.* in(C), Occ(C)*) be the set of variables* modified *by (resp.* used *by,* occurring *in)* C *as defined in Table 7.1. In the* use(x₁, ..., xₙ) *case,* f *is a fresh variable introduced for this command.*

Our treatment of arrays is an over-approximation: we consider the array as a single entity, and that changing one value in it changes it completely. While this may seem a rough approximation, it turns out to be satisfactory for automatic parallelisation: since we do not split loop 'vertically' (*e.g.* distributing the iteration space between threads) but 'horizontally' (*e.g.* distributing the tasks between threads), we want each thread in the parallel command to have control of the array it modifies, and not to have to synchronize its writes with other commands.

7.2. Discretisation of abstract programs

I will now describe how the dependency analysis used in some of this work can be understood as a discretisation of abstract programs. For this, we define an interpretation of the language considered in the previous section. We will later show how it relates to MWP matrices.

Definition 7.2.1 *We define the AMC $\alpha_{\text{MWP}} : M \curvearrowright \mathbf{Z}^{\mathbf{Z}}$ as the monoid action generated by:*

- *arithmetic operations: $+(i,j;k)$ (resp. $\times(i,j;k)$, resp. $C(\lambda;k)$ for $\lambda \in \mathbf{Z}$) act as maps $\alpha_{\text{MWP}}(+(i,j;k)) : (x_p)_{p\in\mathbf{Z}} \mapsto (\bar{x}_p)_{p\in\mathbf{Z}}$ where $\bar{x}_k = x_i + x_j$ (resp. $\bar{x}_k = x_i \times x_j$, resp. $\bar{x}_k = C$) and $\bar{x}_p = x_p$ for $p \neq k$.*
- *copy operation: $\mathsf{copy}(i;k)$ act as maps $\alpha_{\text{MWP}}(\mathsf{copy}(i;k)) : (x_p)_{p\in\mathbf{Z}} \mapsto (\bar{x}_p)_{p\in\mathbf{Z}}$ where $\bar{x}_k = x_i$ and $\bar{x}_p = x_p$ for $p \neq k$.*
- *tests to zero: $\pi(;k)$ the projection of domain $\{(x_p) \mid x_k = 0\}$, and $\bar{\pi}(;k)$ the complement projection on $\{(x_p) \mid x_k \neq 0\}$.*

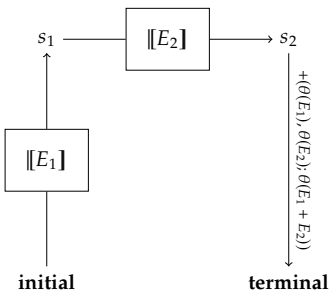
This can be used to interpret programs written in the above language as programs in Programs(α_{MWP}) adequately.

Theorem 7.2.1 *While programs as soundly interpreted are abstract α_{MWP}-programs.*

Proof. The key element here is the translation. For simplicity, we will only treat conditionals and loops; the case of skip and use are not problematic (modulo the introduction of fresh variables for occurrences of use). We will translate each command as an abstract program. The interpretation of a program (a sequence of commands) will then simply correspond to a *composition* of such atomic subprograms by identifying the state **terminal**(C) of the program interpreting a command with the state **initial**(C') of the program interpreting the following command.

Each variable will be assigned a register, i.e. a positive index copy of $\mathbf{R}$ in $\mathbf{R}^{\mathbf{Z}}$. To ease the proof, we consider that variables are of the form X_i, with corresponding index i. Negative index copies of $\mathbf{R}^{\mathbf{Z}}$ will be used for computing expressions. For simplicity, we consider fixed a function assigning a unique natural number $\theta(E)$ to each expression. Expressions are then computed using arithmetic operations; the interpretation $[\![E]\!]$ of an expression is defined inductively as follows:

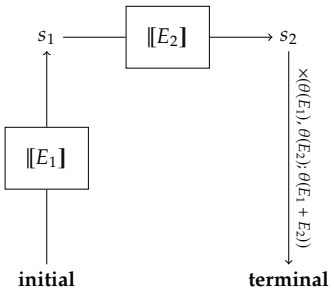
- a variable X_i is represented as the program $[\![X_i]\!]$ with two states and one edge $\mathsf{copy}(i;\theta(X_i))$ from **initial** to **terminal**;
- a value λ is represented as the program $[\![\lambda]\!]$ with two states **initial** and terminal, and one edge $C(\lambda;k)$ from **inital** to terminal;
- an operation $E_1 + E_2$ is represented as the program:



where:

- the initial and terminal states of $[\![E_1]\!]$ are respectively identified with **initial** and s_1;
- the initial and terminal states of $[\![E_2]\!]$ are respectively identified with s_1 and s_2.

- similarly, an operation $E_1 \times E_2$ is represented as the program:

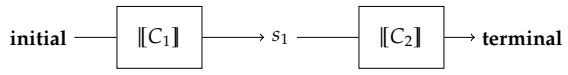


where:

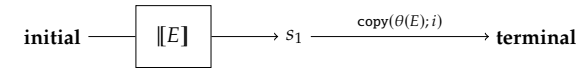
- the initial and terminal states of $[\![E_1]\!]$ are respectively identified with **initial** and s_1;
- the initial and terminal states of $[\![E_2]\!]$ are respectively identified with s_1 and s_2.

Now we can define the interpretation of commands and programs inductively:

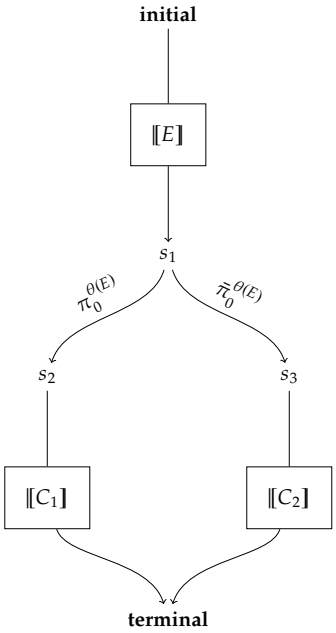
- assignments $X_i \leftarrow E$ are represented as the program obtained from composing the program $[\![E]\!]$ and the program with two states and one edge $\mathtt{copy}(\theta(E); i)$ from **initial** to **terminal** as shown in Figure 7.2a;
- conditionals **`if`** E `then` C_1 **`else`** C_2 are represented as shown in Figure 7.2b where:
  - the initial and terminal states of the program $[\![E]\!]$ computing the expression E are respectively identified with **initial** and s_1;
  - the initial and terminal states of the program $[\![C_1]\!]$ are respectively identified with s_2 and **terminal**;
  - the initial and terminal states of the program $[\![C_2]\!]$ are respectively identified with s_3 and **terminal**.
- loops **`while`** E **`do`** C are interpreted as shown in Figure 7.2c where:
  - the initial and terminal states of the program $[\![E]\!]$ computing the expression E are respectively identified with **initial** and s_1;
  - the initial and terminal states of the program $[\![C]\!]$ are respectively identified with s_2 and **initial**.
- the composition $C_1; C_2$ is interpreted as the program obtained by identifying the initial state of $[\![C_2]\!]$ with the terminal state of $[\![C_1]\!]$:



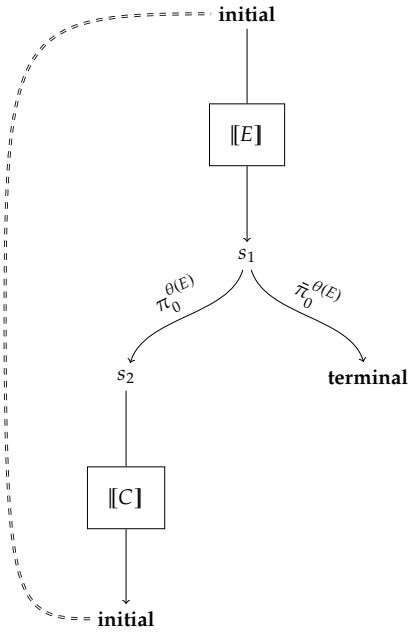
It is not difficult to check that the Interpretation thus defined soundly represents the initial program. ❀



(a) Interpretation of an assignment $X_i \leftarrow E$



(b) Interpretation of a conditional



(c) Interpretation of a loop

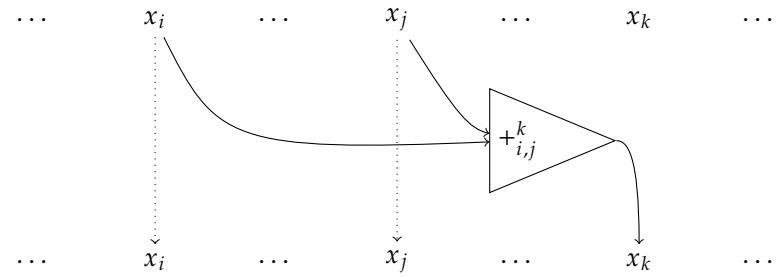
Figure 7.2.: Interpretation of while programs as abstract α_{MWP}-programs

Remark 7.2.1 We note that in the interpretation, we are forced to compute expressions. This is quite realistic, and one aspect of while programs that would otherwise be invisible. This explicitation implies that the interpretation is not quantitatively sound w.r.t. while programs, unless one considers the cost of evaluating an expression as non-trivial (which it is not in practice but is still usually overlooked when considering the running time of a program). I.e. if one considers the time complexity of $C :=$ `if` E `then` $X_0 \leftarrow 1$ `else` $X_0 \leftarrow 0$ to be equal to $\text{time}(C) = 1$, then the interpretation of the program will compute the same result in time that cannot be bounded w.r.t. $\text{time}(C)$ since the length of the orbit will depend on the expression E and could be arbitrarily large. But it could be argued that such a definition of time complexity is not a satisfactory cost model for the while langage.

Let us also note that one very interesting outcome of this explicitation is that it explains and justifies the *conditional* and *loop* correction terms that appear in the dependency (and MWP) analysis. This will be detailed below, when describing the discretisation of the approach.

This straightforward translation associates to each program a graphing, i.e. a complex dynamical system representing the behaviours of this program. We will now explain how this representation can be discretised: instead of considering an action α onto a continuous space $\mathbf{Z}^{\mathbf{Z}}$, we will replace the latter by a discrete space $\{\star\}^{\mathbf{Z}}$. The projection $\mathbf{R} \to \{\star\}$ forgets everything about the values contained in the registers; the maps $\alpha_{\text{MWP}}(+^k_{i,j})$ (resp. $\alpha_{\text{MWP}}(\times^k_{i,j})$) are now trivial; the only information one can keep is the information of dependency between the registers. I.e. the new value of register k depends on the previous value of registers i and j

exclusively, and everything else is preserved. This can be illustrated as:



This representation shows two different kinds of dependencies: the dotted arrows represent the *propagation* of the values of registers (i.e. an unaffected register will keep its value, but note that the register x_k has no propagation arrow since its value is modified), and *proper dependencies* which indicates the value of a register (here x_k) was computed using other values (in this case x_i and x_j). In practice, we are therefore discretising the operations $\alpha_{\text{MWP}}(+^k_{i,j})$ and $\alpha_{\text{MWP}}(\times^k_{i,j})$ as matrices over a simple semiring $\{0, 1, \infty\}$, where 1 (the unit) represents propagation and ∞ represents proper dependency.

Remark 7.2.2 The matrices are infinite since they act on an infinite number of variables, hence one could consider those as sort of *abstract* matrices. However, we note that almost all operations ι in fact corresponds to a Fredholm operator $1+\mathbb{C}(\iota)$: it is the sum of the identity (which propagates the value of all registers) and a compact operator $\mathbb{C}(\iota)$ (involving only the registers concerned by the operation). We will therefore define abusively the interpretation of an instruction ι (and by extension of abstract programs) in α_{MWP} as $\mathbb{C}(\iota)$. The only instructions for which the corresponding matrices are not Fredholm operators are the projections $\pi_0(k)$ and $\bar{\pi}_0(k)$: those are of the form $1 + \mathbb{C}(\pi_0(k))$ and $1 + \mathbb{C}(\bar{\pi}_0(k))$ where $\mathbb{C}(\pi_0(k))$ and $\mathbb{C}(\bar{\pi}_0(k))$ do not even correspond to bounded linear operators. This problem will however not extend to the interpretation of programs in the while langage.

This discretisation performed on each operation in the AMC defined above yields the interpretations shown in Figure 7.3. This extends to the representation of while programs, from which we can deduce a direct interpretation of basic commands. Note that the projections $\pi_0(k)$ and $\bar{\pi}_0(k)$ are always used composed with the body of a conditional or a loop. As a consequence, the overall interpretation of the construct (conditional or loop) remains a Fredholm operator since the projection can be restricted to the domain of the compact operator corresponding to the body.

We recall that Fredholm operators are closed under sums and products. Moreover, we note that composition is quite easily interpreted graphically in this case: the operator C such that $(1 + C_2)(1 + C_1) = 1 + C$ is obtained by collecting paths of length 2 with one edge in the graph of C_1 followed by an edge in the graph of C_2 such that at least one of these edges has weight ∞. This composition, applied to the graphical representation (shown on the left of subfigures in Figure 7.4) of the discretisation of the interpretation of loops and conditionals defined in the proof

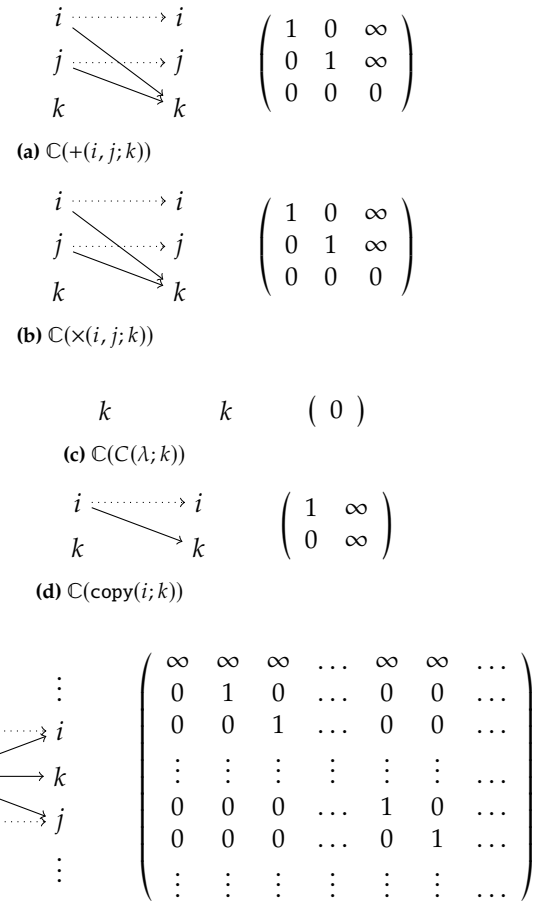


Figure 7.3.: Operators defined by the discretisation of α_{MWP}

of Theorem 7.2.1, yields matricial expressions (shown on the right of subfigures in Figure 7.4).

This discretisation leads us to a dependency analysis that will be formally defined in the next section, and exploited to automatically optimise (section 7.3) and parallelise loops (section 7.4). However, what we juste presented is not the origin story of this dependency analysis; the latter comes from a simplification of Jones and Kristiansen's MWP flow calculus. In a similar manner, MWP flow analysis associates matrices to programs, over a more complex (but still finite) semi-ring $\{0, m, w, p\}$ instead of $\{0, 1, \infty\}$. One interesting aspect of understanding the dependency analysis as a discretisation is the justification for so-called *loop correction* and *conditional correction* terms (here $\mathtt{in}(E)^t\,\mathtt{out}(C)$ and $\mathtt{in}(E)^t\,\mathtt{out}(C+D)$). Those terms appeared previously but the justification was somehow ad-hoc: it was natural to add these dependencies, and adding them made the whole approach work, but no deep understanding of these terms was provided. Here we can trace back their origin in a very natural aspect of conditions: the expressions need to be computed to be evaluated, creating a dependency of every command in the body on the variables involved.

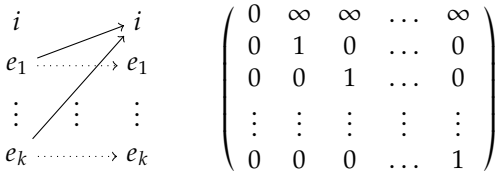
Despite being closely related, the original MWP flow analysis is not obtained directly as a *discretisation* of abstract programs, for two reasons. First, Jones and Kristiansen's calculus is not totally defined: some programs (in particular those not admitting polynomial bounds on the size

of variables) cannot be associated with a MWP matrix. Secondly, the derivation system is non-deterministic: a given program can be associated to multiple distinct MWP matrices.

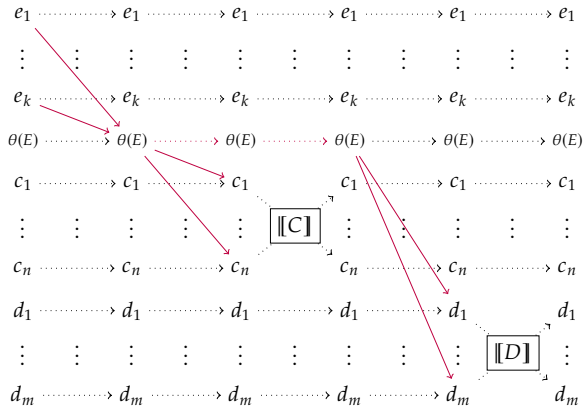
While we will now focus on the dependency analysis, the original MWP flow calculus will be recalled later in this chapter (section 7.5). We will then introduce an alternative system, arguably providing a better approach to MWP flow analysis, which will not suffer from the two drawbacks mentionned in the previous paragraph: the derivation system will be total and deterministic. This new formalism may be obtained as a more refined discretisation of the above representation of while programs as abstract α_{MWP}-programs. The question of how to deduce the MWP analysis in this way will however be left open in this document.

The constructions above provide the definition of *data flow graph* (DFG) of a program P [54], or rather their representation as a matrix $\mathbb{C}(P)$. We refer to the literature for a proper definition independent from the considerations on abstract programs and discretizations.

[54]: Moyen et al. (2017), *Loop Quasi-Invariant Chunk Detection*

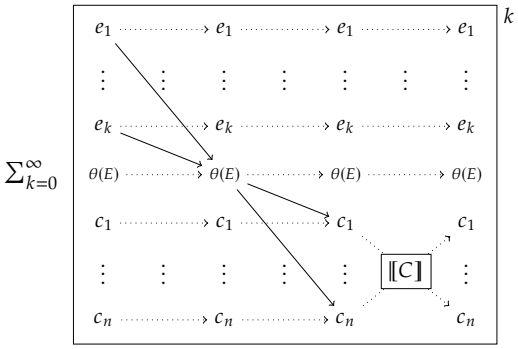


(a) Assignments $X_i \leftarrow E$ ($\mathtt{in}(E) = \{e_1, \ldots, e_k\}$)



$[\![\mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ D]\!] = \mathtt{in}(E)^t(\mathtt{out}(C) + \mathtt{out}(D)) + [\![C]\!] + [\![D]\!]$

(b) Conditional **`if`** E `then` C **`else`** D



$$[\![\mathtt{while}\ E\ \mathtt{do}\ C]\!] = 1 + \mathtt{in}(E)^t(\mathtt{out}(C)) + [\![C]\!] + (\mathtt{in}(E)^t(\mathtt{out}(C)) + [\![C]\!])^2 + \ldots$$
$$= \mathtt{in}(E)^t(\mathtt{out}(C)) + (1 + [\![C]\!] + [\![C]\!]^2 + \ldots)$$

(c) Loop **`while`** E **`do`** C

Figure 7.4.: Compact operators defined by the discretisation of while programs

7.3. Dependency analysis and loop quasi-invariants

Invariance Degree

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

Consider a loop $\mathtt{C} := \mathtt{while\ E\ do}\ [\mathtt{C}_1; \mathtt{C}_2; \ldots, \mathtt{C}_n]$. We will build a *dependence graph* $\mathrm{Dep}(\mathtt{C})$ from the information given by the DFGs.

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

Notations 7.3.1. Given a variable i, we define the set $i^<$ as $\{\mathtt{C}_k \mid i \in \mathrm{Out}(\mathtt{C}_k)\}$, the set of command modifying variable i. Given a command $\mathtt{C}_m$ and a variable $i \in \mathrm{In}(\mathtt{C}_m)$, we denote as $\mathrm{PrD}_i(\mathtt{C}_m)$ the subset of $i^<$ – the set of principal dependences – defined as follows:

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

$$\mathtt{C}_{m-1} <_m \mathtt{C}_{m-2} <_m \cdots <_m \mathtt{C}_1 <_m \mathtt{C}_n <_m \mathtt{C}_{n-1} <_m \cdots <_m \mathtt{C}_m;$$

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

Definition 7.3.1 *Let* $\mathtt{C} := \mathtt{while\ E\ do}\ [\mathtt{C}_1; \mathtt{C}_2; \ldots, \mathtt{C}_n]$ *be a command. We define the directed graph* $\mathrm{Dep}(\mathtt{C})$ *as follows:*

- *the set of vertices* $V^{\mathrm{Dep}(\mathtt{C})}$ *is equal to* $\{\mathtt{C}_1, \ldots, \mathtt{C}_n\}$ *(the set of commands in the loop);*
- *the set of edges* $E^{\mathrm{Dep}(\mathtt{C})}$ *is equal to* $\uplus_{m=1}^{n} \uplus_{i \in \mathrm{In}(\mathtt{C}_m)} \mathrm{PrD}_i(\mathtt{C}_m)$ *(the set of all principal dependencies);*
- *the source* $s(i)$ *of the edge* $\mathtt{C}_k \in \mathrm{PrD}_i(\mathtt{C}_m)$ *is* $\mathtt{C}_k$*;*
- *the target* $t(i)$ *of the edge* $\mathtt{C}_k \in \mathrm{PrD}_i(\mathtt{C}_m)$ *is* $\mathtt{C}_m$*.*

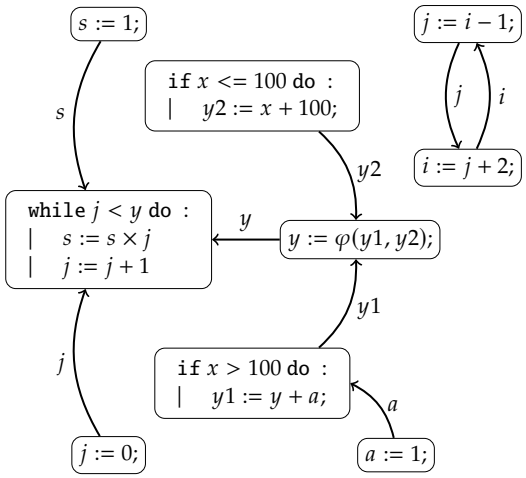
The *invariance degree* $\deg_{\mathtt{C}}(\mathtt{C}_m)$ of a command $\mathtt{C}_m$ w.r.t. $\mathtt{C}$ is then defined as follows. When clear, we will avoid writing the subscript $\mathtt{C}$ to ease notations. If $\mathtt{C}_m$ is a source in $\mathrm{Dep}(\mathtt{C})$, then $\deg(\mathtt{C}_m) =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

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

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

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

(a) An example



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

Figure 7.5.: Exemple of dependency graph

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

Lemma 7.3.2 *Consider the set* $\deg^{-1}(\mathtt{i})$ *for an integer* $i > 0$ *and the relation induced from the dependency graph, i.e.* $\mathtt{C_i} \longrightarrow \mathtt{C_j}$ *if and only if there is a sequence of edges from* $\mathtt{C_i}$ *to* $\mathtt{C_j}$ *in* Dep(C)*. Then* $(\deg^{-1}(\mathtt{i}), \longrightarrow)$ *is a partial order.*

Based on the invariance degree, we will be able to *peel* loops. For this purpose, we define the following notation. Given a sequence of commands $[\mathtt{C_1}; \mathtt{C_2}; \ldots; \mathtt{C_n}]$, we write $[\check{\mathtt{C}}_1; \check{\mathtt{C}}_2; \ldots; \check{\mathtt{C}}_n]^{(i)}$ the subsequence in which all commands of degree strictly less than i are removed. We then define $\mathtt{if}^i = \mathtt{if\ E\ then}\ [\check{\mathtt{C}}_1; \check{\mathtt{C}}_2; \ldots; \check{\mathtt{C}}_n]^{(i)}$, and $\mathtt{while}^i = \mathtt{while\ E\ then}\ [\check{\mathtt{C}}_1; \check{\mathtt{C}}_2; \ldots; \check{\mathtt{C}}_n]^{(i)}$. We can now state the main theorem of the paper, denoting by $[\![\mathtt{C}]\!]$ the semantics of the command C.

Theorem 7.3.3 *Let* $\mathtt{C} := \mathtt{while\ E\ do}\ [\mathtt{C_1}; \mathtt{C_2}; \ldots, \mathtt{C_n}]$ *be a command. Then*

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

In practice: implementations

In the previous section, we have seen that the transformation is possible from and to a While-language. This section will progressively show that we apply do it on real programming languages by introducing two implementations. We first present a *proof of concept*[2] which both analyses and transforms transformation C programs. We will then explain how we implemented a prototype analysis in a mainstream compiler.

2: https://github.com/statycc/LQICM_On_C_Toy_Parser

Proof of concept. To easily and quickly integrate our transformation, we decided to use "pycparser"[3], a C parser written in Python. The principal interest was to simply get and manipulate an Abstract Syntax Tree. Using a "WhileVisitor" we list all nested `while`-loops, then, with a bottom-up

3: https://github.com/eliben/pycparser

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 DFG construction and the invariance degree computation.

The first part aims to list relations between statements. In this implementation we decided to define a relation object by one list of pairs (for the direct dependencies) and two sets (for the propagations and reinitializations) of variables. 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 described previously, we compute the correction and the maximum relations possible for a `while` or `if` statement. With those relations, we compute an invariance degree for each statement in the loop.

This implementation is around 400 lines of `Python`. It is able to compute relations of each commands or sequence of commands. This tool focuses on a restricted `C` syntax and considers all functions as non-pure. Functions with side effects can be seen as an anchor in the sequence of statements, commands can not be moved around them. But we can restrain the conditions for peeling. We can allow to hoist pure functions. All other side effects can be broken by this transformation, and thus should not be moved.

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

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

The main drawback of SSA comes when several different paths in the Control Flow reach the same point (typically, after a conditional). Then, the values used after this point may come from any branch and this cannot be statically decided. For example, if the original program is

```
if (y) then x:=0 else x:=1;C,
```

it is relatively easy to turn it into a pseudo-SSA form by α-converting the `x`:

```
if (y) then x0:=0 else x1:=1;C,
```

but we do not know in `C` which of x_0 or x_1 should be used.

This problem is solved by using φ-*functions*. That is, the correct SSA form will be

```
if (y) then x0:=0 else x1:=1; X:=φ(x0, x1); C.
```

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

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

The implementation[4] (including a preliminary version of peeling) is almost 3000 lines of C++. It is able[5] to compute relations of each commands or sequence of commands, with some restrictions on the form of the loop analyzed. First, loops with several exit blocks are ignored and left intact (typically a loop including a `break`); furthermore, this tool considers all functions as non-pure as for the Proof of Concept. Even with these restrictions, the pass is able to optimise code that was previously left untouched, thus illustrating the power of the method.

4: `https://github.com/ThomasRuby/LQICM_pass`

5: Or rather was able, as it was not updated for compatibility with LLVM updates.

7.4. Dependency analysis and parallelisation

This section now explains how the dependency analysis used in the previous section can also be used to automatically parallelise loops. We will first expose our loop transformation technique, then prove its correctness, and finally present some benchmarks showing how the loop splitting induced by our algorithm outperforms standard automatic tools.

Our algorithm requires essentially to

1. Pick a loop;
2. Compute its condensation graph (Definition 7.4.2) – this requires first the dependence graph (Definition 7.4.1);
3. Compute a covering (Definition 7.4.3) of the condensation graph;
4. Create a loop per element of the covering.

Definition 7.4.1 (Dependence graph) *The* dependence graph *of the loop* $\mathtt{W} \coloneqq \mathtt{while}\ \mathtt{e}\ \mathtt{do}\ \{\mathtt{C}_1;\cdots;\mathtt{C}_n\}$ *is the graph whose vertices is the set of commands* $\{\mathtt{C}_1;\cdots;\mathtt{C}_n\}$*, and there exists a directed edge from* $\mathtt{C}_i$ *to* $\mathtt{C}_j$ *if and only if there exists variables* $\mathtt{x} \in \mathit{out}(\mathtt{C}_j)$ *and* $\mathtt{y} \in \mathit{in}(\mathtt{C}_i)$ *such that* $\mathrm{M}(\mathtt{W})(\mathtt{x},\mathtt{y}) = \infty$.

The remainder of the loop transforming principle is simple: once the graph representing the dependencies between commands is obtained, it remains to determine the cliques in the graph to form *strongly connected components* (sccs) and then separate the sccs into subgraphs to produce the final parallelizable loops that contain a copy of the loop header and update commands.

Definition 7.4.2 *Given the dependence graph of a loop* W,

- *its* strongly connected components *(*sccs*) are its strongly connected subgraphs,*
- *its* condensation graph $G_{\mathtt{W}}$ *is the graph whose vertices are* sccs *and edges are the edges whose source and target belong to distinct* sccs*.*

In our example, the sccs are the nodes themselves, and the condensation graph is

$$\texttt{s1[i] = j*j} \longrightarrow \texttt{i++} \longleftarrow \texttt{s2[i] = 1/j}$$

Excluding the update command `i++`, there are now two nodes in the condensation graph, and we can construct the parallel loops by 1. inserting a `parallel` command, 2. duplicating the loop header and update command, 3. inserting the command in the remaining nodes of the condensation graph in each loop. For our example, we obtain, as expected,

$$\texttt{parallel}\left\{\left|\begin{array}{l}\texttt{while(t[i] != j)\{}\\ \texttt{\ \ s1[i] = j*j;}\\ \texttt{\ \ i++\}}\end{array}\right.\right\}\quad\left\{\left|\begin{array}{l}\texttt{while(t[i] != j)\{}\\ \texttt{\ \ s2[i] = 1/j;}\\ \texttt{\ \ i++\}}\end{array}\right.\right\}.$$

Formally, what we just did was to split the *saturated covering*.

Definition 7.4.3 *A* covering of a graph *G [116] is a collection of subgraphs* $G_1, G_2, \ldots, G_j$ *such that* $G = \cup_{i=1}^{j} G_i$*.*

A saturated covering of *G is a covering* $G_1, G_2, \ldots, G_k$ *such that for all edge in G with source in* G_i*, its target belongs to* G_i *as well. It is* proper *if none of the subgraph is a subgraph of another.*

[116]: Chung (1980), *On the coverings of graphs*

The algorithm then simply consists in finding a proper saturated covering of the loop's condensation graph, and to split the loop accordingly. In our example, the only proper saturated covering is

$$\{\ \texttt{s1[i] = j*j} \longrightarrow \texttt{i++}\ ,\ \texttt{i++} \longleftarrow \texttt{s2[i] = 1/j}\ \}.$$

If the covering was not proper, then the `i++` node on its own would be in it, leading to create a useless loop that performs nothing but updating its own condition.

We now need to prove that the semantics of the initial loop `W` is equal to the semantics of $\tilde{\mathtt{W}}$ given by algorithm 1. This is done by showing that for any variable `x` appearing in `W`, its final value after running `W` is equal to its final value after running $\tilde{\mathtt{W}}$.

Theorem 7.4.1 *The transformation* $\mathtt{W} \rightsquigarrow \tilde{\mathtt{W}}$ *given in algorithm 1 preserves the semantic.*

From the theory to benchmarks

We performed an experimental evaluation of our loop fission technique on a suite of parallel benchmarks. Taking the sequential baseline, we applied the loop fission transformation and parallelization. We compared the result of our technique to the baseline and to an alternative loop fission method implemented in ROSE.

Algorithm 1: Loop splitting algorithm

Data: A loop $\mathtt{W} \coloneqq \mathtt{while\ e\ do}\ \{\mathtt{C}_1; \cdots ; \mathtt{C}_n\}$ (Pick a loop W at top level)
Result: If $k > 1$, $\tilde{\mathtt{W}} \coloneqq \mathtt{parallel}\{\mathtt{W}_1\} \ldots \{\mathtt{W}_k\}$, else $\tilde{\mathtt{W}} \coloneqq \mathtt{W}$

1 Compute the condensation graph G_W of W, (c.f. Definition 7.4.2);
`/* Compute the saturated covering of` G_W `(c.f. Definition 7.4.3) */`
2 **while** *a node n in* G_W *is not part of a subgraph* G_l **do**
3 Create a new subgraph G_i containing n;
4 Recursively add to G_i the nodes targeted by edges whose source is in G_i;
`/* Extract the` *proper* `saturated covering of` G_W `*/`
5 **forall** G_i *in the saturated covering* **do**
6 **if** $\exists G_l$ *in the saturated covering s.t.* G_i *is a subgraph of* G_l **then**
7 remove G_i;
`/* Create one while loop per subgraph in the proper saturated covering */`
8 **forall** G_i *in the proper saturated covering* **do**
9 Let $\mathtt{W}_i \coloneqq$ while e do $\{\mathtt{C}_{i_1}; \cdots ; \mathtt{C}_{i_m}\}$ where $\{\mathtt{C}_{i_1}, \ldots, \mathtt{C}_{i_m}\}$ are the vertices of G_i, inserted in the same order as they are in W

We conducted this experiment in C programming language because it naturally maps to the syntax of the imperative **while** language presented in subsection 7.1. We implement the `parallel` command as OpenMP directives. For instance, the sequential baseline program on the left of Figure 7.6 becomes the parallel version on right[6], after applying our loop fission transformation and parallelization.

6: This example is inspired by benchmark `bicg` from PolyBench/C and presented in our artifact.

The evaluation experimentally substantiated two claims about our technique:

1. It can parallelize loops that are completely ignored by other automatic loop transformation tools, and results in appreciable gain, upper-bounded by the number of parallelizable loops produced by loop fission.
2. Concerning loops that other automatic loop transformation tools can distribute, it yields comparable results in speedup potential. We also demonstrate how insertion of parallel directives can be automated, which supports the practicality of our method.

These results combined confirm that our loop fission technique can easily be integrated into existing tools to improve the performances of the resulting code.

Results In analyzing the results, we distinguish two cases: distributing and parallelizing loops with potentially unknown iterations, and loops with pre-determined iterations (typically **while** and **for** loops, respectively). The difficulty of parallelizing the former arises from the need to synchronize evaluation of the loop recurrence and termination condition. Improper synchronization results in overshooting the iterations [117], rendering such loops effectively sequential.

[117]: Rauchwerger et al. (1995), *Parallelizing While Loops for Multiprocessor Systems*

Loop fission addresses this challenge by recognizing independence between statements and producing parallelizable loops. Special care is

```
j = 0;
while (j<M)
{
  s[j] += r[j]*A[j];
  q[j] += A[j]*p[j];
  j++;
}
```

```
#pragma omp parallel private(j)
{ // Each "pragma" block below
  // have its own copy of j.
  #pragma omp single nowait
  { // "nowait" lets the next
    // block start in parallel.
    j = 0;
    while (j<M) {
      s[j] += r[j]*A[j];
      j++;
    }
  }
  #pragma omp single
  {
    j = 0;
    while (j<M) {
      q[j] += A[j]*p[j];
      j++;
    }
  }
} // Both blocks must be terminated
  // before passing this point.
```

Figure 7.6.: Code transformation example

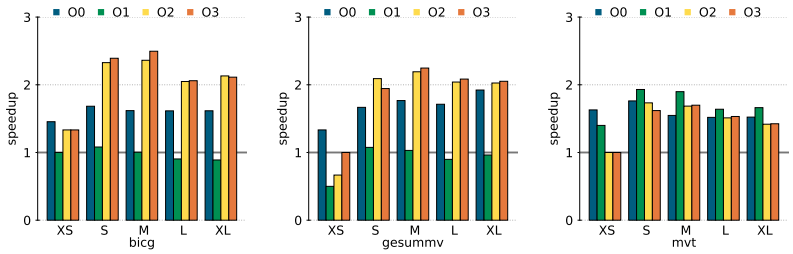


Figure 7.7.: Speedup of selected benchmarks implemented using **`while`** loops. Note the influence of various compiler optimization levels, `-O0` to `-O3` on each problem, and how parallelization overhead tends to decrease as input data size grows from `MINI` to `EXTRALARGE`. The gain is lower for `mvt` because it assumes fissioned form in the original benchmark. `bicg` and `gesummv` obtain higher gain from applied loop distribution.

needed when inserting parallelization directives for such loops. This remains a limitation of automated tools and is not natively supported by OpenMP. We resolved this issue by using the OpenMP `single` directive, to prevent overshooting the loop termination condition and need for synchronization between threads, enabling parallel execution by multiple threads on individual loop statements. The strategy is simple, implementable, and we show it to be effective. However, it is also upper-bounded in speedup potential by the number of parallelizable loops produced by the transformation. This is a syntactic constraint, rather than one based on number of available cores.

The results, presented in Table 7.2, show that our approach, paired with the described parallelization strategy, yields a gain relative to the number of independent parallelizable loops in the transformed benchmark. We observe this *e.g.* for benchmarks `bicg`, `gesummv`, and `mvt`, as presented in Figure 7.7. We also confirm that ROSE's approach did not transform these loops, and report no gain for the alternative approach.

The remaining benchmarks, with known iteration spaces, can be transformed by both evaluated loop fission techniques: ours and ROSE's LoopProcessor. In terms of transformation results, we observed relatively similar results for both techniques. We discovered one interesting transformation difference, with benchmark `gemm`, which ROSE handles differently from our technique.

After transformation, the program must be parallelized by inserting OpenMP directives. This parallelization step can be fully automatic and performed with *e.g.* ROSE or `Clava`, demonstrating that pipelining the transformed programs is feasible. For evaluations, we used manual parallelization for our technique and automatic approach for ROSE. However, we also noted that the automatic insertion of parallelization directives yielded, in some cases, suboptimal choices, such as parallelization of loop nests. This added unnecessary overhead to execution time, and negatively impacted the results obtained for ROSE, *e.g.* for benchmarks `fdtd-2d` and `gemm`, as observable in the results. It is possible this issue could be mitigated by providing annotations and more detailed instructions for applying the parallelization directives. In other experiments with alternative parallelization tools, we have been successful at finding optimal parallelization directives automatically, and therefore conclude it is achievable. We again refer to Table 7.2 for a detailed presentation of the experimental evaluation results.

Benchmark		-O0		-O1		-O2		-O3	
Name	Size	ours	rose	ours	rose	ours	rose	ours	rose
3mm	XS	2.71	0.07	2.26	0.02	1.71	0.02	1.73	0.01
	S	2.80	0.22	3.78	0.09	3.49	0.05	3.35	0.05
	M	2.20	0.46	3.44	0.27	3.08	0.13	3.05	0.13
	L	2.85	1.92	3.11	1.16	2.89	0.66	2.97	0.66
	XL	2.16	2.31	3.13	1.83	2.24	1.05	2.25	1.04
bicg	XS	1.45	0.96	1.00	1.00	1.33	1.00	1.33	1.00
	S	1.68	0.98	1.08	1.00	2.33	1.01	2.39	1.02
	M	1.62	0.97	1.00	0.98	2.36	0.96	2.50	1.00
	L	1.61	0.96	0.90	0.94	2.05	0.95	2.06	0.95
	XL	1.62	0.96	0.89	0.95	2.13	0.93	2.11	0.94
colormap	XS	2.14	1.01	1.50	1.02	1.54	1.04	1.52	1.01
	S	2.08	0.97	1.57	1.00	1.54	1.02	1.43	0.99
	M	1.98	0.95	1.46	0.96	1.49	0.98	1.19	1.00
	L	1.93	1.03	1.42	0.98	1.44	0.98	1.20	1.01
	XL	1.82	1.00	1.53	0.97	1.55	0.99	1.16	1.00
conjgrad	XS	2.43	1.45	1.82	0.69	2.77	0.65	2.50	0.52
	S	2.50	2.39	1.91	2.03	2.84	1.88	2.96	1.65
	M	2.56	2.58	1.94	2.66	2.93	2.44	3.20	2.33
	L	2.38	2.62	1.73	2.96	2.92	2.92	3.24	2.91
	XL	2.29	2.61	1.59	2.55	2.72	2.57	2.99	2.39
cp50	XS	1.90	0.97	1.97	1.00	2.18	1.01	2.09	1.01
	S	1.94	0.95	2.00	1.02	2.08	1.00	2.07	1.00
	M	1.89	0.98	1.76	0.97	1.83	0.99	1.82	0.98
	L	1.74	0.98	1.49	0.96	1.51	0.96	1.50	0.96
	XL	1.63	0.99	1.16	0.96	1.07	0.98	1.11	0.96
deriche	XS	2.00	0.90	1.93	0.51	2.18	0.53	2.11	0.51
	S	2.30	1.49	2.16	1.05	2.17	1.04	2.14	1.03
	M	2.68	2.35	2.88	2.20	2.68	2.22	2.72	2.20
	L	1.79	1.75	2.08	2.03	2.05	2.05	2.07	2.04
	XL	1.12	1.12	1.65	1.61	1.67	1.67	1.60	1.64
fdtd-2d	XS	2.34	0.27	1.48	0.05	1.81	0.06	1.15	0.03
	S	2.57	0.59	2.68	0.15	3.12	0.17	2.47	0.09
	M	2.23	0.82	2.01	0.29	2.47	0.30	2.60	0.24
	L	2.15	1.20	1.89	0.65	1.98	0.61	2.16	0.71
	XL	2.17	1.38	1.47	0.79	1.50	0.73	1.68	0.86
gemm	XS	2.73	0.09	2.33	0.02	2.43	0.02	1.20	0.01
	S	2.87	0.21	3.98	0.05	3.09	0.04	3.01	0.02
	M	2.57	0.56	3.42	0.12	3.40	0.12	2.73	0.05
	L	2.44	1.50	1.79	0.35	1.87	0.36	2.20	0.25
	XL	2.44	1.95	1.85	0.60	1.85	0.70	1.96	0.50
gesummv	XS	1.33	1.00	0.50	0.67	0.67	0.67	1.00	1.00
	S	1.67	0.95	1.08	1.03	2.09	1.03	1.94	1.01
	M	1.77	0.98	1.03	1.00	2.19	1.00	2.25	1.00
	L	1.71	0.94	0.90	0.93	2.04	0.93	2.08	0.97
	XL	1.92	0.98	0.96	0.98	2.03	0.99	2.05	0.98
mvt	XS	1.63	1.00	1.40	0.88	1.00	1.00	1.00	1.00
	S	1.76	1.01	1.93	1.01	1.73	1.02	1.62	1.00
	M	1.55	0.96	1.90	1.00	1.69	1.02	1.70	1.03
	L	1.52	0.98	1.64	0.97	1.51	0.98	1.53	1.00
	XL	1.52	0.98	1.66	0.99	1.42	1.00	1.42	1.00
remap	XS	1.43	0.97	0.54	1.00	0.54	1.00	0.64	1.00
	S	2.07	0.94	1.20	1.02	1.13	1.03	1.19	1.01
	M	2.43	0.99	3.13	0.96	3.36	0.98	2.89	0.97
	L	2.09	1.00	1.34	0.97	1.54	1.02	1.74	1.00
	XL	2.11	1.00	1.28	0.99	1.52	0.99	1.57	1.00
tblshft	XS	3.19	3.27	2.70	2.65	2.68	2.73	2.82	2.82
	S	3.37	3.45	2.82	2.84	2.89	2.86	3.05	3.08
	M	3.31	3.62	2.93	3.00	2.79	2.85	3.21	3.19
	L	3.05	3.40	2.17	2.32	2.38	2.32	2.40	2.39
	XL	3.08	3.48	1.91	1.85	1.64	1.69	1.96	1.96

Table 7.2.: Speedup comparison between original sequential and transformed parallel benchmarks, comparing our loop fission technique with ROSE Compiler, for various data sizes and compiler optimization levels. We note that the problems containing only `while` loop (in **bold**) are not transformed by ROSE and therefore report no gain. The other results vary depending on parallelization strategy, but as noted with *e. g.* problems `conjgrad` and `tblshft`, we obtain similar speedup for both fission strategies when automatic parallelization yields optimal OpenMP directives.

Benchmark	Description	`for` loop	`while` loop	Source
3mm	3D matrix multiplication	✓		PolyBench/C
bicg	BiCG sub kernel of BiCGStab linear solver		✓	PolyBench/C
colormap	TIFF image conversion of photometric palette		✓	MiBench
conjgrad	Conjugate gradient routine	✓		NAS-CG
cp50	Ghostscript/CP50 color print routine	✓	✓	MiBench
deriche	Edge detection filter	✓		PolyBench/C
fdtd-2d	2-D finite different time domain kernel	✓		PolyBench/C
gemm	Matrix-multiply C=alpha.A.B+beta.C	✓		PolyBench/C
gesummv	Scalar, vector and matrix multiplication		✓	PolyBench/C
mvt	Matrix vector product and transpose		✓	PolyBench/C
remap	4D matrix memory remapping	✓		NAS-UA
tblshift	TIFF PixarLog compression main table bit shift	✓	✓	MiBench

Table 7.3.: Descriptions of evaluated parallel benchmarks.

7.5. Implementing MWP

While the dependency analysis used in previous sections was inspired by the MWP flow technique of Jones and Kristiansen [2], it lacks the quantitative aspects of the latter. One natural challenge was thus to implement an MWP static analysis that would allow for deducing polynomial bounds on the growth of variables.

[2]: Jones et al. (2009), *A Flow Calculus of Mwp-bounds for Complexity Analysis*

Let us first recall the original MWP flow calculus of Jones and Kristiansen.

A flow calculus of mwp-bounds for complexity analysis

Flows characterize controls from one variable to another, and can be, in increasing growth rate, of type 0 – the absence of any dependency – *m*aximum, *w*eak polynomial and *p*olynomial. The bounds on programs written in the syntax of (the non-parallel fragment of) subsection 7.1 are represented and calculated thanks to vectors and matrices whose coefficients are elements of the mwp semi-ring.

Definition 7.5.1 (The mwp semi-ring and matrices over it) *Letting* $\text{MWP} = \{0, m, w, p\}$ *with* $0 < m < w < p$, *and* α, β, γ *range over* MWP, *the* mwp semi-ring $(\text{MWP}, 0, m, +, \times)$ *is defined with* $+ = \max$, $\alpha \times \beta = \max(\alpha, \beta)$ *if* $\alpha, \beta \neq 0$, *and* 0 *otherwise.*

We denote $\mathcal{M}\text{at}(\text{MWP})$ *the matrices over* MWP, *and, fixing* $n \in \mathbb{N}$, M *for* $n \times n$ *matrices over* MWP, M_{ij} *for the coefficient in the* i*th row and* j*th column of* M, $\oplus$ *for the componentwise addition, and* $\otimes$ *for the product of matrices defined in a standard way. The* **0***-element for addition is* $\mathbf{0}_{ij} = 0$ *for all* i, j, *and the* **1***-element for product is* $\mathbf{1}_{ii} = m$, $\mathbf{1}_{ij} = 0$ *if* $i \neq j$, *and the resulting structure* $(\mathcal{M}\text{at}(\text{MWP}), \mathbf{0}, \mathbf{1}, \otimes, \oplus)$ *is a semi-ring that we simply write* $\mathcal{M}\text{at}(\text{MWP})$. *The closure operator* $\cdot^*$ *is* $M^* = \mathbf{1} \oplus M \oplus (M^2) \oplus \ldots$, *for* $M^0 = \mathbf{1}$, $M^{m+1} = M \otimes M^m$.

Below, we let V_1, V_2 be column vectors with values in MWP, αV_1 be the usual scalar product, and $V_1 \oplus V_2$ be defined componentwise. We write $\{^{\alpha}_{i}\}$ for the vector with 0 everywhere except for α in its ith row, and $\{^{\alpha}_{i}{}^{\beta}_{j}\}$ for $\{^{\alpha}_{i}\} \oplus \{^{\beta}_{j}\}$.

Replacing in a matrix M the jth column vector by V is denoted $M \overset{j}{\leftarrow} V$. The matrix M with $M_{ij} = \alpha$ and 0 everywhere else is written $\{^{\alpha}_{i} \rightarrow j\}$, and the set of variables in the expression `e` is written Var(`e`). The assumption is made that exactly n different variables are manipulated throughout the analyzed program, so that n-vectors are assigned to expressions – in a non-deterministic way, to capture larger classes of programs [2, Section 8] – and $n \times n$ matrices are assigned to commands using the rules presented Figure 7.8 [2, Section 5].

The intuition is that if $\vdash_{\text{JK}}$ `C` : M can be derived, then all the values computed by `C` will grow at most polynomially w.r.t. its inputs [2, Theorem 5.3], *e.g.* will be bounded by $\max(\vec{x}, p_1(\vec{y})) + p_2(\vec{z})$, where p_1 and p_2 are polynomials and $\vec{x}$ (resp. $\vec{y}$, $\vec{z}$) are m-(resp. w-, p-)annotated variables in the vector for the considered output. Since the derivation system is non-deterministic, multiple matrices and polynomial bounds – that sometimes coincide – may be assigned to the same program. Furthermore, the coefficient at $M_{\texttt{ij}}$ carries quantitative information about the way `Xi`

$$\frac{}{\vdash_{\text{JK}} \texttt{Xi} : \{_i^m\}}\,\text{E1} \qquad \frac{}{\vdash_{\text{JK}} \texttt{e} : \{_i^w \mid \texttt{Xi} \in \text{Var(e)}\}}\,\text{E2}$$

$$\frac{\vdash_{\text{JK}} \texttt{Xi} : V_1 \quad \vdash_{\text{JK}} \texttt{Xj} : V_2 \quad (\star \in \{+,-\})}{\vdash_{\text{JK}} \texttt{Xi} \star \texttt{Xj} : pV_1 \oplus V_2}\,\text{E3} \qquad \frac{\vdash_{\text{JK}} \texttt{Xi} : V_1 \quad \vdash_{\text{JK}} \texttt{Xj} : V_2 \quad (\star \in \{+,-\})}{\vdash_{\text{JK}} \texttt{Xi} \star \texttt{Xj} : V_1 \oplus pV_2}\,\text{E4}$$

(a) Rules for assigning vectors to expressions

$$\frac{\vdash_{\text{JK}} \texttt{e} : V}{\vdash_{\text{JK}} \texttt{Xj} = \texttt{e} : \mathbf{1} \overset{j}{\leftarrow} V}\,\text{A} \qquad \frac{\vdash_{\text{JK}} \texttt{C1} : M_1 \quad \vdash_{\text{JK}} \texttt{C2} : M_2}{\vdash_{\text{JK}} \texttt{C1; C2} : M_1 \otimes M_2}\,\text{C}$$

$$\frac{\vdash_{\text{JK}} \texttt{C1} : M_1 \quad \vdash_{\text{JK}} \texttt{C2} : M_2}{\vdash_{\text{JK}} \texttt{if b then C1 else C2} : M_1 \oplus M_2}\,\text{I} \qquad \frac{\vdash_{\text{JK}} \texttt{C} : M \quad (\forall i, M^*_{ii} = m)}{\vdash_{\text{JK}} \texttt{loop Xl \{C\}} : M^* \oplus \{_{\text{l}}^{p}\!\!\rightarrow j \mid \exists i, M^*_{ij} = p\}}\,\text{L}$$

$$\frac{\vdash_{\text{JK}} \texttt{C} : M \quad (\forall i, M^*_{ii} = m \text{ and } \forall i,j, M^*_{ij} \neq p)}{\vdash_{\text{JK}} \texttt{while b do \{C\}} : M^*}\,\text{W}$$

(b) Rules for assigning matrices to commands

Figure 7.8.: Original non-deterministic ('Jones-Kristiansen') flow analysis rules

depends on `Xj`, knowing that 0- and m-flows are harmless and without constraints, but that w- and p- flows are more harmful w.r.t. polynomial bounds and need to be handled with care, particularly in loops – hence the condition on the L and W rules. The derivation may fail – some programs may not be assigned a matrix – , if at least one of the variables used in the body of a loop depends 'too strongly' upon another, making it impossible to ensure polynomial bounds on the loop itself. We will use the following example as a common basis to discuss possible failure, non-determinism, and our improvements.

Example 7.5.1 Consider `loop X3 {X2 = X1 + X2}`. The body of the `loop` command admits three different derivations, obtained by applying A to one of the three derivation of the expression `X1 + X2`, that we name π_0, π_1 and π_2: From π_0, the derivation of `loop X3 {X2 = X1 + X2}` can be completed using A and L, but since L requires having only m coefficients on the diagonal, π_1 cannot be used to complete the derivation, because of the p coefficient in a box below:

$$\dfrac{\dfrac{\dfrac{\pi_0}{\vdots}}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}}\,\text{A}}{\vdash_{\text{JK}} \texttt{X2 = X1 + X2} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix}}\,\text{L}}{\vdash_{\text{JK}} \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & p & m \end{pmatrix}} \qquad \dfrac{\dfrac{\pi_1}{\vdots}}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}\,\text{A}}{\vdash_{\text{JK}} \texttt{X2 = X1 + X2} : \begin{pmatrix} m & m & 0 \\ 0 & \boxed{p} & 0 \\ 0 & 0 & m \end{pmatrix}}$$

Similarly, using A after π_2 gives a w coefficient on the diagonal and makes it impossible to use L, hence only one derivation for this program exists.

Implementing MWP: challenges

Implementing the MWP analysis however raised two challenges. The first was that the interpretation of programs as flow matrices was only partial: if some variable could not be given a polynomial bound, the derivation system implied that no rules could be applied. This is easily managed

through a modified derivation system and extending the semi-ring with an infinite value.

The second challenge was more complicated: the derivation system proposed by Jones and Kristiansen was non-deterministic. This is an essential property of their approach, as it allows to capture more programs. But the question of whether a given program can be given an mwp bound is strongly non-deterministic and conjectured to be NP-complete. We manage this complication by:

- internalising choices in the semi-ring: using the isomorphism $A \to \mathcal{M}_n(S) \cong \mathcal{M}_n(A \to S)$, we consider matrices over some kinds of *polynomials* over the MWP semiring. A choice (i.e. a non-deterministic branch) then corresponds to an assignment of the choice variables;
- these polynomials can be represented w.r.t. a basis that allows for efficient implementation of basic operations (using techniques inspired from work on Gröbner bases);
- we collect during the analysis the set of impossible assignments for these polynomials, allowing to decide whether a given program has a MWP bound *without computing such a bound explicitly*;
- we implement a specific iterator that allows to produce all MWP bounds in an efficient manner.

While this would not make the problem polynomial (where it NP-complete initially), it opens the question of whether *deciding* the existence of a bound (without computing it) is polynomial. Moreover, we prove the analysis to be compositional, allowing one to compute the possible bounds once and for all, reusing the result in programs calling the program as a subroutine. In practice, this makes the analysis usable: a potential programmer may check the existence of mwp bounds while writing up the code for a given function, and compute the set of all bounds only once the code is finalised; these bounds can then be used in the analysis of other programs calling it without having to recompute the bounds each time.

A deterministic, always-terminating, declension of the mwp analysis

The problem of finding a derivation in the original calculus is in NPTIME [2, Theorem 8.1], and conjectured NPTIME-hard. But since all the non-determinism is in the rules to assigning a vector, the potentially exponential number of derivations are actually extremely similar. Hence, instead of having the analysis stop when failing to establish a derivation and re-starting from scratch, storing the different vectors and constructing the derivation while keeping all the options open seems to be a better strategy, but, as we have seen, this causes a memory blow-up. We address it by fine-tuning the internal machinery: to represent non-determinism, we let the matrices take as values either functions from choices to coefficients in MWP or coefficients in MWP, so that instead of mapping choices to derivations, all the derivations are represented by the same matrix that internalizes the different choices. We discuss this improvement in subsection 7.5, which results in a notable gain: a program involving 6 variables, with 3 choices, would now be assigned a (unique) 6×6 matrix

$$\frac{(\star \in \{+,-\})}{\vdash \texttt{Xi} \star \texttt{Xj} : (0 \mapsto \{{}^{m}_{i}\,{}^{p}_{j}\}) \oplus (1 \mapsto \{{}^{p}_{i}\,{}^{m}_{j}\}) \oplus (2 \mapsto \{{}^{w}_{i}\,{}^{w}_{j}\})}\,\mathrm{E^A}$$

$$\frac{}{\vdash \texttt{Xi} * \texttt{Xj} : \{{}^{w}_{i}\,{}^{w}_{j}\}}\,\mathrm{E^M} \qquad \frac{}{\vdash \texttt{Xi} : \{{}^{m}_{i}\}}\,\mathrm{E^S}$$

(a) Rules for assigning vectors to expressions

$$\frac{\vdash \texttt{e} : V}{\vdash \texttt{Xj} = \texttt{e} : \texttt{1} \overset{j}{\leftarrow} V}\,\mathrm{A} \qquad \frac{\vdash \texttt{C1} : M_1 \quad \vdash \texttt{C2} : M_2}{\vdash \texttt{C1;C2} : M_1 \otimes M_2}\,\mathrm{C} \qquad \frac{\vdash \texttt{C1} : M_1 \quad \vdash \texttt{C2} : M_2}{\vdash \texttt{ifbthenC1elseC2} : M_1 \oplus M_2}\,\mathrm{I}$$

$$\frac{\vdash \texttt{C} : M}{\vdash \texttt{loopX\{C\}} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{p}_{l} \to j \mid \exists i, M^*_{ij} = p\}}\,\mathrm{L^\infty}$$

$$\frac{\vdash \texttt{C} : M}{\vdash \texttt{whilebdo\{C\}} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{\infty}_{i} \to j \mid M^*_{ij} = p\}}\,\mathrm{W^\infty}$$

(b) Rules for assigning matrices to commands

Figure 7.9.: Deterministic improved flow analysis rules

that requires 66 coefficients instead of the 324 we previously had – this is because 30 coefficients are 'simple' values in MWP, and 6 are functions from a set of choices $\{0, 1, 2\}$ to values in MWP, each represented with 6 coefficients.

For the choices that give coefficients fulfilling the side condition of L or W, the derivation can proceed as usual, but when a particular choice gives a coefficient that violates it, we decided against simply removing it. Instead, to guarantee that all derivations always terminate, we mark that choice by indicating that it would not provide a polynomial bound. This requires extending the MWP semi-ring with a special value ∞ that represents failure in a local way, marking non-polynomial flows, and is detailed in subsection 7.5. As a by-product, this enables fine-grained information on programs that *do not* have polynomially bounded growth, since the precise dependencies that break this growth rate can be localized.

Taken together (subsection 7.5), our improvements ensure that exactly one matrix will always be assigned to a program while carrying over the correctness of the original analysis. We give in Figure 7.9 the deterministic system we are introducing in full, but will gently introduce it though the remaining parts of this section: note that the rules A, C and I are unchanged, up to the fact that the matrices, sum and product are in a different semi-ring.

Internalizing non-determinism: the choice data flow semi-rings

Internalizing the choice requires altering the semi-ring used in the analysis: we want to replace the three vectors over MWP that can be assigned to an expression by a single vector over $\{0, 1, 2\} \to$ MWP that captures the same three choices. For a program needing to decide p times between the 3 available choices, this means replacing the $3 \times p$ different matrices in $\mathcal{M}\mathrm{at}(\mathrm{MWP})$ by a single matrix in $\mathcal{M}\mathrm{at}(\{0, 1, 2\}^p \to \mathrm{MWP})$. For any strong semi-ring $\mathbb{S}$ and family of sets $(A_i)_{i=1,\dots,p}$, both $A_i \to \mathbb{S}$ and $\mathcal{M}\mathrm{at}(\prod_{i=1}^{p} A_i \to \mathbb{S})$ are semi-rings, using the usual cartesian product of sets, and there exists an isomorphism $\mathcal{M}\mathrm{at}(\prod_{i=1}^{p} A_i \to \mathbb{S}) \cong \prod_{i=1}^{p} A_i \to \mathcal{M}\mathrm{at}(\mathbb{S})$. This dual nature of the semi-ring considered is useful:

- the analysis will now assign an element M of $\mathcal{M}\text{at}(\prod_{i=1}^{p} A_i \to \text{MWP})$ to a program;
- representing M as an element of $\prod_{i=1}^{p} A_i \to \mathcal{M}\text{at}(\text{MWP})$ allows one to use an *assignment* $\vec{a} = (a_1, \dots, a_p) \in \prod_{i=1}^{p} A_i$ to produce a matrix $M[\vec{a}] \in \mathcal{M}\text{at}(\text{MWP})$, recovering the mwp-flow that would have been computed by making the choices $a_1, \dots, a_p$ in the derivation.

Remark 7.5.1 As the unique degree of non-determinism to assign a matrix to commands is 3, our modification of the analysis flow consists simply of recording the different choices by letting $A_i = \{0, 1, 2\}$ for all $i = 1, \dots, p$ where p is the number of times a choice had to be taken. Starting with subsection 7.5, function calls will require potentially different sets A_i.

Notations 7.5.1. In the following and in the implementation alike, we will denote a function $(a_1^0 \times \cdots \times a_p^0 \mapsto \alpha_0) + \cdots + (a_1^k \times \cdots \times a_p^k \mapsto \alpha_k)$ in $A^p \to \text{MWP}$ with $\text{Card}(A) = k$ by, omitting the product, $(\alpha_0 \delta^0_{a_1^0} \cdots \delta^p_{a_p^0}) + \cdots + (\alpha_k \delta^0_{a_1^k} \cdots \delta^p_{a_p^k})$, with $\delta_i^j = m$ if the jth choice is i, 0 otherwise. Example 7.5.4 will justify and explain this choice.

Our derivation system replaces the E3 and E4 rules with a single rule E^{A} ('additive'), and splits E2 in two exclusive rules, E^{M} for 'multiplicative' and E^{S} for 'simple' (atomic) expressions – Theorem 7.5.3 will prove how they are equivalent.

Example 7.5.2 We represent the three possibles vectors $\begin{pmatrix} p \\ m \\ 0 \end{pmatrix}$, $\begin{pmatrix} m \\ p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$ from Example 7.5.1 with a single vector

$$\begin{pmatrix} p\delta_0^0 + m\delta_1^0 + w\delta_2^0 \\ m\delta_0^0 + p\delta_1^0 + w\delta_2^0 \\ 0 \end{pmatrix},$$

that can be read as

$$\begin{pmatrix} \{0 \mapsto p, 1 \mapsto m, 2 \mapsto w\} \\ \{0 \mapsto m, 1 \mapsto p, 2 \mapsto w\} \\ 0 \end{pmatrix},$$

where we write 0 for $\{0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0\}$[7]. Since in particular[8], $\mathcal{M}\text{at}(\{0, 1, 2\} \to \text{MWP}) \cong \{0, 1, 2\} \to \mathcal{M}\text{at}(\text{MWP})$, the obtained vector can be rewritten as

$$0 \mapsto \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}, 1 \mapsto \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}, 2 \mapsto \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}.$$

7: The implementation supports both coefficients from MWP *and* coefficients from $\{0, 1, 2\}^p \to \text{MWP}$, c.f. *e.g.* a simple assignment example `assign_expression.c`.

8: While the latter lemma applies to algebras of square matrices, a similar result holds for rectangular matrices of a fixed size; the algebraic structure is no longer that of a semi-ring as rectangular matrices do not possess a proper multiplication, but the proof can be adapted to show the existence of an isomorphism of modules between the considered spaces.

Internalizing failure: de-correlating derivations and bounds

The original analysis stops when detecting a non-polynomial flow, puts an end to the chosen strategy (*i.e.* set of choices) and restarts from scratch with another one. We adapt the rules so that every derivation can be completed even in the presence of non-polynomial flows, thanks to a new top element, ∞, representing failure in a local way.

Ignoring our previous modification in this subsection, the semi-ring MWP^∞ we need to consider is $(\text{MWP} \cup \{\infty\}, 0, m, +^\infty, \times^\infty)$, with $\infty > \alpha$ for all $\alpha \in \text{MWP}$, $+^\infty = \max$ as before, and $\alpha \times^\infty \beta = 0$ if $\alpha, \beta \neq \infty$ and α or β is 0, $\max(\alpha, \beta)$ otherwise. This different condition in the definition of $\times^\infty$

ensures that once non-polynomial flows have been detected, they cannot be erased (as $\infty \times^{\infty} 0 = \infty$).

The only cases where the original analysis may fail is if the side conditions of L or W (Figure 7.8) are not met. We replace those by L^{∞} and W^{∞} (Figure 7.9), which replace the problematic coefficients with ∞, marking non-polynomial dependencies, and carry on the analysis.

Example 7.5.3 The program from Example 7.5.1 would now receive three derivations (omitting the one obtained from π_0, as the resulting matrix is identical):

$$\dfrac{\dfrac{\dfrac{\pi_1}{\vdots}}{\vdash \texttt{X1} + \texttt{X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}\,\mathrm{A}}{\vdash \texttt{X2} = \texttt{X1} + \texttt{X2} : \begin{pmatrix} m & m & 0 \\ 0 & p & 0 \\ 0 & 0 & m \end{pmatrix}}\,\mathrm{L}^{\infty}}{\vdash \texttt{loopX3}\{\texttt{X2} = \texttt{X1} + \texttt{X2}\} : \begin{pmatrix} m & p & 0 \\ 0 & \infty & 0 \\ 0 & p & m \end{pmatrix}} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{X1} + \texttt{X2} : \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}}\,\mathrm{E2}}{\vdash \texttt{X2} = \texttt{X1} + \texttt{X2} : \begin{pmatrix} m & w & 0 \\ 0 & w & 0 \\ 0 & 0 & m \end{pmatrix}}\,\mathrm{A}}{\vdash \texttt{loopX3}\{\texttt{X2} = \texttt{X1} + \texttt{X2}\} : \begin{pmatrix} m & w & 0 \\ 0 & \infty & 0 \\ 0 & 0 & m \end{pmatrix}}\,\mathrm{L}^{\infty}$$

Of course, neither of those two derivations would yield polynomial bound – since they contain ∞ coefficients – but it becomes possible to determine that the last one is 'better' – since $\begin{pmatrix} p \\ \infty \\ p \end{pmatrix} > \begin{pmatrix} w \\ \infty \\ 0 \end{pmatrix}$ – and to observe how their 'failure' would propagate in larger programs, possibly establishing that one fares better than the other in terms of non-polynomial growths. This could imply, for instance, that particular programs without polynomial bounds could still be considered 'reasonable' if they are exponential only in some variables that are known to have smaller values in input.

Merging the improvements: illustrations and proofs

We prove that our system captures the original system in the sense that set aside ∞ coefficients, both systems agree (Theorem 7.5.3), but also that exactly one matrix is produced per program (Theorem 7.5.2) – *i.e.* that we can analyze as many programs as originally, and still be correct regarding the bounds. Before doing so, we would like to give more specifics on our system, by combining the semi-rings and intuitions from the previous two subsections. We have discussed our 'axiomatic' (E^A, E^M, E^S) and 'loop' rules (L^{∞} and W^{∞}), but remain to discuss the rules for assignment (A), **if** (I) and composition (C) – which is where both improvements meet. Mathematically speaking, adopting the semi-ring defined over matrices with coefficients in $\{0, 1, 2\}^p \to \text{MWP} \cup \{\infty\}$ is straightforward, and we simply write $\oplus$ and $\otimes$ the operations resulting from merging the two transformations. We discuss in subsection 10 how, however, those operations are computationally costly and how we address this challenge.

Example 7.5.4 Using our deterministic system presented in Figure 7.9, consider the following:

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{X1 + X2} : V}\,\mathrm{E}^{\mathrm{A}}}{\vdash \texttt{X1 = X1 + X2} : 1 \overset{1}{\leftarrow} V}\,\mathrm{A} \qquad \dfrac{\dfrac{}{\vdash \texttt{X1} - \texttt{X3} : V'}\,\mathrm{E}^{\mathrm{A}}}{\vdash \texttt{X1} = \texttt{X1} - \texttt{X3} : 1 \overset{1}{\leftarrow} V'}\,\mathrm{A}}{\vdash \texttt{if b then } \{\texttt{X1} = \texttt{X1} + \texttt{X2}\} \texttt{ else } \{\texttt{X1} = \texttt{X1} - \texttt{X3}\} : (1 \overset{1}{\leftarrow} V) \oplus (1 \overset{1}{\leftarrow} V')}\,\mathrm{I}$$

where

$$V = 0 \mapsto \{{}^{m\ p}_{1\,,2}\} \oplus 1 \mapsto \{{}^{p\ m}_{1,2}\} \oplus 2 \mapsto \{{}^{w\ w}_{1\,,2}\}$$
$$V' = 0 \mapsto \{{}^{m\ p}_{1\,,3}\} \oplus 1 \mapsto \{{}^{p\ m}_{1,3}\} \oplus 2 \mapsto \{{}^{w\ w}_{1\,,3}\}$$
$$1 \overset{1}{\leftarrow} V \cong \begin{pmatrix} (0\mapsto m)\oplus(1\mapsto p)\oplus(2\mapsto w) & 0 & 0 \\ (0\mapsto p)\oplus(1\mapsto m)\oplus(2\mapsto w) & m & 0 \\ 0 & 0 & m \end{pmatrix} = \begin{pmatrix} m\delta_0^0\oplus p\delta_1^0\oplus w\delta_2^0 & 0 & 0 \\ p\delta_0^0\oplus m\delta_1^0\oplus w\delta_2^0 & m & 0 \\ 0 & 0 & m \end{pmatrix}$$
$$1 \overset{1}{\leftarrow} V' \cong \begin{pmatrix} (0\mapsto m)\oplus(1\mapsto p)\oplus(2\mapsto w) & 0 & 0 \\ 0 & m & 0 \\ (0\mapsto p)\oplus(1\mapsto m)\oplus(2\mapsto w) & 0 & m \end{pmatrix} = \begin{pmatrix} m\delta_0^1\oplus p\delta_1^1\oplus w\delta_2^1 & 0 & 0 \\ 0 & m & 0 \\ p\delta_0^1\oplus m\delta_1^1\oplus w\delta_2^1 & 0 & m \end{pmatrix}$$

Some care is needed to perform the addition for the I rule: the choices in the left and right branches are independent, so we must use coefficients in $\{0,1,2\}^2 \to \text{MWP}$ for the 2^3 choices. While the mapping notation would require to use positions to describe which choice is being refereed to, the δ notation makes it immediate, as it encodes in the second value of δ that two choices are considered, numbering the choice in the left branch 0. Hence we can sum the coefficients and obtain the matrix that can be observed in our implementation by analyzing `example7.c`.

Example 7.5.5 Our deterministic system now assigns to `loop X3 {X2 = X1 + X2}` from Example 7.5.1 the unique matrix

$$\begin{pmatrix} m & (0\mapsto p)\oplus(1\mapsto m)\oplus(2\mapsto w) & 0 \\ 0 & (0\mapsto m)\oplus(1\mapsto\infty)\oplus(2\mapsto\infty) & 0 \\ 0 & (0\mapsto p)\oplus(1\mapsto 0)\oplus(2\mapsto 0) & m \end{pmatrix} = \begin{pmatrix} m & p\delta_0^0\oplus m\delta_1^0\oplus w\delta_2^0 & 0 \\ 0 & m\delta_0^0\oplus\infty\delta_1^0\oplus\infty\delta_2^0 & 0 \\ 0 & p\delta_0^0\oplus 0\delta_1^0\oplus 0\delta_2^0 & m \end{pmatrix}$$

where we observe that

1. only one choice, one assignment, 0, gives a matrix without ∞ coefficient, corresponding to the fact that, in the original system, only π_0 could be used to complete the proof,
2. the choice impacts the matrix locally, the coefficients being mostly the same, independently from the choice,
3. the influence of `X2` on itself is where possible non-polynomial growth rates lies, as the ∞ coefficient are in the second column, second row.

We are now in possession of all the material and intuitions needed to state the correspondence between our system and the original one of Jones and Kristiansen.

Theorem 7.5.2 (Determinancy and termination) *Given a program P, there exists unique $p \in \mathbb{N}$ and $M \in \mathcal{M}\text{at}(\{0,1,2\}^p \to \text{MWP}^\infty)$ such that $\vdash P : M$.*

Theorem 7.5.3 (Adequacy) *If $\vdash P : M$, then for all $\vec{a} \in A^p$, $\vdash_{\text{JK}} P : M[\vec{a}]$ iff $\infty \notin M[\vec{a}]$.*

Corollary 7.5.4 (Soundness) *If $\vdash P : M$ and there exists $\vec{a} \in A^p$ such that $\infty \notin M[\vec{a}]$, then every value computed by P is bounded by a polynomial in the inputs.*

This proves that the two analyses coincide, when excluding ∞, and that we can re-use the original proofs of existence of polynomial bounds. However, our alternative definition should be understood as an important improvement, as it enables a better proof-search strategy while optimizing the memory usage, and hence enables the implementation

(subsection 13). It also lets the programmer gain more fine-grained feedback, and illustrates the flexibility of the analysis: the latter will also be demonstrated by the improvements we discuss in the next section.

Extending and improving the analysis: functions and efficiency

To improve this analysis, one could try to extract a tight bound, to certify it, or to port it to a compiler's intermediate representation. Adding constant values is arguably immediate [2, p. 3] but handling pointers, even if technically possible, would probably require significant work. This illustrates at the same time the flexibility of the analysis, and the distance separating ICC-inspired techniques from their usage on actual programs. We decided to narrow this gap along two axes: the first one consists of allowing function definitions and calls in our syntax. It is arguably a small improvement, but illustrates nicely the compositionality of the analysis, and includes recursively defined functions. The second extension intersects the theory and the implementation: it details how our semi-ring structure can be leveraged to maintain a tractable algorithm to compute costly operations on our matrices, and to separate the problem of deciding if a bound exists from computing its form.

Leveraging compositionality to analyze function calls. Thanks to its compositionality, this analysis can easily integrate functions and procedures, by re-using the matrix and choices of a program implementing the function called. We begin by adding to the syntax the possibility of defining multiple functions and calling them:

Definition 7.5.2 (Functions) *Letting* `R` *(resp.* `f`*) range over variables (resp. function names), we add* function calls[9] *to the commands and allow* function declarations*:*

`C` ::= `Xi = f(X1, ..., Xn)` `F` ::= `f(X1, ..., Xn){C; return R}`

In a function declaration, `f(X1, ..., Xn)` *is called the* header*, and the* body *is simply* `C` *(i.e.* `return R` *is not part of the body). A* program *is now a series of function declarations such that all the function calls refer to previously declared functions – we deal with recursive calls below – and a* chunk *is a series of commands.*

9: Function calls that discard the output – procedures – could also be dealt with easily, but are vacuous in our effect-free, in particular pointer-free, language

Now, given a function declaration computing f, we can obtain the matrix M_f by analyzing the body of f as previously done. It is then possible to store the assignments $\vec{a}_0, \ldots, \vec{a}_k$, for which no ∞ coefficients appear[10], and to project the resulting matrices to only keep the vector at `R` that provides quantitative information about all the possible dependencies of the output variable `R` w.r.t. input values, possibly merging choices leading to the same result. After this, we are left with a family $(M_f[\vec{a}_0])|_{\mathsf{R}}, \ldots, (M_f[\vec{a}_k])|_{\mathsf{R}}$ of vectors – as the syntax here is restricted to functions with a single output value, even if accommodating multiple return values would be dealt with the same way – that we can re-use when calling the function.

10: Allowing ∞ coefficients would not change the method described nor its results, but it does not seem relevant to allow calling functions that are not polynomially bounded.

The analysis of the command calling f is then dealt with the F rule below:

$$\frac{}{\vdash \texttt{Xi = F(X1,..., Xn)} : 1 \xleftarrow{\texttt{i}} (((M_f[\vec{a}_0])|_{\texttt{R}})\delta_0^c \oplus \cdots \oplus ((M_f[\vec{a}_k])|_{\texttt{R}})\delta_k^c)}\,\text{F}$$

This rule introduces a choice c over k possible matrices, and it is possible that $k \neq 3$, but this is not an issue, since our semi-ring construction can accommodate any set of choice A.

Example 7.5.6 Consider the following two programs `Q` and `P`:

```
Q=
int f(X1, X2)
{
  while b do {X2=X1+X1};
  return X2;
}
```

```
P =
int foo(X1, X2)
{
  X2=X1+X1;
  X1=f(X2, X2);
}
```

We first have $\vdash$ `X2 = X1 + X1` $: V$ for

$$V = \begin{pmatrix} m & p\delta_0^0 \oplus p\delta_1^0 \oplus w\delta_2^0 \\ 0 & 0 \end{pmatrix},$$

and since $V^* = V$, applying W^∞ gives

$$\vdash \texttt{Q} : \begin{pmatrix} m & \infty\delta_0^0 \oplus \infty\delta_1^0 \oplus w\delta_2^0 \\ 0 & m \end{pmatrix}.$$

Noting that only one choice gives an ∞-free matrix, we can now carry on the analysis of `P`:

$$\frac{\begin{array}{cc}\vdots & \\ \vdash \texttt{X2 = X1 + X1} : V & \vdash \texttt{X1 = f(X2, X2)} : 1 \xleftarrow{1} ((\tbinom{w}{m})\,\delta_0^c)\end{array}\,\text{F}}{\vdash \texttt{P} : V \otimes 1 \xleftarrow{1} ((\tbinom{w}{m})\,\delta_0^c)}\,\text{C}$$

In this particular case, the c choice can be discarded, since only one option is available.

Now, to prove that the F rule faithfully extends the analysis (Theorem 7.5.6), *i.e.* preserves Corollary 7.5.4, we prove that the analysis of the program 'inlining' the function call – as defined below – is, up to some bureaucratic variable manipulation and ignoring some ∞ coefficients, the same as the analysis resulting from using our rule. Intuitively, this mechanism provides the expected result because the choices in the function *do not* affect the program calling it, and because their sets of variables are disjoint – except for the return variable.

Definition 7.5.3 (In-lining function calls) *Let P be a chunk containing a call to the function f, and F be the function declaration computing the function f. The* context $P[\cdot]$*, a chunk containing a slot* $[\cdot]$*, is obtained by replacing in P the function call* `Xi=f(X1, ..., Xn)`*, with* `X'1=X1; ...; X'n=Xn; [·] Xi=R`*, for* `R, X'1, ..., X'n` *fresh variables added to the header containing the chunk.*

The chunk $\tilde{F}$ is obtained from the body of F by renaming the input variables to `X'1, ..., X'n`*, and the variable returned by F to* `R`*. The code $P[F]$ is finally obtained by computing the chunk $\tilde{F}$, and inserting it in place of the symbol* $[\cdot]$ *in $P[\cdot]$.*

That P and $P[F]$ have, at the end of their executions, the same values stored in the variables of P is straightforward in our imperative programming language.

Example 7.5.7 The in-lining of `Q` in `P` from Example 7.5.6 would give the following chunk $\tilde{Q}$ and context `P[·]`, `P[Q]` being obtained by replacing in the latter $[\cdot]$ with the former:

$\tilde{Q} =$
```
while b do {R=X'1+X'1};
```
$P[\cdot] =$
```
int foo(X1, X2, X'1, R)
{
    X2=X1+X1;
    X'1=X2;
    [·]
    X1=R;
}
```

The analysis of `P` (excluding the function call) and `Q` is implemented at `example15a.c`, and of `P[Q]` at `example15b.c`: this latter diverges with Example 7.5.6 only up to projection and ∞-coefficients that are removed by F but not when in-lining the function call.

Now, we need to prove that the matrices $M(P)$ – obtained by analyzing P and using the F rule for `Xi=f(X1, ..., Xn);` – and $M(P[F])$ – obtained by analyzing the inlined $P[F]$ – are the same. However, to avoid conflict with the variables and to project the matrices on the relevant values, some bureaucracy is needed: we write $\Pi_P(M(P[F]))$ (resp. $(1-\Pi_P)(M(P[F]))$) the projection of $M(P[F])$ onto the variables in (resp. *not* in) P. Some non-deterministic choices may appear within the (modified) chunk $\tilde{F}$ inside $P[F]$, *i.e.*

- the coefficients of $M(P)$ are elements of the semi-ring $\prod_{i=1}^{p+1} A_i \to \mathcal{M}\text{at}(\text{MWP})$, with one particular choice corresponding to the F rule – we write the corresponding index i_0;
- the coefficients of $M(P[F])$ are elements of the semi-ring $\prod_{i=1}^{p+k} B_i \to \mathcal{M}\text{at}(\text{MWP})$, where k choices are made within the chunk $\tilde{F}$ – we write the corresponding indexes $j_1, j_2, \ldots, j_k$ (note these are in fact consecutive indexes).

Notations 7.5.5. We note $\pi : \{1, \ldots, p+k\} \to \{1, \ldots, p+1\}$ the projection of the choices in $P[F]$ onto the corresponding choices in P, *i.e.*

$$\pi(j) = \begin{cases} j & \text{if } j < j_1 \\ i_0 & \text{if } j_1 \leqslant j < j_k \\ j-k+1 & \text{if } j_k < j \end{cases}$$

We note that each matrix used as axiom in the function call corresponds to a specific assignment on indexes $j_1, \ldots, j_k$. We write $\Psi : A_{i_0} \to \prod_{i=j_1}^{j_k} B_i$ the corresponding injection, extended to $\bar{\Psi} : \prod_{i=1}^{p+1} A_i \to \prod_{i=0}^{p+k} B_i$ straightforwardly.

Theorem 7.5.6 *For all $\vec{a}$ in $\prod_{i=1}^{p+1} A_i$,*

$$(M(P))[\vec{a}] = (1-\Pi_P)(M(P[F]))[\bar{\Psi}(\vec{a})],$$

and for all $\beta \in \prod_{i=0}^{p+k} B_i$ *which does not belong to the image of* $\bar{\Psi}$,

$$(1 - \Pi_P)(M(P[F])[\beta])$$

contains ∞.

Proof. It is sufficient to prove it for the simplest chunk P containing only one command `Xi = f(X1, ..., Xn)`. This comes from the compositional nature of the analysis, as a sequence of commands is assigned the product of the matrices of each individual command. Then, checking the theorem in this case is a straightforward, though tedious (due to keeping track of all indices), computation. ❀

Integrating recursive calls, the easy way. The question of dealing with self-referential, or recursive, calls, naturally arises when extending to function calls. It turns out that our approach makes such cases easy to handle.

A program implementing a function `rec` calling itself cannot use the F rule presented above as is, since the result of the analysis of `rec` is precisely what we are trying to establish. However, if `rec` takes two input variables `X1` and `X2` and its return value is assigned to a third variable `X3`, then we already know that the vector at 3 will need to be replaced by the vector capturing the dependency between `X1`, `X2`, and the return variable of `rec` (which we will take to be `X3` in our example). The solution consists in replacing the actual values in this vector by variables α, β ranging over values in MWP^∞, terminating the analysis with those variables, and then to resolve the equation – which is easy given the small size of the MWP^∞ semiring.

Example 7.5.8 As an example[11], consider the following program:

11: Where we use variables that are not parameters, and where our recursive call does not terminate: we are focusing on growth rates and not on termination, and keep the example compact.

```
int rec(X1, X2)
{
  X1 = X1 + X2;
  X3 = rec(X1, X2);
  return X3;
}
```

and compute the corresponding matrix:

$$\begin{pmatrix} m\delta_0^0 \oplus p\delta_1^0 \oplus w\delta_2^0 & 0 & 0 \\ p\delta_0^0 \oplus m\delta_1^0 \oplus w\delta_2^0 & m & 0 \\ 0 & 0 & m \end{pmatrix} \otimes 1 \overset{3}{\leftarrow} \begin{pmatrix} \alpha \\ \beta \\ 0 \end{pmatrix} = \begin{pmatrix} m\delta_0^0 \oplus p\delta_1^0 \oplus w\delta_2^0 & 0 & \alpha m\delta_0^0 \oplus \alpha p\delta_1^0 \oplus \alpha w\delta_2^0 \\ p\delta_0^0 \oplus m\delta_1^0 \oplus w\delta_2^0 & m & \alpha p\delta_0^0 \oplus \alpha m\delta_1^0 \oplus \alpha w\delta_2^0 \oplus \beta \\ 0 & 0 & 0 \end{pmatrix}$$

Using the assignments 0, 1 and 2 gives

$$\begin{pmatrix} m & 0 & \alpha m \\ p & m & \alpha p \oplus \beta \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} p & 0 & \alpha p \\ m & m & \alpha m \oplus \beta \\ 0 & 0 & 0 \end{pmatrix}, \text{ and } \begin{pmatrix} w & 0 & \alpha w \\ w & m & \alpha w \oplus \beta \\ 0 & 0 & 0 \end{pmatrix},$$

and since the third vector should be equal to $\begin{pmatrix} \alpha \\ \beta \\ 0 \end{pmatrix}$, this gives three systems of equations:

$$\begin{cases} \alpha m = \alpha \\ \alpha p \oplus \beta = \beta \end{cases} \qquad \begin{cases} \alpha p = \alpha \\ \alpha m \oplus \beta = \beta \end{cases} \qquad \begin{cases} \alpha w = \alpha \\ \alpha w \oplus \beta = \beta \end{cases}$$

The smaller solution to the first (resp. second, third) equational system is $\{\alpha = m; \beta = p\}$ (resp. $\{\alpha = p; \beta = p\}$, $\{\alpha = w; \beta = w\}$), and as a consequence, we find two meaningful solutions (all others being larger than those): $\begin{pmatrix} m \\ p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$.

Taking advantage of polynomial structure to compute efficiently

Ensuring that the analysis is tractable is an important part of our contribution. For a program accepting n different derivations and having k different derivations that cannot be completed, the original flow calculus must run at most $k + 1$ times to find *one* derivation, while our analysis outputs the $k + n$ different derivations in one run, and then sorts them – as discussed next – by listing all the evaluations and looking for ∞ values. In this task, the C rule, that lets building programs from commands, is obviously crucial and consists simply in multiplying two matrices: however, since we are internalizing the choices, those matrices contain a mixture of functions from choices to coefficients in MWP^∞ and of coefficients in MWP. Multiplying such matrices is more costly, but also essential: an 8-line program such as `explosion.c` requires to multiply elements of its matrix 34,992 times[12]. This forces to represent and manipulate the elements of $\prod_{i=1}^{p} A_i \to \mathcal{M}\text{at}(\text{MWP})$ – setting aside ∞ coefficients for a moment – cleverly: simple comparison showed that the improved algorithm presented below made the analysis roughly *five times* faster.

12: The need to optimize functions is made even more obvious when we discuss benchmarking in paragraph 13.

As discussed in Notations 7.5.1, elements of this semi-ring are represented as *polynomials* w.r.t. the generating set given by the functions $\delta_i^j : \prod_{i=1}^{p} A_i \to \text{MWP}$ defined by $\delta_i^j(a_1, \ldots, a_p) = m$ if $a_j = i$ and $\delta_i^j(a_1, \ldots, a_p) = 0$ otherwise, *i.e.* an element of $\prod_{i=1}^{p} A_i \to \text{MWP}$ is represented as a polynomial $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta_{a_{i,j}}^{b_{i,j}}$ with $\alpha_i \in \text{MWP}$.

This basis has an important property: the *monomials* $\alpha_i \prod_{j=1}^{k_i} \delta_{a_{i,j}}^{b_{i,j}}$ in a polynomial can be ordered so that the product with another monomial is ordered, *i.e.* if $\alpha \leqslant \beta$ and both $\alpha \times \gamma$ and $\beta \times \gamma$ are non-zero, then $\alpha \times \gamma \leqslant \beta \times \gamma$. This order is leveraged to obtain efficient algorithms, similar to what is done using Gröbner bases for computation of standard polynomials [118]. For instance, the algorithm for multiplication of polynomials uses this property to compute the product of an ordered polynomial P with $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta_{a_{i,j}}^{b_{i,j}}$:

[118]: Hoeven et al. (2019), *Fast Gröbner basis computation and polynomial reduction for generic bivariate ideals*

1. compute the products $P_i = P \times \alpha_i \prod_{j=1}^{k_i} \delta_{a_{i,j}}^{b_{i,j}}$ for all i;
2. compare and order a list L of all the first elements of those polynomials;
3. append the smallest element to the result and remove it from the corresponding P_i;
4. insert the (new) first element of P_i to the list L if it exists;
5. if L is non-empty, go back to step 3.

When adding or multiplying polynomials, which consist of monomials, we check if a monomial is contained or included by another, and exclude all redundant cases (c.f. `contains` or `includes`). This is also done when

inserting monomials. Thus we keep polynomials free of implementation choices that we would otherwise have to handle during evaluation.

Deciding faster the existence of a bound: delta graphs

Adopting the $\prod_{i=1}^{p} A_i \to \text{MWP}^{\infty}$ semi-ring permits to complete all derivations simultaneously, but remains to determine if there exists an assignment $\vec{a} \in \prod_{i=1}^{p} A_i$ s.t. the resulting matrix is ∞-free, to decide whenever a program accepts a polynomial bound: this is the *evaluation* step. Despite the optimizations detailed above that simplifies the task, this phase remains particularly costly, since the number of assignment grows exponentially w.r.t. the number of choice, which is linear in the number of variables. While this step is necessary (in one form or another) if one wishes to produce the actual mwp matrices certifying polynomial bounds, we implemented a specific data structure to keep track of assignments resulting in ∞ coefficients on the fly, thus allowing the analysis to provide a qualitative answer quickly. This section details how those *delta graphs* allow to immediately determines whenever a polynomial bound exists without having to compute the corresponding matrix, something that was not possible in the original, non-deterministic, calculus.

A delta graph is a graph whose vertices are monomials. The graph is populated during the analysis by adding those monomials that appear with an infinite coefficient – *i.e.* possible choices leading to ∞ in the resulting matrix. This graph is structured in layers: each layer corresponds to the size of the monomials (the number of deltas) it contains. The intuition is that a monomial – or rather a list of deltas $\delta_{\_}$ – defines a subset of the space $\prod_{i=1}^{p} A_i$; the less deltas in the monomial, the greater the subspace represented[13]. As we populate the delta graph, we create edges within a given layer to keep track of differences between monomials: we add an edge labeled i between two monomials if and only if they differ only on one delta $\delta^i_{\_}$ (*i.e.* one is obtained from the other by replacing the first index of $\delta^i_{\_}$). This is used to implement a `fusion` method on delta graphs, which simplifies the structure: as soon as a monomial m in layer n has $\text{Card}(A_i) - 1$ outgoing edges labelled i, we can remove all these monomials and insert a shorter monomial in layer $n-1$, obtained from m by simply removing $\delta^i_{\_}$. This implements the fact that $\sum_{k=0}^{\text{Card}(A_i)-1} m\delta^j_k = m$.

13: Our intuitions here come from the standard topological structure of spaces of infinite sequences, where such a monomial represents a 'cylinder set', *i.e.* an element of the standard basis for open sets.

Now, remember the delta graph represents the subspace of assignments for which an ∞ appears. If at some point the delta graph is completely simplified (*i.e.* 'fusions' to the graph with a unique monomial consisting in an empty list of $\delta_{\_}$), it means the whole space of assignments is represented and no mwp-bounds can be found. On the contrary, if the analysis ends with a delta graph different from the completely simplified one, at least one assignment exists for which no infinite coefficients appear, and therefore at least one mwp-bound exists. This allows one to answer the question 'Is there at least one mwp-bound?' *without actually computing said bounds*. Based on the information collected in the delta graph and the matrix with polynomial coefficients, one can however recover all possible matrix assignments by going through all possible valuations.

This last part is implemented with a specific iterator that leverages the information collected in the delta graph to skip large sets of valuations in a single step. For instance, suppose the monomial δ_1^1 lies in the delta graph – *i.e.* that an infinite coefficient will be reached if the second index is equal to 1. When asked the valuation after $(0, 0, 2, 2)$ (and supposing that $\mathrm{Card}(A_i) = 3$ for all i), our `delta_iterator` will jump directly to $(0, 2, 0, 0)$, skipping all intermediate valuation of the form $(0, 1, a, b)$ in a single step. Similarly, it will jump from $(1, 0, 2, 2)$ to $(1, 2, 0, 0)$, again skipping several valuations at a time, providing a faster analysis. Note that the implementation required care, to correctly jump when given additional informations from the delta graph, *e.g.* to produce $(2, 0, 1, 0)$ as the successor of $(0, 0, 2, 2)$ if δ_0^0, δ_1^1 and δ_0^2 all belong to the delta graph.

Implementing, testing and comparing the analysis

Demonstrating the implementability of the improved and extended mwp-bounds analysis requires an implementation. Our open-source solution, packaged through Python Package Index (PyPI) as `pymwp`, is a standalone command line tool, written in `Python`, that automatically performs growth-rate analysis on programs written in a subset of the `C` programming language. For programs that pass the analysis, it produces a matrix corresponding to the input program and a list of valid derivation choices; and for programs that do not have polynomial bounds, it reports infinity. Our motivation for choosing `C` as the language of analysis resulted from its central role and similarity with the original `while` language. `Python` was an ideal choice for the implementation because of its plasticity, collection of libraries, and because it allowed partial reuse of a previous flow analysis tool [54]. The source code is available on Github, along with an online demo, and detailed documentationdescribing its current supported features and functionality. We now discuss how we tested and assessed it, and how it compares (or, rather *does not* compare) to other similar approaches.

[54]: Moyen et al. (2017), *Loop Quasi-Invariant Chunk Detection*

Experimental evaluation. We allocated extensive focus and effort on testing and profiling our implementation, to ensure the correctness and efficiency of the analysis, and with the terminal objective of obtaining a usable tool. The test suite includes 42 `C` programs, carefully designed to exercise different aspects of the analysis, ranging from basic derivations, to ones producing worst-case behavior (by yielding *e.g.* dense matrices or exponential number of derivations), and classical examples such as computing the greatest common divisor or exponentiation.

We refer to our benchmarks for measured analysis results for each program. The most salient aspect is that our analysis is extremely fast (the time is measured in *milli*seconds) despite important numbers of function calls (in the 10k range, excluding builtin Python language calls, for 10-lines programs). Even examples tailored to stress our implementation cannot make the analysis go over *4 seconds*. We cannot compare our implementation with implementations of the original analysis, since it has never been implemented, and (according to our attempts) cannot be implemented in any realistic manner.

A more complete presentation of the implementation can be accessed in the recent tool paper [56].

[56]: Aubert et al. (2023), *pymwp: A Static Analyzer Determining Polynomial Growth Bounds*

8. Unifying algebraic lower bounds

8.1. Algebraic models of computation

We start by showing how algebraic models of computation can be defined as abstract programs for a well-chosen AMC.

The general arithmetic AMC.

We now define the actions α_{full} and $\alpha_{\mathbf{R}\text{full}}$. Those will capture all arithmetic and algebraic models of computation considered in this chapter, and the main lemma (Lemma 8.2.11) will be stated for this monoid action. All lower bounds results recovered from the literature, as well as the new lower bounds obtained in this work, will be obtained as corollaries of this technical lemma.

As we intend to consider PRAMS at some point, we consider from the beginning the memory of our machines to be separated in two infinite blocks $\mathbf{Z}^\omega$, intended to represent sets of both *shared* and *private* memory cells[1].

1: Obviously, this could be done without any explicit separation of the underlying space, but this will ease the constructions of the next section.

Definition 8.1.1 *The underlying space of α_{full} is $\mathbf{X} = \mathbf{Z}^{\mathbf{Z}} \cong \mathbf{Z}^\omega \times \mathbf{Z}^\omega$. The set of generators is defined by their action on the underlying space, writing $k /\!\!/ n$ the floor $\lfloor k/n \rfloor$ of k/n with the conventions that $k /\!\!/ n = 0$ when $n = 0$ and $\sqrt[n]{k} = 0$ when $k \leqslant 0$:*

- $\texttt{const}_i(c)$ *initialises the register i with the constant $c \in \mathbf{Z}$:*

$$\alpha_{\text{full}}(\texttt{const}_i(c))(\vec{x}) = (\vec{x}\{x_i := c\});$$

- $\star_i(j,k)$ *($\star \in \{+,-,\times,/\!\!/\}$) performs the algebraic operation $\star$ on the values in registers j and k and store the result in register i:*

$$\alpha_{\text{full}}(\star_i(j,k))(\vec{x}) = (\vec{x}\{x_i := x_j \star x_k\});$$

- $\star_i^c(j)$ *($\star \in \{+,-,\times,/\!\!/\}$) performs the algebraic operation $\star$ on the value in register j and the constant $c \in \mathbf{Z}$ and store the result in register i:*

$$\alpha_{\text{full}}(\star_i^c(j))(\vec{x}) = (\vec{x}\{x_i := c \star x_j\});$$

- $\texttt{copy}(i,j)$ *copies the value stored in register j in register i:*

$$\alpha_{\text{full}}(\texttt{copy}(i,j))(\vec{x}) = (\vec{x}\{x_i := x_j\});$$

- $\texttt{copy}(\sharp i, j)$ *copies the value stored in register j in the register whose index is the value stored in register i:*

$$\alpha_{\text{full}}(\texttt{copy}(\sharp i, j))(\vec{x}) = (\vec{x}\{x_{x_i} := x_j\});$$

- $\mathtt{copy}(i, \sharp j)$ *copies the value stored in the register whose index is the value stored in register* j *in register* i*:*

$$\alpha_{\text{full}}(\mathtt{copy}(i, \sharp j))(\vec{x}) = (\vec{x}\{x_i := x_{x_j}\});$$

- $\sqrt[\sharp]{i}(j)$ *computes the floor of the* n*-th root of the value stored in register* j *and store the result in register* i*:*

$$\alpha_{\text{full}}(\sqrt[\sharp]{i}(j))(\vec{x}) = (\vec{x}\{x_i := \sqrt[x_j]{x_j}\}).$$

The general algebraic AMC.

We also define the real-valued equivalent, which will be essential for the proof of lower bounds. The corresponding AMC $\alpha_{\mathbf{R}\text{ram}}$ is defined in the same way than the integer-valued one, but with underlying space $\mathbf{X} = \mathbf{R}^{\mathbf{Z}}$ and with instructions adapted accordingly:

- the division and n-th root operations are the usual operations on the reals;
- the three copy operators are only effective on integers.

Note that we consider the space $\mathbf{R}^{\mathbf{R}}$, i.e. an uncountable number of potential registers. This appears to us as the simplest way to represent the model of real-valued PRAMS which includes indirect addressing. In practise, only a finite number of registers can be accessed during a finite execution (since indexes need to be computed), and therefore a countable number of potential registers would be enough. However this raises the issue of defining the semantics properly: using maps from $\mathbf{R}$ to $\mathbf{N}$ do not work because this creates side-effects giving more expressive power to the machines (e.g. considering that indirect addressing $\mathtt{copy}(\sharp i, j)$ modifies the register of index $\lfloor i \rfloor$ – where $\lfloor \cdot \rfloor$ is the floor function – turns out to provide a way to define euclidean division!). On the other hand, defining a dynamic allocation of register should be possible but would complicate the definitions.

Definition 8.1.2 *The underlying space of* $\alpha_{\mathbf{R}\text{full}}$ *is* $\mathbf{X} = \mathbf{R}^{\mathbf{R}} \cong \mathbf{R}^{\mathbf{R}} \times \mathbf{R}^{\mathbf{R}}$*. The set of generators is defined by their action on the underlying space, with the conventions that* $k/n = 0$ *when* $n = 0$ *and* $\sqrt[n]{k} = 0$ *when* $k \leqslant 0$*:*

- $\mathtt{const}_i(c)$ *initialises the register* i *with the constant* $c \in \mathbf{R}$*:*

$$\alpha_{\mathbf{R}\text{full}}(\mathtt{const}_i(c))(\vec{x}) = (\vec{x}\{x_i := c\});$$

- $\star_i(j, k)$ $(\star \in \{+, -, \times, /\})$ *performs the algebraic operation* $\star$ *on the values in registers* j *and* k *and store the result in register* i*:*

$$\alpha_{\mathbf{R}\text{full}}(\star_i(j, k))(\vec{x}) = (\vec{x}\{x_i := x_j \star x_k\});$$

- $\star_i^c(j)$ $(\star \in \{+, -, \times, /\})$ *performs the algebraic operation* $\star$ *on the value in register* j *and the constant* $c \in \mathbf{R}$ *and store the result in register* i*:*

$$\alpha_{\mathbf{R}\text{full}}(\star_i^c(j))(\vec{x}) = (\vec{x}\{x_i := c \star x_j\});$$

- $\mathtt{copy}(i, j)$ *copies the value stored in register* j *in register* i*:*

$$\alpha_{\mathbf{R}\text{full}}(\mathtt{copy}(i, j))(\vec{x}) = (\vec{x}\{x_i := x_j\});$$

- $\mathsf{copy}(\sharp i, j)$ *copies the value stored in register* j *in the register whose index is the value stored in register* i*:*

$$\alpha_{\mathbf{R}\mathrm{full}}(\mathsf{copy}(\sharp i, j))(\vec{x}) = (\vec{x}\{x_{x_i} := x_j\});$$

- $\mathsf{copy}(i, \sharp j)$ *copies the value stored in the register whose index is the value stored in register* j *in register* i*:*

$$\alpha_{\mathbf{R}\mathrm{full}}(\mathsf{copy}(i, \sharp j))(\vec{x}) = (\vec{x}\{x_i := x_{x_j}\});$$

- $\sqrt[n]{i}(j)$ *computes the* n*-th real root of the value stored in register* j *and store the result in register* i*:*

$$\alpha_{\mathbf{R}\mathrm{full}}(\sqrt[n]{i}(j))(\vec{x}) = (\vec{x}\{x_i := \sqrt[n]{x_j}\}).$$

Relation with standard models

We will now introduce the notion of *quantitative soundness* with respect to a model of computation. This notion will be essential, as it connects the time complexity of programs in the model considered (e.g. PRAMs, algebraic computation trees) with the length of the orbits of the α-program interpreting it.

Quantitative soundness is expressed with respect to a translation of machines as graphings, together with a translation of inputs as points of the configuration space. In the following section, these operations are defined for each model of computation considered in this paper. In all these cases, the representation of inputs is straightforward.

Definition 8.1.3 *Let AMC* $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ *be an abstract model of computation, and* $\mathcal{M}$ *a model of computation. A* translation *of* $\mathcal{M}$ *w.r.t.* α *is a pair of maps* $[\![\cdot]\!]$ *which associate to each program* M *in* $\mathbb{M}$ *computing a decision problem a computational* α*-graphing* $[\![M]\!]$ *and to each input* ι *a point* $[\![\iota]\!]$ *in* $\mathbf{X}$ *identified as a point in* $\mathbf{X} \times \{\mathbf{initial}(M)\}$*.*

Remark 8.1.1 We use here the definition from the original article [47]. This differs from how one could express it using the framework established in Part 1. In particular, we consider here that a graphing computing a decision problem has two final states, an accepting one ($\top$) and a rejecting one ($\bot$). Based on previous chapter, one could instead consider an abstract program (hence with one terminal state) which could output either a 0 or a 1 (hence computing an interpretation of booleans). The next definition would then amount to express that the length of the orbit of $[\![M]\!]$ from $[\![x]\!]$ to the terminal state is equal, up to multiplication by a constant (not depending on x), to the running time of M on input x.

[47]: Seiller et al. (2022), *Unifying lower bounds for algebraic machines, semantically*

Definition 8.1.4 *Let AMC* α *be an abstract model of computation,* $\mathbb{M}$ *a model of computation. The AMC* α *is* quantitatively sound *for* $\mathbb{M}$ *w.r.t. a translation* $[\![\cdot]\!]$ *if for all machine* M *computing a decision problem and input* ι*,* M *accepts* ι *(resp. rejects* ι*) in* k *steps if and only if* $[\![M]\!]^k([\![\iota]\!]) = \top$ *(resp.* $[\![M]\!]^k([\![\iota]\!]) = \bot$*).*

Algebraic computation trees. The first model considered here will be that of *algebraic computation tree* as defined by Ben-Or [119]. Let us note this model refines the *algebraic decision trees* model of Steele and Yao [120], a model of computation consisting in binary trees for which each branching performs a test w.r.t. a polynomial and each leaf is labelled `YES` or `NO`. Algebraic computation trees only allow tests w.r.t. 0, while additional vertices corresponding to algebraic operations can be used to construct polynomials.

[119]: Ben-Or (1983), *Lower Bounds for Algebraic Computation Trees*

[120]: Steele et al. (1982), *Lower bounds for algebraic decision trees*

Definition 8.1.5 *An* algebraic computation tree *[119] on* $\mathbf{R}^n$ *is a binary tree* T *with an function assigning:*

- *to any vertex* v *with only one child (simple vertex) an operational instruction of the form* $f_v = f_{v_i} \star f_{v_j}$, $f_v = c \star f_{v_i}$, *or* $f_v = \sqrt{f_{v_i}}$, *where* $\star \in \{+, -, \times, /\}$, v_i, v_j *are ancestors of* v *and* $c \in \mathbf{R}$ *is a constant;*
- *to any vertex* v *with two children a test instruction of the form* $f_{v_i} \star 0$, *where* $\star \in \{>, =, \geqslant\}$, *and* v_i *is an ancestor of* v *or* $f_{v_i} \in \{x_1, \ldots, x_n\}$;
- *to any leaf an output* `YES` *or* `NO`.

[119]: Ben-Or (1983), *Lower Bounds for Algebraic Computation Trees*

For any algebraic computation tree T, one can define an $\alpha_{\mathbf{R}\text{full}}$ program simulating it. As algebraic computation trees are *trees*, they are in fact represented by treeings, or *loop-free programs* (Definition 4.2.3), i.e. $\alpha_{\mathbf{R}\text{full}}$-programs whose set of control states can be ordered so that any edge in the graphing is strictly increasing on its control states component.

Theorem 8.1.1 *The representation of* ACTs *as* $\alpha_{\mathbf{R}\text{full}}$*-programs is quantitatively sound.*

Algebraic circuits. In order to recover Cucker's proof that $\mathrm{NC}_{\mathbf{R}} \neq \mathrm{P}_{\mathbf{R}}$, we need to introduce the model of *algebraic circuits* which can be represented as $\alpha_{\mathbf{R}\text{full}}$-programs.

Definition 8.1.6 *An algebraic circuit over the reals with inputs in* $\mathbf{R}^n$ *is a finite directed graph whose vertices have labels in* $\mathbf{N} \times \mathbf{N}$, *that satisfies the following conditions:*

- *There are exactly* n *vertices* $v_{0,1}, v_{0,2}, \ldots, v_{0,n}$ *with first index* 0, *and they have no incoming edges;*
- *all the other vertices* $v_{i,j}$ *are of one of the following types:*
  1. *arithmetic vertex: they have an associated arithmetic operation* $\{+, -, \times, /\}$ *and there exist natural numbers* l, k, r, m *with* $l, k < i$ *such that their two incoming edges are of sources* $v_{l,r}$ *and* $v_{k,m}$*;*
  2. *constant vertex: they have an associated real number* y *and no incoming edges;*
  3. *sign vertex: they have a unique incoming edge of source* $v_{k,m}$ *with* $k < i$.

We call depth *of the circuit the largest* m *such that there exist a vertex* $v_{m,r}$, *and* size *of the circuit the total number of vertices. A circuit of depth* d *is* decisional *if there is only one vertex* $v_{d,r}$ *at level* d, *and it is a sign vertex; we call* $v_{d,r}$ *the* end vertex *of the decisional circuit.*

One can define for each algebraic circuit C an $\alpha_{\mathbf{R}\text{full}}$-program such that each step of computation in C is translated as going through a single edge in the corresponding $\alpha_{\mathbf{R}\text{full}}$-program. The following result then follows.

Theorem 8.1.2 *The representation of* ALGCIRC *as $\alpha_{\mathbf{R}\text{full}}$-programs is quantitatively sound.*

Arithmetic RAMs We want to consider arithmetic parallel random access machines, that act not on strings of bits, but on integers. In order to define those properly, we first define the notion of (sequential) arithmetic random access machine (RAM) before considering their parallelisation.

An arithmetic RAM *command* is a pair (ℓ, I) of a *line* $\ell \in \mathbf{N}^\star$ and an *instruction* $\mathtt{I}$ among the following, where $i, j \in \mathbf{N}$, $\star \in \{+, -, \times, /\}$, $c \in \mathbf{Z}$ is a constant and $\ell, \ell' \in \mathbf{N}^\star$ are lines:

$$\begin{array}{c} \mathtt{skip};\ \ \mathtt{X_i} \coloneqq \mathtt{c};\ \ \mathtt{X_i} \coloneqq \mathtt{X_j} \star \mathtt{X_k};\ \ \mathtt{X_i} \coloneqq \mathtt{X_j};\\ \mathtt{X_i} \coloneqq \sharp\mathtt{X_j};\ \ \sharp\mathtt{X_i} \coloneqq \mathtt{X_j};\ \ \mathtt{if}\ \mathtt{X_i} = \mathtt{0}\ \mathtt{goto}\ \ell\ \mathtt{else}\ \ell'. \end{array}$$

An arithmetic RAM *machine* M is then a finite set of commands such that the set of lines is $\{1, 2, \ldots, \mathrm{len}(M)\}$, with $\mathrm{len}(M)$ the *length* of M. We will denote the commands in M by $(i, \mathrm{Inst}_M(i))$, i.e. $\mathrm{Inst}_M(i)$ denotes the line i instruction.

Machines in the arithmetic RAM model can be represented as graphings w.r.t. the action α_{full}. Intuitively the encoding works as follows. The notion of *control state* allows to represent the notion of *line* in the program. Then, the action just defined allows for the representation of all commands but the conditionals. The conditionals are represented as follows: depending on the value of X_i one wants to jumps either to the line ℓ or to the line ℓ'; this is easily modelled by two different edges of respective sources $\mathbb{H}(i) = \{\vec{x} \mid x_i = 0\}$ and $\mathbb{H}(i)^{\mathrm{c}} = \{\vec{x} \mid x_i \neq 0\}$. This interpretation is quantitatively sound.

Theorem 8.1.3 *The representation of arithmetic* RAMS *as α_{full}-programs is quantitatively sound w.r.t. the translation just defined.*

The Crew operation

Based on the notion of arithmetic RAM, we will now consider their parallelisation, namely arithmetic PRAMS. An arithmetic PRAM M is given as a finite sequence of arithmetic RAM machines $M_1, \ldots, M_p$, where p is the number of *processors* of M. Each processor M_i has access to its own, private, set of registers $(\mathtt{X}^i_k)_{k \geqslant 0}$ and a *shared memory* represented as a set of registers $(\mathtt{X}^0_k)_{k \geqslant 0}$.

One has to deal with conflicts when several processors try to access the shared memory simultaneously. We here chose to work with the *Concurrent Read, Exclusive Write* (CREW) discipline: at a given step at which several processors try to write in the shared memory, only the processor with the smallest index will be allowed to do so. In order to model such parallel computations, we abstract the CREW at the level of monoids. For this, we suppose that we have two monoid actions $\mathrm{M}\langle G, \mathrm{R}\rangle \curvearrowright \mathbf{X} \times \mathbf{Y}$ and $\mathrm{M}\langle H, \mathrm{Q}\rangle \curvearrowright \mathbf{X} \times \mathbf{Z}$, where $\mathbf{X}$ represents the shared memory. We then consider the subset $\# \subset G \times H$ of pairs of generators that potentially conflict with one another – the conflict relation.

Definition 8.1.7 (Conflicted sum) *Let* $\mathrm{M}\langle G, \mathrm{R}\rangle$, $\mathrm{M}\langle G', \mathrm{R}'\rangle$ *be two monoids and* $\# \subseteq G \times G'$. *The* conflicted sum of $\mathrm{M}\langle G, \mathrm{R}\rangle$ and $\mathrm{M}\langle G', \mathrm{R}'\rangle$ over $\#$, *noted* $\mathrm{M}\langle G, \mathrm{R}\rangle *_{\#} \mathrm{M}\langle G', \mathrm{R}'\rangle$, *is defined as the monoid with generators* $(\{1\} \times G) \cup (\{2\} \times G')$ *and relations*

$$\begin{aligned}&(\{1\} \times R) \cup (\{2\} \times R') \cup \{(\mathbf{1}, e)\} \cup \{(\mathbf{1}, e')\}\\ &\quad \cup \{\big((1, g)(2, g'), (2, g')(1, g)\big) \mid (g, g') \notin \#\}\end{aligned}$$

where $\mathbf{1}$, e, e' *are the units of* $\mathrm{M}\langle G, \mathrm{R}\rangle *_{\#} \mathrm{M}\langle G', \mathrm{R}'\rangle$, $\mathrm{M}\langle G, \mathrm{R}\rangle$ *and* $\mathrm{M}\langle G', \mathrm{R}'\rangle$ *respectively.*

In the particular case where $\# = (G \times H') \cup (H \times G')$, *with* H, H' *respectively subsets of* G *and* G', *we will write the sum* $\mathrm{M}\langle G, \mathrm{R}\rangle \ {}_{H}*_{H'} \mathrm{M}\langle G', \mathrm{R}'\rangle$.

Remark 8.1.2 When the conflict relation # is empty, this defines the usual direct product of monoids. This corresponds to the case in which no conflicts can arise w.r.t. the shared memory. In other words, the direct product of monoids corresponds to the parallelisation of processes *without shared memory*.

Dually, when the conflict relation is full ($\# = G \times G'$), this defines the free product of the monoids.

Definition 8.1.8 *Let* $\alpha : M \curvearrowright \mathbf{X} \times \mathbf{Y}$ *be a monoid action. We say that an element* $m \in M$ *is* central relatively to α *(or just* central*) if* m *acts as the identity on* $\mathbf{X}$, *i.e.*[2] $\alpha(m); \pi_X = \pi_X$.

Intuitively, central elements are those that will not affect the shared memory. As such, only *non-central elements* require care when putting processes in parallel.

Definition 8.1.9 *Let* $\mathrm{M}\langle G, \mathrm{R}\rangle \curvearrowright \mathbf{X} \times \mathbf{Y}$ *be an* AMC. *We note* Z_α *the set of central elements and* $\bar{Z}_\alpha(G) = \{m \in G \mid m \notin Z_\alpha\}$.

Definition 8.1.10 (The CREW of AMCS) *Let* $\alpha : \mathrm{M}\langle G, \mathrm{R}\rangle \curvearrowright \mathbf{X} \times \mathbf{Y}$ *and* $\beta : \mathrm{M}\langle H, \mathrm{Q}\rangle \curvearrowright \mathbf{X} \times \mathbf{Z}$ *be* AMCS. *We define the* AMC CREW$(\alpha, \beta) : \mathrm{M}\langle G, \mathrm{R}\rangle \ {}_{\bar{Z}_\alpha(G)}*_{\bar{Z}_\beta(G')} \mathrm{M}\langle G', \mathrm{R}'\rangle \curvearrowright \mathbf{X} \times \mathbf{Y} \times \mathbf{Z}$ *by letting* CREW$(\alpha, \beta)(m, m') = \alpha(m) * \beta(m')$ *on elements of* $G \times G'$, *where*[3]:

$$\alpha(m) * \beta(m') = \begin{cases} \Delta_1; [\alpha(m); \pi_Y, \beta(m')]; [\sigma_{X,Y}, \mathrm{Id}Z] & \text{if } m \notin \bar{Z}_\alpha(G), m' \in \bar{Z}_\beta(G'), \\ \Delta_2; [\alpha(m), \beta(m'); \pi_Z] & \text{otherwise,} \end{cases}$$

with $\Delta_i : \mathbf{X} \times \mathbf{Y} \times \mathbf{Z} \to \mathbf{X} \times \mathbf{Y} \times \mathbf{X} \times \mathbf{Z}$ *defined*[4] *as:*

$$\begin{aligned}\Delta_1 &: (x, y, z) \mapsto (x_0, y, x, z)\\ \Delta_2 &: (x, y, z) \mapsto (x, y, x_0, z).\end{aligned}$$

2: Here and in the following, we denote by ; the sequential composition of functions. I.e. $f; g$ denotes what is usually written $g \circ f$.

3: We denote $\sigma_{X,Y} : \mathbf{Y} \times \mathbf{X} \to \mathbf{X} \times \mathbf{Y}$ the map defined as $(y, x) \mapsto (x, y)$.

4: Formally, the definition of Δ_i is parametrised by the choice of a point $x_0 \in \mathbf{X}$, but the map $\alpha(m) * \beta(m')$ does not depend on this choice because of the projections on $\mathbf{Y}$ and $\mathbf{Z}$.

Parallel models

We can now define AMC of arithmetic PRAMS and thus the interpretations of arithmetic PRAMS as abstract programs. For each integer p, we define the AMC CREW$^p(\alpha_{\text{full}})$. This allows the consideration of up to p parallel arithmetic RAMS: the translation of such a RAM with p processors is defined by extending the translation of RAMS by considering a set of states equal

to $L_1 \times L_2 \times \cdots \times L_p$ where for all i the set L_i is the set of lines of the i-th processor.

Now, to deal with arbitrary large arithmetic PRAMs, i.e. with arbitrarily large number of processors, one considers the following AMC defined as a *direct limit*.

Definition 8.1.11 (The AMC of arithmetic PRAMs) *Let $\alpha : M \curvearrowright \mathbf{X} \times \mathbf{X}$ be the AMC α_{full}. The AMC of arithmetic PRAMs is defined as $\alpha_{\text{pram}} = \varinjlim \text{CREW}^k(\alpha)$, where $\text{CREW}^{k-1}(\alpha)$ is identified with a restriction of $\text{CREW}^k(\alpha)$ through the map $\text{CREW}^{k-1}(\alpha)(m_1, \ldots, m_{k-1}) \mapsto \text{CREW}^k(\alpha)(m_1, \ldots, m_{k-1}, 1)$.*

Remark that the underlying space of the PRAM AMC α_{pram} is defined as the union $\cup_{n\in\omega}\mathbf{Z}^\omega \times (\mathbf{Z}^\omega)^n$ which we will write $\mathbf{Z}^\omega \times (\mathbf{Z}^\omega)^{(\omega)}$. In practise a given α_{pram}-program admitting a finite α_{pram} representative will only use elements in $\text{CREW}^p(\alpha_{\text{full}})$, and can therefore be understood as a $\text{CREW}^p(\alpha)$-program.

Theorem 8.1.4 *The representation of arithmetic PRAMs as α_{pram}-programs is quantitatively sound.*

Algebraic PRAMs. These definitions and results stated for integer-valued PRAMs can be adapted to define *algebraic PRAMs* and their translation as $\alpha_{\mathbf{R}\text{full}}$-programs.

An algebraic RAM *command* is a pair (ℓ, I) of a *line* $\ell \in \mathbf{N}^\star$ and an *instruction* $\mathtt{I}$ among the following, where $i, j \in \mathbf{N}$, $\star \in \{+, -, \times, /\}$, $c \in \mathbf{Z}$ is a constant and $\ell, \ell' \in \mathbf{N}^\star$ are lines:

$$\begin{array}{c} \mathtt{skip};\quad \mathtt{X_i} \coloneqq \mathtt{c};\quad \mathtt{X_i} \coloneqq \mathtt{X_j} \star \mathtt{X_k};\quad \mathtt{X_i} \coloneqq \mathtt{X_j};\\ \mathtt{X_i} \coloneqq \sharp\mathtt{X_j};\quad \sharp\mathtt{X_i} \coloneqq \mathtt{X_j};\quad \mathtt{if}\ \mathtt{X_i} = \mathtt{0}\ \mathtt{goto}\ \ell\ \mathtt{else}\ \ell'. \end{array}$$

We consider a restriction for pointers similar to that considered in the case of integer-valued RAMs. An algebraic RAM *machine* M is then a finite set of commands such that the set of lines is $\{1, 2, \ldots, \text{len}(M)\}$, with $\text{len}(M)$ the *length* of M. We will denote the commands in M by $(i, \text{Inst}_M(i))$, i.e. $\text{Inst}_M(i)$ denotes the line i instruction.

An algebraic PRAM M is given as a finite sequence of algebraic RAM machines $M_1, \ldots, M_p$, where p is the number of *processors* of M. Each processor M_i has access to its own, private, set of registers $(\mathtt{X}_k^i)_{k\geqslant 0}$ and a *shared memory* represented as a set of registers $(\mathtt{X}_k^0)_{k\geqslant 0}$. Again, we chose to work with the *Concurrent Read, Exclusive Write* (CREW) discipline as it is well translated through the CREW operation of AMCs.

Definition 8.1.12 (The AMC of algebraic PRAMs) *Let $\alpha : M \curvearrowright \mathbf{X} \times \mathbf{X}$ be the AMC $\alpha_{\mathbf{R}\text{full}}$. The AMC of algebraic PRAMs is defined as*

$$\alpha_{\mathbf{R}\text{pram}} = \varinjlim \text{CREW}^k(\alpha),$$

where $\text{CREW}^{k-1}(\alpha)$ is identified with a restriction of $\text{CREW}^k(\alpha)$ through

$$\text{CREW}^{k-1}(\alpha)(m_1, \ldots, m_{k-1}) \mapsto \text{CREW}^k(\alpha)(m_1, \ldots, m_{k-1}, 1).$$

Then the following results are quite straightforward.

Theorem 8.1.5 *The representation of algebraic* RAMs *as $\alpha_{\mathbf{R}\text{full}}$-programs is quantitatively sound. The representation of algebraic* PRAMs *as $\alpha_{\mathbf{R}\text{pram}}$-programs is quantitatively sound.*

8.2. Entropy and Cells

Topological Entropy

Topological Entropy was introduced in the context of dynamical systems in an attempt to classify the latter w.r.t. conjugacy. The topological entropy of a dynamical system is a value representing the average exponential growth rate of the number of orbit segments distinguishable with a finite (but arbitrarily fine) precision. The definition is based on the notion of open covers.

Open covers. Given a topological space $\mathbf{X}$, an *open cover* of $\mathbf{X}$ is a family $\mathcal{U} = (U_i)_{i\in I}$ of open subsets of $\mathbf{X}$ such that $\cup_{i\in I} U_i = \mathbf{X}$. A finite cover $\mathcal{U}$ is a cover whose indexing set is finite. A *subcover* of a cover $\mathcal{U} = (U_i)_{i\in I}$ is a sub-family $\mathcal{S} = (U_j)_{j\in J}$ for $J \subseteq I$ such that $\mathcal{S}$ is a cover, i.e. such that $\cup_{j\in J} U_j = \mathbf{X}$.

We will denote by $\mathrm{Cov}(\mathbf{X})$ (resp. $\mathrm{FCov}(\mathbf{X})$) the set of all open covers (resp. all finite open covers) of the space $\mathbf{X}$.

We now define two operations on open covers that are essential to the definition of entropy. An open cover $\mathcal{U} = (U_i)_{i\in I}$, together with a continuous function $f : \mathbf{X} \to \mathbf{X}$, defines the inverse image open cover $f^{-1}(\mathcal{U}) = (f^{-1}(U_i))_{i\in I}$. Note that if $\mathcal{U}$ is finite, $f^{-1}(\mathcal{U})$ is finite as well. Given two open covers $\mathcal{U} = (U_i)_{i\in I}$ and $\mathcal{V} = (V_j)_{j\in J}$, we define their join $\mathcal{U} \vee \mathcal{V}$ as the family $(U_i \cap V_j)_{(i,j)\in I\times J}$. Once again, if both initial covers are finite, their join is finite.

Entropy. Usually, entropy is defined for continuous maps on a compact set, following the original definition by Adler, Konheim and McAndrews [121]. Using the fact that arbitrary open covers have a finite subcover, this allows one to ensure that the smallest subcover of any cover is finite. I.e. given an arbitrary cover $\mathcal{U}$, one can consider the smallest – in terms of cardinality – subcover $\mathcal{S}$ and associate to $\mathcal{U}$ the finite quantity $\log_2(\mathrm{Card}(\mathcal{S}))$. This quantity, obviously, need not be finite in the general case of an arbitrary cover on a non-compact set.

[121]: Adler et al. (1965), *Topological Entropy*

However, a generalisation of entropy to non-compact sets can easily be defined by restricting the usual definition to *finite* covers[5]. This is the definition we will use here.

5: This is discussed by Hofer [122] together with another generalisation based on the Stone-Čech compactification of the underlying space.

Definition 8.2.1 *Let $\mathbf{X}$ be a topological space, and $\mathcal{U} = (U_i)_{i\in I}$ be a finite cover of $\mathbf{X}$. We define the quantity $H^0_{\mathbf{X}}(\mathcal{U})$ as*

$$\min\{\log_2(Card(J)) \mid J \subset I, \cup_{j\in J} U_j = \mathbf{X}\}.$$

In other words, if k is the cardinality of the smallest subcover of $\mathcal{U}$, $H^0(\mathcal{U}) = \log_2(k)$.

Definition 8.2.2 *Let* $\mathbf{X}$ *be a topological space and* $f : \mathbf{X} \to \mathbf{X}$ *be a continuous map. For any finite open cover* $\mathcal{U}$ *of* $\mathbf{X}$*, we define:*

$$H^k_{\mathbf{X}}(f, \mathcal{U}) = \frac{1}{k} H^0_{\mathbf{X}}(\mathcal{U} \vee f^{-1}(\mathcal{U}) \vee \cdots \vee f^{-(k-1)}(\mathcal{U})).$$

One can show that the limit $\lim_{n\to\infty} H^n_{\mathbf{X}}(f, \mathcal{U})$ exists and is finite; it will be noted $h(f, \mathcal{U})$. The topological entropy of f is then defined as the supremum of these values, when $\mathcal{U}$ ranges over the set of all finite covers $\mathrm{FCov}(\mathbf{X})$.

Definition 8.2.3 *Let* $\mathbf{X}$ *be a topological space and* $f : \mathbf{X} \to \mathbf{X}$ *be a continuous map. The* topological entropy *of* f *is defined as* $h(f) = \sup_{\mathcal{U} \in \mathrm{FCov}(\mathbf{X})} h(f, \mathcal{U})$.

Graphings and Entropy

We now need to define the entropy of a *deterministic graphing*. As mentioned briefly already, deterministic graphings on a space $\mathbf{X}$ are in one-to-one correspondence with partial dynamical systems on $\mathbf{X}$. To convince oneself of this, it suffices to notice that any partial dynamical system can be represented as a graphing with a single edge, and that if the graphing G is deterministic its edges can be glued together to define a partial continuous function $[G]$. Thus, we only need to extend the notion of entropy to partial maps, and we can then define the entropy of a graphing G as the entropy of its corresponding map $[G]$.

Given a finite cover $\mathcal{U}$, the only issue with partial continuous maps is that $f^{-1}(\mathcal{U})$ is not in general a cover. Indeed, $\{f^{-1}(U) \mid U \in \mathcal{U}\}$ is a family of open sets by continuity of f but the union $\cup_{U\in\mathcal{U}} f^{-1}(U)$ is a strict subspace of $\mathbf{X}$ (namely, the domain of f). It turns out the solution to this problem is quite simple: we notice that $f^{-1}(\mathcal{U})$ is a cover of $f^{-1}(\mathbf{X})$ and now work with covers of subspaces of $\mathbf{X}$. Indeed, $\mathcal{U} \vee f^{-1}(\mathcal{U})$ is itself a cover of $f^{-1}(\mathbf{X})$ and therefore the quantity $H^2_{\mathbf{X}}(f, \mathcal{U})$ can be defined as $(1/2)H^0_{f^{-1}(\mathbf{X})}(\mathcal{U} \vee f^{-1}(\mathcal{U}))$.

We now generalise this definition to arbitrary iterations of f by extending Definitions 8.2.2 and 8.2.3 to partial maps as follows.

Definition 8.2.4 *Let* $\mathbf{X}$ *be a topological space and* $f : \mathbf{X} \to \mathbf{X}$ *be a continuous partial map. For any finite open cover* $\mathcal{U}$ *of* $\mathbf{X}$*, we define:*

$$H^k_{\mathbf{X}}(f, \mathcal{U}) = \frac{1}{k} H^0_{f^{-k+1}(\mathbf{X})}(\mathcal{U} \vee f^{-1}(\mathcal{U}) \vee \cdots \vee f^{-(k-1)}(\mathcal{U})).$$

The entropy *of* f *is then defined as* $h(f) = \sup_{\mathcal{U} \in \mathrm{FCov}(\mathbf{X})} h(f, \mathcal{U})$*, where* $h(f, \mathcal{U})$ *is again defined as the limit* $\lim_{n\to\infty} H^n_{\mathbf{X}}(f, \mathcal{U})$.

Now, let us consider the special case of a graphing G with set of control states S^G. For an intuitive understanding, one can think of G as the representation of a PRAM machine. We focus on the specific open cover indexed by the set of control states, i.e. $\mathcal{S} = (\mathbf{X} \times \{s\}_{s\in S^G})$, and call it *the states cover*. We will now show how the partial entropy $H^k(G, \mathcal{S})$ is related to the set of *admissible sequence of states*. Let us define those first.

Definition 8.2.5 *Let G be a graphing, with set of control states S^G. An admissible sequence of states is a sequence $\mathbf{s} = s_1 s_2 \dots s_n$ of elements of S^G such that for all $i \in \{1, 2, \dots, n-1\}$ there exists a subset C_i of $\mathbf{X}$ – i.e. a set of configurations – such that G contains an edge from $C_i \times \{s_i\}$ to a subspace of $C_{i+1} \times \{s_{i+1}\}$ (noting $C_n = \mathbf{X}$).*

Example 8.2.1 As an example, let us consider the very simple graphing with four control states a, b, c, d and edges from $\mathbf{X} \times \{a\}$ to $\mathbf{X} \times \{b\}$, from $\mathbf{X} \times \{b\}$ to $\mathbf{X} \times \{c\}$, from $\mathbf{X} \times \{c\}$ to $\mathbf{X} \times \{b\}$ and from $\mathbf{X} \times \{c\}$ to $\mathbf{X} \times \{d\}$. Then the sequences $abcd$ and $abcbcbc$ are admissible, but the sequences aba, $abcdd$, and $abcba$ are not.

Lemma 8.2.1 *Let G be a graphing, and $\mathcal{S}$ its states cover. Then for all integer k, the set $\mathrm{Adm}_k(G)$ of admissible sequences of states of length $k > 1$ is of cardinality $2^{k.H^k(G,\mathcal{S})}$.*

Proof. We show that the set $\mathrm{Adm}_k(G)$ of admissible sequences of states of length k has the same cardinality as the smallest subcover of $\mathcal{S} \vee [G]^{-1}(\mathcal{S}) \vee \dots \vee [G]^{-(k-1)}(\mathcal{S}))$. Hence

$$H^k(G, \mathcal{S}) = \frac{1}{k} \log_2(\mathrm{Card}(\mathrm{Adm}_k(G))),$$

which implies the result.

The proof is done by induction. As a base case, let us consider the set of $\mathrm{Adm}_2(G)$ of admissible sequences of states of length 2 and the open cover $\mathcal{V} = \mathcal{S} \vee [G]^{-1}(\mathcal{S})$ of $D = [G]^{-1}(\mathbf{X})$. An element of $\mathcal{V}$ is an intersection $\mathbf{X} \times \{s_1\} \cap [G]^{-1}(\mathbf{X} \times \{s_2\})$, and it is therefore equal to $C[s_1, s_2] \times \{s_1\}$ where $C[s_1, s_2] \subset \mathbf{X}$ is the set $\{x \in \mathbf{X} \mid [G](x, s_1) \in \mathbf{X} \times \{s_2\}\}$. This set is empty if and only if the sequence $s_1 s_2$ belongs to $\mathrm{Adm}_2(G)$. Moreover, given another sequence of states $s_1' s_2'$ (not necessarily admissible), the sets $C[s_1, s_2]$ and $C[s_1, s_2]$ are disjoint. Hence a set $C[s_1, s_2]$ is *removable from the cover* $\mathcal{V}$ if and only if the sequence $s_1 s_2$ is not admissible. This implies the result for $k = 2$.

The step for the induction is similar to the base case. It suffices to consider the partition $\mathcal{S}_k = \mathcal{S} \vee [G]^{-1}(\mathcal{S}) \vee \dots \vee [G]^{-(k-1)}(\mathcal{S}))$ as $\mathcal{S}_{k-1} \vee [G]^{-(k-1)}(\mathcal{S})$. By the same argument, one can show that elements of $\mathcal{S}_{k-1} \vee [G]^{-(k-1)}(\mathcal{S})$ are of the form $C[\mathbf{s} = (s_0 s_1 \dots s_{k-1}), s_k] \times \{s_1\}$ where $C[\mathbf{s}, s_k] \subset \mathbf{X}$ is the set $\{x \in \mathbf{X} \mid \forall i = 2, \dots, k, [G]^{i-1}(x, s_1) \in \mathbf{X} \times \{s_i\}\}$. Again, these sets $C[\mathbf{s}, s_k]$ are pairwise disjoint and empty if and only if the sequence $s_0 s_1 \dots s_{k-1}, s_k$ is not admissible. ❀

A tractable bound on the number of admissible sequences of states can be obtained by noticing that the sequence $H^k(G, \mathcal{S})$ is *sub-additive*, i.e. $H^{k+k'}(G, \mathcal{S}) \leqslant H^k(G, \mathcal{S}) + H^{k'}(G, \mathcal{S})$. A consequence of this is that $H^k(G, \mathcal{S}) \leqslant kH^1(G, \mathcal{S})$. Thus the number of admissible sequences of states of length k is bounded by $2^{k^2 H^1(G,\mathcal{S})}$. We now study how the cardinality of admissible sequences can be related to the entropy of G.

Lemma 8.2.2 *For all $\epsilon > 0$, there exists an integer N such that for all $k \geqslant N$, $H^k(G, \mathcal{U}) < h([G]) + \epsilon$.*

Proof. Let us fix some $\epsilon > 0$. Notice that if we let $H_k(G,\mathcal{U}) = H^0(\mathcal{U} \vee [G]^{-1}(\mathcal{U}) \vee \cdots \vee [G]^{-(k-1)}(\mathcal{U})))$, the sequence $H_k(U)$ satisfies $H_{k+l}(\mathcal{U}) \leqslant H_k(\mathcal{U}) + H_l(\mathcal{U})$. By Fekete's lemma on subadditive sequences, this implies that $\lim_{k\to\infty} H_k/k$ exists and is equal to $\inf_k H_k/k$. Thus $h([G],\mathcal{U}) = \inf_k H_k/k$.

Now, the entropy $h([G])$ is defined as $\sup_{\mathcal{U}} \lim_{k\to\infty} H_k(\mathcal{U})/k$. This then rewrites as $\sup_{\mathcal{U}} \inf_k H_k(\mathcal{U})/k$. We can conclude that $h([G]) \geqslant \inf_k H_k(\mathcal{U})/k$ for all finite open cover $\mathcal{U}$.

Since $\inf_k H_k(\mathcal{U})/k$ is the limit of the sequence H_k/k, there exists an integer N such that for all $k \geqslant N$ the following inequality holds: $|H_k(\mathcal{U})/k - \inf_k H_k(\mathcal{U})/k| < \epsilon$, which rewrites as $H_k(\mathcal{U})/k - \inf_k H_k(\mathcal{U})/k < \epsilon$. From this we deduce $H_k(\mathcal{U})/k < h([G]) + \epsilon$, hence $H^k(G,\mathcal{U}) < h([G]) + \epsilon$ since $H^k(G,\mathcal{U}) = H_k(G,\mathcal{U})$. ❀

Lemma 8.2.3 *Let G be a graphing, and let $c : k \mapsto Card(\mathrm{Adm}_k(G))$. Then $c(k) = O(2^{k.h([G])})$ as k goes to infinity.*

Lastly, we prove a result bounding the entropy of a map $\alpha(m) * \beta(m')$ in the CREW of AMCS. The result is essentially a consequence of the product rule (Theorem 3 in [123], or [124]) stating that the entropy of a product $h(f \times g)$ is bounded above by the sum $h(f) + h(g)$.

[123]: Adler et al. (1965), *Topological Entropy*

[124]: Goodwyn (1971), *The Product Theorem for Topological Entropy*

Lemma 8.2.4 *Let $\alpha : \mathrm{M}\langle G, \mathrm{R}\rangle \curvearrowright \mathbf{X} \times \mathbf{Y}$ and $\beta : \mathrm{M}\langle H, \mathrm{Q}\rangle \curvearrowright \mathbf{X} \times \mathbf{Z}$ be AMCS such that every non-central element of β acts as the identity on $\mathbf{Z}$. Then for all $m \in \mathrm{M}\langle G, \mathrm{R}\rangle$ and $m' \in \mathrm{M}\langle H, \mathrm{Q}\rangle$, the entropy of $\alpha(m) * \beta(m')$ is bounded by the sum of the entropies of $\alpha(m)$ and $\beta(m')$:*

$$h(\alpha(m) * \beta(m') \leqslant h(\alpha(m)) + h(\beta(m')).$$

Proof. We show that the entropy of $\alpha(m) * \beta(m')$ is bounded by the entropy of $\alpha(m) \times \beta(m)$. The result then follows by the product rule [124]. We distinguish two cases: the first case is when one of $\alpha(m)$ or $\beta(m')$ is central, i.e. $\alpha(m); \pi_{\mathbf{X}} = \pi_{\mathbf{X}}$ or $\beta(m'); \pi_{\mathbf{X}} = \pi_{\mathbf{X}}$, the second case is when both $\alpha(m)$ and $\beta(m')$ act non-trivially on $\mathbf{X}$.

[124]: Goodwyn (1971), *The Product Theorem for Topological Entropy*

For the first case, we may consider that $\beta(m')$ is central without loss of generality. It is then of the form $\tilde{\beta} \times \mathrm{Id}_{\mathbf{X}}$ with $\tilde{\beta} : \mathbf{Z} \to \mathbf{Z}$, and the key observation is that

$$\alpha(m) * \beta(m') = \alpha(m) \times \tilde{\beta}$$

in this case. We now apply the product rule on both identities. From the first identity, we get

$$h(\beta(m')) = h(\tilde{\beta}) + h(\mathrm{Id}_{\mathbf{X}} = h(\tilde{\beta}),$$

since the entropy of the identity is equal to 0, and from the second identity, we get

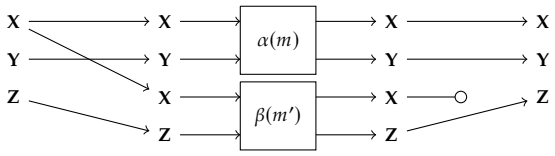
$$h(\alpha(m) * \beta(m')) \leq h(\alpha(m')) + h(\tilde{\beta}).$$

Combining both we obtain that $h(\alpha(m) * \beta(m')) = h(\alpha(m)) + h(\beta(m'))$.

For the second case, the definition of $\alpha(m) * \beta(m')$ states that it is equal to the following map:

$$\Delta_2; \alpha(m) \times (\beta(m'); \pi_{\mathbf{Z}}).$$

Diagrammatically, this is defined as:



We will now bound the entropy of $\alpha(m) * \beta(m')$. This is where we will use the hypothesis that $\beta(m')$ acts as the identity on $\mathbf{Z}$, i.e. that $\beta(m')(x, z) = (x', z)$. Indeed, from this hypothesis, one can deduce that

$$\alpha(m) * \beta(m')(x, y, z) = \alpha(m)(x, y) \times \mathrm{Id}_{\mathbf{Z}}(z),$$

hence

$$h(\alpha(m) * \beta(m')) = h(\alpha(m)) + h(\mathrm{Id}_{\mathbf{Z}}) = h(\alpha(m)) \leq h(\alpha(m)) + h(\beta(m')),$$

since $h(\mathrm{Id}_{\mathbf{Z}}) = 0$ and $h(\beta(m')) \geq 0$.

❀

Cells Decomposition

Now, let us consider a deterministic graphing G, with its state cover $\mathcal{S}$. We fix a length $k > 2$ and reconsider the sets $C[\mathbf{s}] = C[(s_1 s_2 \dots s_{k-1}, s_k)]$ (for a sequence of states $\mathbf{s} = s_1 s_2 \dots s_k$) that appear in the proof of Lemma 8.2.1. The set $(C[\mathbf{s}])_{\mathbf{s} \in \mathrm{Adm}_k(G)}$ is a partition of the space $[G]^{-k+1}(\mathbf{X})$.

This decomposition splits the set of initial configurations into cells satisfying the following property: *for any two initial configurations contained in the same cell $C[\mathbf{s}]$, the k-th first iterations of G goes through the same admissible sequence of states* $\mathbf{s}$.

Definition 8.2.6 *Let G be a deterministic graphing, with its state cover $\mathcal{S}$. Given an integer k, we define the k-th cell decomposition of $\mathbf{X}$ along G as the partition $\{C[\mathbf{s}] \mid \mathbf{s} \in \mathrm{Adm}_k(G)\}$.*

Then Lemma 8.2.1 provides a bound on the cardinality of the k-th cell decomposition. Using the results in the previous section, we can then obtain the following proposition.

Proposition 8.2.5 *Let G be a deterministic graphing, with entropy $h(G)$. The cardinality of the k-th cell decomposition of $\mathbf{X}$ w.r.t. G, as a function $c(k)$ of k, is asymptotically bounded by $g(k) = 2^{k.h([G])}$, i.e. $c(k) = O(g(k))$.*

We also state another bound on the number of cells of the k-th cell decomposition, based on the state cover entropy, i.e. the entropy with respect to the state cover rather than the usual entropy which takes the supremum of cover entropies when the cover ranges over all finite covers of the space. This result is a simple consequence of Lemma 8.2.1.

Proposition 8.2.6 *Let G be a deterministic graphing. We consider the* state cover entropy $h_0([G]) = \lim_{n \to \infty} H^n_{\mathbf{X}}([G], \mathcal{S})$ *where $\mathbf{S}$ is the state cover. The cardinality of the k-th cell decomposition of $\mathbf{X}$ w.r.t. G, as a function $c(k)$ of k, is asymptotically bounded by $g(k) = 2^{k.h_0([G])}$, i.e. $c(k) = O(g(k))$.*

Entropic co-trees and k-th computational forests

The results from the last section providing bounds on the number of cells in the k-th cell decomposition are in fact enough to recover most of the lower bounds results that we reprove in this chapter. Indeed, Steele and Yao result, Cucker's result, and Mulmuley's bounds on PRAMS without bit operations only use bounds on the k-th cell decomposition. However, Ben-Or's improvement of Steele and Yao bounds rests upon a more detailed decomposition that we now abstract under the name of *entropic co-trees*. We will prove the main technical lemma of this chapter that provides upper bounds on the number and degrees of polynomial equations and inequations defining the k-th cell decomposition based on the entropic co-trees. This more general lemma will allow us to recover all the results states above as well as Ben-Or's result, and will be used to prove a strengthening of Mulmuley's result.

Definition 8.2.7 (k-th entropic co-tree) *Consider a deterministic graphing representative T, and fix an element $\top$ of the set of control states. We can define the k-th entropic co-tree of T along $\top$ and the state cover inductively:*

- *$k = 0$, the co-tree $coT_0(T)$ is simply the root $n^\epsilon = \mathbf{R}^n \times \{\top\}$;*
- *$k = 1$, one considers the preimage of n^ϵ through T, i.e. $T^{-1}(\mathbf{R}^n \times \{\top\})$ the set of all non-empty sets $\alpha(m_e)^{-1}(\mathbf{R}^n \times \{\top\})$ and intersects it pairwise with the state cover, leading to a finite family (of cardinality bounded by the number of states multiplied by the number of edges fo T) $(n_e^i)_i$ defined as $n^i = T^{-1}(n^\epsilon) \cap \mathbf{R}^n \times \{i\}$. The first entropic co-tree $coT_1(T)$ of T is then the tree defined by linking each n_e^i to n^ϵ with an edge labelled by m_e;*
- *$k + 1$, suppose defined the k-th entropic co-tree of T, defined as a family of elements $n_{\mathbf{e}}^\pi$ where π is a finite sequence of states of length at most k and $\mathbf{e}$ a sequence of edges of T of the same length, and where $n_{\mathbf{e}}^\pi$ and $n_{\mathbf{e}'}^{\pi'}$ are linked by an edge labelled f if and only if $\pi' = \pi.s$ and $\mathbf{e}' = f.\mathbf{e}$ where s is a state and f an edge of T. We consider the subset of elements $n_{\mathbf{e}'}^\pi$, where π is exactly of length k, and for each such element we define new nodes $n_{e.\mathbf{e}'}^{\pi.s}$, defined as $\alpha(m_e)^{-1}(n_{\mathbf{e}'}^\pi) \cap \mathbf{R}^n \times \{s\}$ when it is non-empty. The $k+1$-th entropic co-tree $coT_{k+1}(T)$ is defined by extending the k-th entropic co-tree $coT_k(T)$, adding the nodes $n_{e.\mathbf{e}}^{\pi.s}$, and linking them to $n_{\mathbf{e}'}^\pi$ with an edge labelled by e.*

Remark 8.2.1 The co-tree can alternatively be defined non-inductively in the following way: the $n_{\mathbf{e}}^\pi$ for π is a finite sequence of states and $\mathbf{e}$ a sequence of edges of T of the same length by $n_\epsilon^\epsilon = \mathbf{R}^n \times \{\top\}$ and

$$n_{\mathbf{e}.e}^{\pi.s} = \left[\alpha(m_e)^{-1}(n_{\mathbf{e}}^\pi)\right] \cap [\mathbf{R}^n \times \{s\}]$$

The k-th entropic co-tree of T along $\top$ has as vertices the non-empty sets $n_{\mathbf{e}}^\pi$ for π and $\mathbf{e}$ of length at most k and as only edges, links $n_{\mathbf{e}.e}^{\pi.s} \to n_{\mathbf{e}}^\pi$ labelled by m_e.

This definition formalises a notion that appears more or less clearly in the work of Lipton and Steele, and of Ben-Or, as well as in the proof by Mulmuley. The nodes for paths of length k in the k-th co-tree corresponds to the k-th cell decomposition, and the corresponding path defines the polynomials describing the semi-algebraic set decided by a computational tree. The co-tree can be used to reconstruct the algebraic computation tree T from the graphing representative $[T]$, or constructs *some* algebraic

computation tree (actually a forest) that approximates the computation of the graphing F under study when the latter is not equal to $[T]$ for some tree T.

Definition 8.2.8 (k-th computational forest) *Consider a deterministic graphing T, and fix an element $\top$ of the set of control states. We define the k-th computational forest of T along $\top$ and the state cover as follows. Let $\mathrm{coT}_k(T)$ be the k-th entropic co-tree of T. The k-th computational forest of T is defined by regrouping all elements $n^{\pi}_{e.\vec{e}'}$, of length m: if the set $N^m_e = \{n^{\pi}_{e.\vec{e}'}, \in \mathrm{coT}_k(T) \mid \mathrm{len}(\pi) = m\}$ is non-empty it defines a new node N^m_e. Then one writes down an edge from N^m_e to $N^{m-1}_{e'}$, labelled by e, if and only if there exists $n^{s.\pi}_{e.e'.\vec{f}} \in N^m_e$ such that $n^{\pi}_{e'.\vec{f}} \in N^{m-1}_{e'}$.*

One checks easily that the k-th computational forest is indeed a forest: an edge can exist between N^m_e and N^n_f only when $n = m + 1$, a property that forbids cycles. The following proposition shows how the k-th computational forest is linked to computational trees.

Proposition 8.2.7 *If T is a computational tree of depth k, the k-th computational forest of $[T]$ is a tree which defines straightforwardly a graphing (treeing) representative of T.*

We now state and prove an easy bound on the size of the entropic co-trees.

Proposition 8.2.8 (Size of the entropic co-trees) *Let T be a graphing representative, E its set of edges, and $\mathrm{Seq}_k(E)$ the set of length k sequences of edges in T. The number of nodes of its k-th entropic co-tree $\mathrm{coT}_k(T)$, as a function $n(k)$ of k, is asymptotically bounded by $Card(\mathrm{Seq}_k(E)).2^{(k+1).h([G])}$.*

Proof. For a fixed sequence $\vec{e}$, the number of elements $n^{\pi}_{\vec{e}}$ of length m in $\mathrm{coT}_k(T)$ is bounded by the number of elements in the m-th cell decomposition of T, and is therefore bounded by $g(m) = 2^{m.h([T])}$ by Proposition 8.2.5. The number of sequences $\vec{e}$ is bounded by $\mathrm{Card}(\mathrm{Seq}_k(E))$ and therefore the size of $\mathrm{coT}_k(T)$ is thus bounded by $\mathrm{Card}(\mathrm{Seq}_k(E)).2^{(k+1).h([T])}$. ❀

From the proof, one sees that the following variant of Proposition 8.2.6 holds.

Proposition 8.2.9 *Let G be a deterministic graphing with a finite set of edges E, and $\mathrm{Seq}_k(E)$ the set of length k sequences of edges in G. We consider the* state cover entropy $h_0([G]) = \lim_{n\to\infty} H^n_{\mathbf{X}}([G], \mathcal{S})$ *where $\mathbf{S}$ is the state cover. The cardinality of the length k nodes of the entropic co-tree of G, as a function $c(k)$ of k, is asymptotically bounded by $g(k) = Card(\mathrm{Seq}_k(E)).2^{k.h_0([G])}$, which is itself bounded by $2^{Card}(E).2^{k.h_0([G])}$.*

The technical lemma

This definition formalises a notion that appears more or less clearly in the work of Steele and Yao, and of Ben-Or, as well as in the proof by Mulmuley. The vertices for paths of length k in the k-th co-tree corresponds to the k-th cell decomposition, and the corresponding path

defines the polynomials describing the semi-algebraic set decided by a computational tree. While in Steele and Yao and Mulmuley's proofs, one obtain directly a polynomial for each cell, we here need to construct a system of equations for each branch of the co-tree.

Given a CREW$^p(\alpha_{\mathbf{R}\text{full}})$-graphing representative G we will write $\sqrt[\partial]{G}$ the maximal value of n for which an instruction $\sqrt[n]{i}(j)$ appears in the realiser of an edge of G.

The proof of this theorem is long but simple to understand as it follows Ben-Or's method. We define, for each vertex of the k-th entropic co-tree, a system of algebraic equations (each of degree at most 2). The system is defined by induction on k, and uses the information of the specific instruction used to extend the sequence indexing the vertex at each step. For instance, the case of division follows Ben-Or's method, introducing a fresh variable and writing down two equations. Following Mulmuley [125], the input variables are split into numerical and non-numerical inputs, and one assumes that indirect references do not depend on non-numerical inputs. This implies that all indirect references have a fixed value determined by the non-numerical input; hence in the analysis below – which focuses on numerical inputs – indirect references correspond to references to a fixed value register.

[125]: Mulmuley (1999), *Lower Bounds in a Parallel Model without Bit Operations*

Lemma 8.2.10 *Let G be a computational graphing representative with edges realised only by generators of the* AMC CREW$^p(\alpha_{\mathbf{R}\text{full}})$*, and* $\mathrm{Seq}_k(E)$ *the set of length k sequences of edges in G. Suppose G computes the membership problem for $W \subseteq \mathbf{R}^n$ in k steps, i.e. for each element of $\mathbf{R}^n$, $\pi_{\mathbf{S}}(G^k(x)) = \top$ if and only if $x \in W$. Then W is a semi-algebraic set defined by at most* $Card(\mathrm{Seq}_k(E)).2^{k.h_0([G])}$ *systems of pk equations of degree at most* $\max(2, \sqrt[\partial]{G})$ *and involving at most $p(k+n)$ variables and $p(k+n)$ inequalities.*

This theorem extends to the case of general computational graphings by considering the *algebraic degree* of the graphing.

Definition 8.2.9 (Algebraic degree) *Let $\alpha : \langle G, \mathrm{R}\rangle \curvearrowright \mathbf{X}$ be an* AMC*. The algebraic degree of an element of $\mathrm{M}\langle G, \mathrm{R}\rangle$ is the minimal number of generators needed to express it. The algebraic degree of an α-graphing is the maximum of the algebraic degrees of the realisers of its edges.*

If an edge is realised by an element m of algebraic degree D, then the method above applies by introducing the D new equations corresponding to the D generators used to define m. The general result then follows.

Lemma 8.2.11 *Let G be a* CREW$^p(\alpha_{\mathbf{R}\text{full}})$*-computational graphing representative,* $\mathrm{Seq}_k(E)$ *the set of length k sequences of edges in G, and D its algebraic degree. Suppose G computes the membership problem for $W \subseteq \mathbf{R}^n$ in k steps, i.e. for each element of $\mathbf{R}^n$, $\pi_{\mathbf{S}}(G^k(x)) = \top$ if and only if $x \in W$. Then W is a semi-algebraic set defined by at most* $Card(\mathrm{Seq}_k(E)).2^{k.h_0([G])}$ *systems of pkD equations of degree at most* $\max(2, \sqrt[\partial]{G})$ *and involving at most $pD(k+n)$ variables.*

8.3. Digression: a bit of algebraic geometry and topology

The Milnor-Thom theorem

Some of the results above rely on the Milnor-Thom theorem. This theorem, proved independently by Milnor [126] and Thom [127], provides a bound on the sum of the Betti numbers of algebraic varieties in $\mathbf{R}^n$, depending on n and the maximal degree k of the polynomials defining it. A consequence of this bound is that it also provides a bound on the zero-th Betti number, i.e. the number of connected components. The following statement of the Milnor-Thom theorem is adapted to semi-algebraic sets and is due to Ben-Or [119].

[126]: Milnor (1964), *On the Betti numbers of real varieties*

[127]: Thom (1965), *Sur l'homologie des variétés algébriques réelles*

[119]: Ben-Or (1983), *Lower Bounds for Algebraic Computation Trees*

Theorem 8.3.1 *Let $V \subseteq \mathbf{R}^n$ be a set defined by polynomial in·equations ($n, m, h \in \mathbf{N}$):*

$$\begin{cases} q_1(x_1, \ldots, x_n) = 0 \\ \vdots \\ q_m(x_1, \ldots, x_n) = 0 \\ p_1(x_1, \ldots, x_n) > 0 \\ \vdots \\ p_s(x_1, \ldots, x_n) > 0 \\ p_{s+1}(x_1, \ldots, x_n) \geqslant 0 \\ \vdots \\ p_h(x_1, \ldots, x_n) \geqslant 0 \end{cases}$$

for $p_i, q_i \in \mathbf{R}[X_1, \ldots, X_n]$ of degree lesser than d.

Then $\beta_0(V)$ is at most $d(2d-1)^{n+h-1}$, where $d = \max\{2, \deg(q_i), \deg(p_j)\}$.

Algebraic surfaces for an optimization problem

Geometric Interpretation of Optimization Problems. We start by showing how decision problems of a particular form induce a binary partition of the space $\mathbf{Z}^d$: the points that are accepted and those that are rejected. Intuitively, the machine decides the problem if the partition it induces refines the one of the problem.

We will consider problems of a very specific form: decisions problems in $\mathbf{Z}^3$ associated to optimization problems. Let $\mathcal{P}_{\text{opt}}$ be an optimization problem on $\mathbf{R}^d$. Solving $\mathcal{P}_{\text{opt}}$ on an instance t amounts to optimizing a function $f_t(\cdot)$ over a space of parameters. We note $\text{Max}\mathcal{P}_{\text{opt}}(t)$ this optimal value. An affine function $\mathsf{Param} : [p; q] \to \mathbf{R}^d$ is called a *parametrization* of $\mathcal{P}_{\text{opt}}$. Such a parametrization defines naturally a decision problem $\mathcal{P}_{\text{dec}}$: for all $(x, y, z) \in \mathbf{Z}^3$, $(x, y, z) \in \mathcal{P}_{\text{dec}}$ iff $z > 0$, $x/z \in [p; q]$ and $y/z \leq \text{Max}\mathcal{P}_{\text{opt}} \circ \mathsf{Param}(x/z)$.

In order to study the geometry of $\mathcal{P}_{\text{dec}}$ in a way that makes its connection with $\mathcal{P}_{\text{opt}}$ clear, we consider the ambient space to be $\mathbf{R}^3$, and we define the *ray* $[p]$ of a point p as the half-line starting at the origin and containing p. The projection $\Pi(p)$ of a point p on a plane is the intersection of $[p]$ and the affine plane $\mathcal{A}_1$ of equation $z = 1$. For any point $p \in \mathcal{A}_1$,

and all $p_1 \in [p]$, $\Pi(p_1) = p$. It is clear that for $(p, p', q) \in \mathbf{Z}^2 \times \mathbf{N}^+$, $\Pi((p, p', q)) = (p/q, p'/q, 1)$.

The *cone* $[C]$ of a curve C is the set of rays of points of the curve. The projection $\Pi(C)$ of a surface or a curve C is the set of projections of points in C. We note Front the frontier set

$$\mathsf{Front} = \{(x, y, 1) \in \mathbf{R}^3 \mid y = \mathrm{Max}\mathcal{P}_{\mathrm{opt}} \circ \mathsf{Param}(x)\}.$$

and we remark that

$$[\mathsf{Front}] = \{(x, y, z) \in \mathbf{R}^2 \times \mathbf{R}^+ \mid y/z = \mathrm{Max}\mathcal{P}_{\mathrm{opt}} \circ \mathsf{Param}(x/z)\}.$$

Finally, a machine M decides the problem $\mathcal{P}_{\mathrm{dec}}$ if the sub-partition of accepting cells in $\mathbf{Z}^3$ induced by the machine is finer than the one defined by the problem's frontier [Front] (which is defined by the equation $y/z \leq \mathrm{Max}\mathcal{P}_{\mathrm{opt}} \circ \mathsf{Param}(x/z)$).

Parametric Complexity. We now further restrict the class of problems we are interested in: we will only consider $\mathcal{P}_{\mathrm{opt}}$ such that Front is simple enough. Precisely:

Definition 8.3.1 *We say that* Param *is an* affine parametrization *of* $\mathcal{P}_{\mathrm{opt}}$ *if* $\mathsf{Param}; \mathrm{Max}\mathcal{P}_{\mathrm{opt}}$ *is*

- *convex*
- *piecewise linear, with breakpoints* $\lambda_1 < \cdots < \lambda_\rho$
- *such that the* $(\lambda_i)_i$ *and the* $(\mathrm{Max}\mathcal{P}_{\mathrm{opt}} \circ \mathsf{Param}(\lambda_i))_i$ *are all rational.*

The (parametric) complexity $\rho(\mathsf{Param})$ *is defined as the number of breakpoints of* $\mathsf{Param}; \mathrm{Max}\mathcal{P}_{\mathrm{opt}}$.

An optimization problem that admits an affine parametrization of complexity ρ is thus represented by a surface [Front] that is quite simple: the cone of the graph of a piecewise affine function, constituted of ρ segments. We say that such a surface is a *ρ-fan*. This restriction seems quite serious when viewed geometrically. Nonetheless, many optimization problems admit such a parametrization.

Definition 8.3.2 *Let* $\mathcal{P}_{\mathrm{opt}}$ *be an optimization problem and* Param *be an affine parametrization of it. The* bitsize *of the parametrization is the maximum of the bitsizes of the numerators and denominators of the coordinates of the breakpoints of* $\mathsf{Param}; \mathrm{Max}\mathcal{P}_{\mathrm{opt}}$.

In the same way, we say that a ρ-fan is of bitsize β *if all its breakpoints are rational and the bitsize of their coordinates is lesser thant β.*

The following results are due to Murty [128] and Carstensen [129].

[128]: Murty (1980), *Computational complexity of parametric linear programming*

[129]: Carstensen (1983), *The Complexity of Some Problems in Parametric Linear and Combinatorial Programming*

Theorem 8.3.2 *There exists:*

1. *an affine parametrization of bitsize $O(n)$ and complexity $2^{\Omega(n)}$ of combinatorial linear programming, where n is the total number of variables and constraints of the problem.*
2. *an affine parametrization of bitsize $O(n^2)$ and complexity $2^{\Omega(n)}$ of the* `maxflow` *problem for directed and undirected networks, where n is the number of nodes in the network.*

We refer the reader to Mulmuley's paper [125, Thm. 3.1.3] for proofs, discussions and references.

Algebraic Surfaces. An algebraic surface in $\mathbf{R}^3$ is a surface defined by an equation of the form $p(x, y, z) = 0$ where p is a polynomial. If S is a set of surfaces, each defined by a polynomial, the *total degree* of S is defined as the sum of the degrees of polynomials defining the surfaces in S.

Let K be a compact of $\mathbf{R}^3$ delimited by algebraic surfaces and S be a finite set of algebraic surfaces, of total degree δ. We can assume that K is actually delimited by two affine planes of equation $z = \mu$ and $z = 2\mu_z$ and the cone of a rectangle $\{(x, y, 1) \mid |x|, |y| \leqslant \mu_{x,y}\}$, by taking any such compact containing K and adding the surfaces bounding K to S. S defines a partition of K by considering maximal compact subspaces of K whose boundaries are included in surfaces of S. Such elements are called the *cells* of the decomposition associated to S.

Definition 8.3.3 *Let K be a compact of $\mathbf{R}^3$. A finite set of surfaces S on K* separates *a ρ-fan* Fan *on K if the partition on $\mathbf{Z}^3 \cap K$ induced by S is finer than the one induced by* Fan*.*

Theorem 8.3.3 (Mulmuley) *Let S be a finite set of algebraic surfaces of total degree δ. There exists a polynomial P such that, for all $\rho > P(\delta)$, S does not separate ρ-fans.*

8.4. Lower Bounds results

Ben-Or

We now recover Ben-Or result by obtaining a bound on the number of connected components of the subsets $W \subseteq \mathbf{R}^n$ whose membership problem is computed by a graphing in less than a given number of iterations. This theorem is obtained by applying the Milnor-Thom theorem on the obtained systems of equations to bound the number of connected components of each cell. Notice that in this case $p = 1$ and $\sqrt[2]{G} = 2$ since the model of algebraic computation trees use only square roots. A mode general result holds for algebraic computation trees extended with arbitrary roots, but we here limit ourselves here to the original model.

Theorem 8.4.1 *Let G be a computational $\alpha_{\mathbf{R}\text{full}}$-graphing representative translating an algebraic computational tree, $\mathrm{Seq}_k(E)$ the set of length k sequences of edges in G. Suppose G computes the membership problem for $W \subseteq \mathbf{R}^n$ in k steps. Then W has at most Card$(\mathrm{Seq}_k(E)).2^{k.h_0([G])+1}3^{2k+n-1}$ connected components.*

Proof. By Lemma 8.2.10 (using the fact that $p = 1$ and $\sqrt[2]{G} = 2$), the problem W decided by G in k steps is described by at most $\mathrm{Card}(\mathrm{Seq}_k(E)).2^{k.h_0([G])}$ systems of k equations of degree 2 involving at most $k + n$ variables and at most k inequalities. Applying Theorem 8.3.1, we deduce that each such system of in·equations (of k equations of degree 2 in $\mathbf{R}^{k+n}$) describes a

semi-algebraic variety S such that $\beta_0(S) < 2.3^{(n+k)+k-1}$. This being true for each of the $\mathrm{Card}(\mathrm{Seq}_k(E)).2^{k.h_0([G])}$ cells, we have that

$$\beta_0(W) < \mathrm{Card}(\mathrm{Seq}_k(E)).2^{k.h_0([G])+1}3^{2k+n-1}. \qquad ❀$$

Since a subset computed by a tree T of depth k is computed by $[\![T]\!]$ in k steps by Theorem 8.1.1, we get as a corollary the original theorem by Ben-Or relating the number of connected components of a set W and the depth of the algebraic computational trees that compute the membership problem for W.

Corollary 8.4.2 ([119, Theorem 5]) *Let $W \subseteq \mathbf{R}^n$ be any set, and let N be the maximum of the number of connected components of W and $\mathbf{R}^n \setminus W$. An algebraic computation tree computing the membership problem for W has height $\Omega(\log N)$.*

Proof. Let T be an algebraic computation tree computing the membership problem for W, and consider the computational treeing $[T]$. Let d be the height of T; by definition of $[T]$ the membership problem for W is computed in exactly d steps. Thus, by the previous theorem, W has at most $\mathrm{Card}(\mathrm{Seq}_k(E)).2^{d.h_0([T])+1}3^{2d+n-1}$ connected components. As the interpretation of an algebraic computational tree, $h_0([T])$ is at most equal to 2, and $\mathrm{Card}(\mathrm{Seq}_k(E))$ is bounded by 2^d. Hence $N \leqslant 2^d 2^{2d+1}3^{n-1}3^{2d}$, i.e. $d = \Omega(\log N)$. ❀

Cucker's theorem

Cucker's proof considers the problem defined as the following algebraic set.

Definition 8.4.1 *Define the set $\mathfrak{Fer} = \{x \in \mathbf{R}^\omega \mid |x| = n \Rightarrow x_1^{2^n} + x_2^{2^n} = 1\}$, where $|x| = \max\{n \in \omega \mid x_n \neq 0\}$.*

It can be shown to lie within $\textsc{Ptime}_{\mathbf{R}}$ (Proposition 3 in [130]), i.e. it is decided by a real Turing machine [131] – i.e. working with real numbers and real operations –, running in polynomial time.

[130]: Cucker (1992), $\mathrm{P}_{\mathbf{R}} \neq \mathrm{NC}_{\mathbf{R}}$

[131]: Blum et al. (1989), *On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines*

Theorem 8.4.3 *The problem $\mathfrak{Fer}$ is in $\textsc{Ptime}_{\mathbf{R}}$.*

We now prove that $\mathfrak{Fer}$ is not computable by an algebraic circuit of polylogarithmic depth Theorem 3.2 in [130]. The proof follows Cucker's argument, but uses the lemma proved in the previous section.

[130]: Cucker (1992), $\mathrm{P}_{\mathbf{R}} \neq \mathrm{NC}_{\mathbf{R}}$

Corollary 8.4.4 *No algebraic circuit of depth $k = \log^i n$ and size kp compute $\mathfrak{Fer}$.*

Proof. For this, we will use the lower bounds result obtained in the previous section. Indeed, by Theorem 8.1.2 and Lemma 8.2.11, any problem decided by an algebraic circuit of depth k is a semi-algebraic set defined by at most $\mathrm{Card}(\mathrm{Seq}_k(E)).2^{k.h_0([G])}$ systems of k equations of degree at most $\max(2, \sqrt[\partial]{G}) = 2$ (since only square roots are allowed in the model) and involving at most $k + n$ variables. But the curve $\mathfrak{F}_{2^n}^{\mathbf{R}}$ defined as $\{x_1^{2^n} + x_2^{2^n} - 1 = 0 \mid x_1, x_2 \in \mathbf{R}\}$ is infinite. As a consequence, one of the

systems of equation must describe a set containing an infinite number of points of $\mathfrak{F}^{\mathbf{R}}_{2^n}$.

This set S is characterized, up to some transformations on the set of equations obtained from the entropic co-tree, by a finite system of inequalities of the form

$$\bigwedge_{i=1}^{s} F_i(X_1, X_2) = 0 \wedge \bigwedge_{j=1}^{t} G_j(X_1, X_2) < 0,$$

where t is bounded by kp and the degree of the polynomials F_i and G_i are bounded by 2^k. Moreover, since $\mathfrak{F}^{\mathbf{R}}_{2^n}$ is a curve and no points in S must lie outside of it, we must have $s > 0$.

Finally, the polynomials F_i vanish on that infinite subset of the curve and thus in a 1-dimensional component of the curve. Since the curve is an irreducible one, this implies that every F_i must vanish on the whole curve. Using the fact that the ideal $(X_1^{2^n} + X_2^{2^n} - 1)$ is prime (and thus radical), we conclude that all the F_i are multiples of $X_1^{2^n} + X_2^{2^n} - 1$ which is impossible if their degree is bounded by $2^{\log^i n}$ as it is strictly smaller than 2^n. ❀

Improving Mulmuley's result

PRAMs over R and maxflow. We will now prove our strengthening of Mulmuley's lower bounds for 'PRAMs without bit operations' [125]. For this, we will combine the results from previous sections to establish the following result.

[125]: Mulmuley (1999), *Lower Bounds in a Parallel Model without Bit Operations*

Theorem 8.4.5 *Let N be a natural number and M be an algebraic PRAM with at most $2^{O((\log N)^c)}$ processors, where c is any positive integer. Then M does not decide* `maxflow` *on inputs of length N in $O((\log N)^c)$ steps.*

Proof. Let N be an integer. Suppose that an algebraic PRAM M with division and roots, with at most $p = 2^{O((\log N)^c)}$ processors, computes `maxflow` on inputs of length at most N in time $k = 2^{O((\log N)^c)}$.

We know that $[\![M]\!]$ has a finite set of edges E. Since the running time of M is equal, up to a constant, to the computation time of the $\mathrm{CREW}^p(\alpha_{\mathbf{R}\mathrm{full}})$-program $[\![M]\!]$, we deduce that if M computes `maxflow` in k steps, then $[\![M]\!]$ computes `maxflow` in at most Ck steps where C is a fixed constant.

By Lemma 8.2.10, the problem decided by $[\![M]\!]$ in Ck steps defines a system of equations separating the integral inputs accepted by M from the ones rejected. I.e. if M computes `maxflow` in Ck steps, then this system of equations defines a set of algebraic surfaces that separate the ρ-fan defined by `maxflow`. Moreover, this system of equation has a total degree bounded by $Ck \max(2, \sqrt[\partial]{G})2p \times 2^{O(\mathrm{Card}(E))} \times 2^{k.h_0([\![\tilde{M}]\!])}$.

By Theorem 8.3.2 and Theorem 8.3.3, there exists a polynomial P such that a finite set of algebraic surfaces of total degree δ cannot separate the $2^{\Omega(N)}$-fan defined by `maxflow` as long as $2^{\Omega(N)} > P(\delta)$. But here the entropy of G is $O(p)$, as a consequence of Lemma 8.2.4. Hence $\delta = O(2^p 2^k)$, contradicting the hypotheses that $p = 2^{O((\log N)^c)}$ and $k = 2^{O((\log N)^c)}$. ❀

This extends Mulmuley's result because of the following fact.

Proposition 8.4.6 *A subset $A \subseteq \mathbf{Z}^k$ is decided by a division-free integer-valued* PRAMs *with k processors in time t if and only if there exists a division-free algebraic* PRAMs *with k processors computing in time t a subset $B \subseteq \mathbf{R}^k$ such that:*

$$\{(x_1, x_2, \ldots, x_k) \in \mathbf{Z}^k \mid (x_1, x_2, \ldots, x_k) \in B\} = A.$$

Proof. The proof is rather straightforward, since addition and multiplication of integers yield integers. As a consequence, the available operations in division-free real PRAMs cannot be used to construct non-integer values from solely integer inputs. ❀

A consequence of this is that Mulmuley's original result becomes a corollary of Theorem 8.4.5. Indeed, suppose a PRAM without bit operations computes the `maxflow` problem in polylogarithmic time. Then there would exist an algebraic PRAM computing `maxflow` in polylogarithmic time, a result contradicting Theorem 8.4.5.

Let us now consider the possibility of lifting this result to integer-valued machines using division. Let M be an integer-valued PRAM. We would like to associate to it an algebraic PRAM $\widetilde{M}$ such that M and $\widetilde{M}$ accept the same (integer) values, with at most a polylogarithmic running time overhead. This implies in particular that algebraic PRAMs (with division, and potentially roots) should be able to compute euclidean division efficiently. It turns out that this is not the case. Indeed, we will show that euclidean division by 2 is in fact not computable in polylogarithmic time by algebraic PRAMs even in the presence of division and arbitrary root operations. This will be obtained using the above results based on entropic co-trees and Mumuley's geometric argument.

Theorem 8.4.7 *Let N be a natural number and M be an algebraic* PRAM *with at most $2^{O((\log N)^c)}$ processors, where c is any positive integer. Then M does not compute euclidean division by 2 on inputs of length N in $O((\log N)^c)$ steps.*

Proof. Suppose that an algebraic PRAM M with division and roots, with at most $p = 2^{O((\log N)^c)}$ processors, computes euclidean division by 2 on inputs of length at most N in time $k = 2^{O((\log N)^c)}$. We know that $[\![M]\!]$ has a finite set of edges E, and the running time of M is equal, up to a constant, to the computation time of the CREW$^p(\alpha_{\mathbf{R}\text{full}})$-program $[\![M]\!]$, we deduce that if M computes euclidean division in k steps, then $[\![M]\!]$ computes euclidean division by 2 in at most Ck steps where C is a fixed constant.

Consider the following problem:

$$\{(x, y, 1) \mid y \leqslant x/\!/2\}$$

It is defined by the frontier set

$$\mathsf{Front} = \{(x, y, 1) \in \mathbf{R}^3 \mid y = x/\!/2\}.$$

and we remark that the induced cone

$$[\mathsf{Front}] = \{(x, y, z) \in \mathbf{R}^2 \times \mathbf{R}^+ \mid y/z = \mathrm{Max}\mathcal{P}_{\mathrm{opt}} \circ \mathsf{Param}(x/z)\}.$$

is a ρ-fan where $\rho = 2^{\Omega(N)}$ is exponential in the maximal size of the inputs.

By Lemma 8.2.10, the problem decided by $[\![M]\!]$ in Ck steps defines a system of equations separating the integral inputs accepted by M from the ones rejected. I.e. if M computes euclidean division by 2 in Ck steps, then this system of equations defines a set of algebraic surfaces that separate the ρ-fan defined above. Moreover, this system of equation has a total degree bounded by $Ck \max(2, \sqrt[\partial]{G})2p \times 2^{O(\mathrm{Card}(E))} \times 2^{k.h_0([\![\tilde{M}]\!])}$.

Now by Theorem 8.3.3, there exists a polynomial P such that a finite set of algebraic surfaces of total degree δ cannot separate the $2^{\Omega(N)}$-fan defined by euclidean division by 2 as long as $2^{\Omega(N)} > P(\delta)$. But here the entropy of G is $O(p)$, as a consequence of Lemma 8.2.4. Hence $\delta = O(2^p 2^k)$, contradicting the hypotheses that $p = 2^{O((\log N)^c)}$ and $k = 2^{O((\log N)^c)}$. ❀

Before ending this chapter, I wish to note that Mulmuley's method can be used to show lower bounds for other problems than `maxflow`. The technique presented here applies in these cases as well, but I chose to focus on `maxflow` because of its particular importance: it is indeed Ptime-complete [132].

[132]: Goldschlager et al. (1982), *The maximum flow problem is log space complete for P*

9. The mathematical structure of abstract programs

I have exposed until this point how abstract models of computation can lead to techniques which can be used in program analysis or to establish complexity lower bounds. However, the notion of AMC did not stem from these fields but emerged from work on semantics of linear logic, and more specifically the models I introduced under the name *Interaction graphs*. One important aspect inherited from these origins is the connection with operator algebras. Indeed, while AMC and abstract programs or graphings do not appeal to notion in operator algebras, they are intimately related to the field. In particular, the set of graphings is a tractable (dense) subset of a von Neumann algebra.

We note here that considering abstract programs and not graphings would make the discussion more complex, as two non-equivalent programs can define the same graphing. In this light, we therefore study only graphings here, noting that a finer distinction (i.e. a smaller equational theory) would lead to more complex algebraic structures.

9.1. Induced action on graphings

We will now establish some formal connection between the abstract setting considered up to this point and operator algebras. Before doing so, we define a notion of *composition* of two graphings.

Definition 9.1.1 *Let α be an AMC, and G, H be two α-graphing representatives. The composition $H \circ G$ is the graphing formally defined by the following representative:*

$$\{(m_e^H m_f^G, \omega_e^H \omega_f^G \mid e \in E^H, f \in E^G\}.$$

Note that in general some products $m_e^H m_f^G$ are such that $\alpha(m_e^H m_f^G)$ is undefined everywhere[1]*, and $H \circ G$ is equivalent (up to the full equational theory) to a graphing representative with less edges (as one can remove those 'trivial' products).*

1: For instance, $\mathtt{read}_0 \circ \mathtt{read}_1$ in the AMC of Turing machines α_{TM}.

We now notice that the set of graphings:

- contains the identity $\{(1, 1)\}$,
- is closed under linear Ω-combinations where Ω is the set of weights, as defined by $G + H = G \uplus H$ (the disjoint union) and:

$$\lambda\{(m_e^G, \omega_e^G) \mid e \in E^G\} = \{(m_e^G, \lambda\omega_e^G) \mid e \in E^G\},$$

- is closed under product, where the product is defined as the composition.

As such it has the structure of an abstract Ω-algebra.

Involutions. If in addition the monoid action is a group action and the monoid Ω is equipped with an involution $(\cdot)^*$, then one can define an involution edge-wise. If $\phi_e : S_e \to T_e$ is an edge, then we define $\phi_{e^*} = \phi^{-1} : T_e \to S_e$. This is clearly a product-reversing involution, making the set of graphings an involutive algebra.

Theorem 9.1.1 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be an AMC, and Ω a monoid of weights. Then the set of Ω-weighted α-graphings* $\mathrm{Graphings}_\Omega(\alpha)$ *has a structure of Ω-algebra. Moreover, if $\mathbb{M}(I)$ is a group and Ω possesses an involution,* $\mathrm{Graphings}_\Omega(\alpha)$ *has the structure of an involutive algebra.*

Moreover, there is a straightforward representation of this algebra as a linear operator on the Hilbert space $L^2(\mathbf{X}, \Omega)$ of square-summable measurable maps from $\mathbf{X} \to \Omega$. First, one defines the following generalisation of the Koopman (of composition) operator:

$$\Lambda_{(m_e,\omega_e)} : \left(f \in L^2(\mathbf{X}, \Omega) \mapsto \omega_e f \circ \alpha(m_e) \in L^2(\mathbf{X}, \Omega)\right).$$

This is extended to graphings by taking the sum of edges:

$$\Lambda_G = \sum_{e \in E^G} \Lambda_{(m_e^G, \omega_e^G)}.$$

Since all operation on $\mathcal{M}(\mathbf{X})$ are defined pointwise, it is not difficult to check that this defines a linear operator:

$$\begin{aligned}
\Lambda_{(m_e,\omega_e)}(f+g) &= \Lambda_{(m_e,\omega_e)}(f) + \Lambda_{(m_e,\omega_e)}(g),\\
\Lambda_{(m_e,\omega_e)}(f \times g) &= \Lambda_{(m_e,\omega_e)}(f) \times \Lambda_{(m_e,\omega_e)}(g),\\
\Lambda_{(m_e,\omega_e)}(\lambda f) &= \lambda \Lambda_{(m_e,\omega_e)}(f).
\end{aligned}$$

Moreover, the Ω-algebra structure is compatible with this representation. Indeed:

$$\lambda \Lambda_{(m_e,\omega_e)} = \Lambda_{(m_e,\lambda\omega_e)},$$

and by definition $\Lambda_{F+G} = \Lambda_F + \Lambda_G$ and $\Lambda_{FG} = \Lambda_G \Lambda_F$. Note the representation reverses the product; it is an *anti-representation*[2].

A consequence of these identities is that the algebra $\mathrm{Graphings}_\Omega(\alpha)$ is generated by elements of the form $(\iota, 1)$ for $\iota \in I$.

In case $\alpha(\iota)$ has an essentially bounded Radon-Nikodym derivative, the operator $\Lambda_{(\iota,1)}$ is bounded and its norm is

$$\left\| \frac{d\mu}{d\alpha(\iota) \circ \mu} \right\|_\infty = \left(\sup_{\mathbf{X}} \left(\frac{d\mu}{d\alpha(\iota) \circ \mu} \right) \right)^{\frac{1}{2}}.$$

2: When $\mathbb{M}(I)$ is a group, one can similarly define a representation (i.e. not reversing the product) by simply precomposing by $\alpha(\iota^{-1})$ instead of $\alpha(\iota)$.

Indeed, this is established by the following computation:

$$\begin{aligned}
\left\|\Lambda_{(\iota,1)}(f)\right\|_2 &= \int_{\mathbf{X}} \overline{(f\circ\alpha(\iota))(x)}(f\circ\alpha(\iota))(x)d\mu \\
&= \int_{\mathbf{X}} |(f\circ\alpha(\iota))(x)|^2 d\mu \\
&= \int_{\mathbf{X}} |(f\circ\alpha(\iota))(x)|^2 \left(\frac{d\mu}{d\alpha(\iota)\circ\mu}\right) d\alpha(\iota)\circ\mu \\
&\leq \int_{\mathbf{X}} |f(x)|^2 \sup_{\mathbf{X}} \left(\frac{d\mu}{d\alpha(\iota)\circ\mu}\right) d\mu \\
&\leq \int_{\mathbf{X}} |f(x)|^2 \left\|\frac{d\mu}{d\alpha(\iota)\circ\mu}\right\|_\infty^2 d\mu \\
&\leq \left\|\frac{d\mu}{d\alpha(\iota)\circ\mu}\right\|_\infty^2 \int_{\mathbf{X}} |f(x)|^2 d\mu \\
&\leq \left\|\frac{d\mu}{d\alpha(\iota)\circ\mu}\right\|_\infty^2 \|f\|_2^2 .
\end{aligned}$$

Graphings as generalising elements of a von Neumann algebra. We now restrict to the case of a group action by measure-preserving transformation. We can show the following theorem which relates graphings with elements of Murray and von Neumann's *group measure space construction*. Note that nowadays the latter construction stemming from a group action $\alpha : G \curvearrowright \mathbf{X}$ is understood as the crossed product $L^\infty(\mathbf{X}) \rtimes_{\tilde{\alpha}} G$ of the algebra of essentially bounded functions $\mathbf{X} \to \mathbf{C}$ by the action induced by α. This point of view shows clearly the two essential algebras arising from the construction:

- the algebra $L^\infty(\mathbf{X})$ which is an abelian von Neumann algebra, which will be *maximal* in the crossed product;
- the algebra $\mathfrak{U}$ generated by the unitaries defined by the elements of G.

Theorem 9.1.2 *Let $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ be a group action by measure-preserving transformations. Then the algebra of finite $\mathbf{C}$-weighted α-graphings can be represented as a dense sub-algebra of $\mathfrak{U}$ in the group measure space von Neumann algebra.*

Proof. The group measure space construction can be defined as the crossed product of $L^\infty(\mathbf{X})$ with the action induced by α. More precisely, we define $L^\infty(\mathbf{X})$ as the algebra of essentially bounded functions $\mathbf{X} \to \mathbf{C}$. The elements of this algebra act naturally on $L^2(\mathbf{X})$ by left multiplication;

writing M_f the operator induced by $f \in L^\infty(\mathbf{X})$, this comes from:

$$\begin{aligned}\left\|M_f g\right\|_2^2 &= \int_{\mathbf{X}} \overline{f(x)g(x)} f(x)g(x) d\mu \\ &= \int_{\mathbf{X}} |f(x)|^2 |g(x)|^2 d\mu \\ &\leq \int_{\mathbf{X}} \left\|f\right\|_\infty^2 |g(x)|^2 d\mu \\ &\leq \left\|f\right\|_\infty^2 \int_{\mathbf{X}} |g(x)|^2 d\mu \\ &\leq \left\|f\right\|_\infty^2 \left\|g\right\|_2^2.\end{aligned}$$

Now the action α lifts to $L^\infty(\mathbf{X})$ by letting $\bar{\alpha}(\iota) : f \in L^\infty(\mathbf{X}) \mapsto f \circ \alpha(\iota) \in L^\infty(\mathbf{X})$. Note here that since α acts by measure-preserving transformations, the lifted action $\bar{\alpha}$ is an action by unitaries.

The crossed product $L^\infty(\mathbf{X}) \rtimes_{\bar{\alpha}} \mathbb{M}(I)$ is then defined as the von Neumann algebra acting on the space $L^2(\mathbb{M}(I), L^2(\mathbf{X}))$ generated by the following operators (for $g \in L^\infty(\mathbf{X})$ and $\iota \in \mathbb{M}(I)$):

$$\begin{aligned}\rho_g &: \xi \in L^2(\mathbb{M}(I), L^2(\mathbf{X})) \mapsto \left(m \mapsto \bar{\alpha}(m^{-1})(g)\xi(m)\right) \\ \pi_\iota &: \xi \in L^2(\mathbb{M}(I), L^2(\mathbf{X})) \mapsto \xi(m\iota^{-1})\end{aligned}$$

This defines:

- a representation $\mathfrak{A}$ of $L^\infty(\mathbf{X})$ as the von Neumann algebra generated by all ρ_g for $g \in L^\infty(\mathbf{X})$;
- the crossed product algebra $\mathfrak{R} = L^\infty(\mathbf{X}) \rtimes_{\bar{\alpha}} \mathbb{M}(I)$ as the von Neumann algebra generated by all ρ_g for $g \in L^\infty(\mathbf{X})$ and all π_ι for $\iota \in \mathbb{M}(I)$;
- a von Neumann algebra $\mathfrak{U} \subset \mathfrak{R}$ generated by the unitaries π_ι for $\iota \in \mathbb{M}(I)$;

We note that $\mathfrak{A}$ is then a maximal abelian sub-algebra[3] of the crossed product $L^\infty(\mathbf{X}) \rtimes_{\bar{\alpha}} \mathbb{M}(I)$.

3: Maximal abelian sub-algebra will appear and be discussed in chapter 11.

Now, we can represent the algebra $\text{Graphings}_{\mathbf{C}}(\alpha)$ as follows on $L^\infty(\mathbf{X}) \rtimes_{\bar{\alpha}} \mathbb{M}(I)$. We simply define the representation of elements of the form $(\iota, 1)$:

$$\Pi_{(\iota,1)} : \xi \in L^2(\mathbb{M}(I), L^2(\mathbf{X})) \mapsto \xi(m\iota^{-1}),$$

and extend to linear combination by letting $\Pi_{\lambda(\iota_1,1)+(\iota_2)} = \lambda\Pi_{(\iota_1,1)} + \Pi_{(\iota_2,1)}$. We can show this is a representation by checking that it preserves products:

$$\begin{aligned}\Pi_{(\iota_1,1)}\Pi_{(\iota_2,1)}(\xi) &= \Pi_{(\iota_1,1)}(\xi(m\iota_2^{-1})) \\ &= \xi(m\iota_2^{-1}\iota_1^{-1}) \\ &= \xi(m(\iota_1\iota_2)^{-1}) \\ &= \Pi_{(\iota_1\iota_2)}(\xi).\end{aligned}$$

Now, since π_ι and $\Pi_{(\iota,1)}$ coincide, and since $\mathfrak{U}$ is the von Neumann algebra generated by all π_ι, we have that all finite linear combinations of elements of the form $\Pi_{(\iota,1)}$ belong to $\mathfrak{U}$. More precisely, the algebra $\mathfrak{U}$ is the closure of all such finite linear combinations with respect to the weak

operator topology. This implies that the set of all finite α-graphings is represented as a dense sub-algebra of $\mathfrak{U}$. ❀

This establishes more formally the intuition behind the introduction of graphings. Graphings generalise the operators induced by a group-measure space construction, while keeping to a more tractable setting. This is one of the most important intuitions behind the work exposed in the remaining parts of this section and the upcoming chapters.

9.2. Lifting the action, defining execution

As we have seen in the previous section, an abstract model of computation defines an algebra of graphings. One interesting fact is that the action $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ then lifts to an action $\lfloor \alpha \rfloor : \mathbb{M}(I) \curvearrowright \mathrm{Graphings}(\alpha)$. The corresponding set of machines $\mathrm{Graphings}(\lfloor \alpha \rfloor)$ is in fact isomorphic to the set $\mathrm{Graphings}(\alpha)$. Indeed, abstract $\lfloor \alpha \rfloor$-graphings act trivially – i.e. through left multiplication – on $\mathrm{Graphings}(\alpha)$.

However, there is a more involved and interesting structure, in which one does not simply act through left multiplication, but through *execution*. Execution was historically defined as the solution of a functional equation [8]. From a category-theoretic point of view it corresponds to a *trace* [133] or in some cases a *partial trace* [134].

[8]: Girard (2006), *Geometry of Interaction IV: the Feedback Equation*

[133]: Joyal et al. (1996), *Traced monoidal categories*

[134]: Malherbe et al. (2012), *Partially traced categories*

More formally, consider that a machine $M \in \mathrm{Graphings}(\alpha)$. We can define its *support* as the minimal subspace $Y \subset \mathbf{X}$ such that for all edge $e \in E^M$, $\mathrm{dom}(\alpha(m_e)) \cup \mathrm{im}(\alpha(m_e)) \subset Y$. Now, given two machines M, N, one can write down their support as Y_M and Y_N and define $C = Y_M \cap Y_N$. Intuitively, if M and N were programs corresponding to proofs via the geometry of interaction representation, Y_M and Y_N represent the occurrences of formulas appearing in M and N respectively, and C would be those occurrences that appear in both proofs, i.e. the formulas appearing in *cut* rules.

Operator-theoretic version

The feedback equation is a functional equation that can in this case be expressed as follows, where we understand M and N as acting on the subspace[4] $L^2(Y_M \setminus C) \oplus L^2(C) \oplus L^2(Y_N \setminus C)$ of $L^2(\mathbf{X})$:

$$\begin{aligned} M(y \oplus c) &= y' \oplus c' \\ N(c' \oplus z) &= c \oplus z' \end{aligned} \tag{9.1}$$

It has a solution when there exists $\mathrm{Ex}(M, N)$ such that $\mathrm{Ex}(M, N)(y \oplus z) = y' \oplus z'$ if and only if there exists c, c' satisfying the equation.

4: We here voluntarily avoid talking about sets of states, but those can be accommodated: a machine does not act on $L^2(\mathbf{X})$ but on $L^2(\mathbf{X}) \otimes L^2(S^M)$; it should be clear that this implies the solution to the equation, if it exists, will act on $L^2(\mathbf{X}) \otimes L^2(S^M) \otimes L^2(S^N)$, i.e. $L^2(\mathbf{X}) \otimes L^2(S^M \times S^N)$, and therefore will have set of control states $S^M \times S^N$.

The special case of matrices

If M and N are matrices, one can decompose them in block matrices along $Y_M \setminus C$, C, and $Y_N \setminus C$, i.e.

$$\begin{pmatrix} M_{Y,Y} & M_{C,Y} \\ M_{Y,C} & M_{C,C} \end{pmatrix}, \qquad \begin{pmatrix} N_{C,C} & N_{Z,C} \\ N_{C,Z} & N_{Z,Z} \end{pmatrix}.$$

When it is defined, the following matrix is a solution to the equation:

$$\begin{pmatrix} \mathrm{Ex}(M,N)_{Y,Y} & \mathrm{Ex}(M,N)_{Y,Z} \\ \mathrm{Ex}(M,N)_{Z,Y} & \mathrm{Ex}(M,N)_{Z,Z} \end{pmatrix},$$

where

$$\begin{aligned}
\mathrm{Ex}(M,N)_{Y,Y} &= M_{Y,Y} + M_{Y,C}(1 - N_{C,C}M_{C,C})^{-1}N_{C,C}M_{C,Y} \\
\mathrm{Ex}(M,N)_{Y,Z} &= M_{Y,C}(1 - N_{C,C}M_{C,C})^{-1}N_{C,Z} \\
\mathrm{Ex}(M,N)_{Z,Y} &= N_{Z,C}(1 - M_{C,C}N_{C,C})^{-1}M_{C,Y} \\
\mathrm{Ex}(M,N)_{Z,Z} &= N_{Z,Z} + N_{Z,C}(1 - M_{C,C}N_{C,C})^{-1}M_{C,C}N_{C,Z}.
\end{aligned}$$

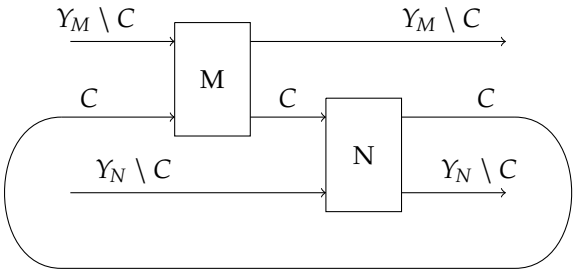
When the norm of M and N is strictly smaller than 1, the product MN has norm at most 1, and this is well-defined. When the norm of both M and N is *smaller or equal* to 1, this operator can still be defined by continuity [8]. But if one does not care about convergence, it is possible to define a general solution at the cost of extracting ourselves from the operator-theoretic framework; this is what we are doing when working with *graphings*.

[8]: Girard (2006), *Geometry of Interaction IV: the Feedback Equation*

Diagrammatically

Before explaining this, let us notice that the equation can be expressed through the following diagram, where we identify the C component of the input of each operator to the C component of the output of the other. This is a particular case of the diagram used to define categorical traces [133].

[133]: Joyal et al. (1996), *Traced monoidal categories*



We can see here that $\mathrm{Ex}(M,N)$ is defined as a fixpoint. This is also visible in the concrete expression on matrices, the fixpoint being computed by the expression $(1-A)^{-1}$ which corresponds to $1 + A + A^2 + \ldots$ (when $\|A\| < 1$). We will first try to define this directly on graphs.

Graphs

Now, if M and N are graphs, say $M = (V^M, E^M, s^M, t^M)$ and $N = (V^N, E^N, s^N, t^N)$, we can try to construct a graph which is a solution to the above equation. Here we identify graphs to matrices (the matrix associated being the adjacency matrix of the graph), and we therefore consider that a graph acts on its set of vertices as a sort of relations: for instance M produces, for each $v \in V^M$, a set $Mv \subset V^M$. If the graph is weighted, we can rather express Mv as a vector in $\mathbf{C}^{(V^M)}$: the coefficient of v' in Mv is computed as[5] $\sum_{e \in E^M(v,v')} \omega(e)$.

5: We here write $E^M(v, v')$ the set $\{e \in E^M \mid s^M(e) = v, t^M(e) = v'\}$.

It is know that under this identification, if G is identified to the matrix M then the graph of paths of length exactly k in the graph G is identified to M^k. The graph fixpoint of the above equation is then equal to the following: $\mathrm{Ex}(M, N)$ is defined as (V, E, s, t), where[6]:

- $V = (V^M \setminus V^M \cap V^N) \cup (V^N \setminus V^M \cap V^N)$, i.e. the symmetric difference between V^M and V^N;
- $E = \{\pi = e_0 e_1 \dots e_{n_\pi} \mid \forall i, (e_i \in E^M \Leftrightarrow e_{i+1} \in E^N) \wedge s^\cdot(e_{i+1} = t^\cdot(e_i)\}$, the set of *alternating paths* between M and N;
- $s : E \to V$ is defined by $s(\pi) = s^\cdot(e_0)$;
- $t : E \to V$ is defined by $t(\pi) = t^\cdot(e_{n_\pi})$;

6: We write $s^\cdot$ to denote either ^{M} or s^N depending on the given argument; we will use $t^\cdot$ in a similar way.

Now, when the graphs M, N correspond to operators of norm at most 1, it can be shown that the graph $\mathrm{Ex}(M, N)$ corresponds to the matrix above defining the solution to the feedback equation [16]. But note that this graph is always defined, even when the corresponding operators are of norm more than 1! However, up to this point, we are only working with finite graphs. We will now extend this construction to α-graphings.

[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*

Graphings

The above definition of the execution on graphs provides a natural way to generalise the operation to α-graphings. Indeed, the naive approach is to define the operation in the same way, i.e. define an α-graphing of alternating paths. Giving it some thoughts, one can realise that this requires some adaptations, but it will turn out that this approach works out. For simplicity, and coherence with published papers [18], we will here work with CAMC in order to specify conditions.

[18]: Seiller (2017), *Interaction Graphs: Graphings*

The main difficulty in defining the α-graphing of alternating paths is that the α-graphing representatives M, N we are starting from may not be decomposed along the separation between the subspaces $Y_M \setminus C$, C, and $Y_N \setminus C$, as illustrated in Figure 9.1. This will require a splitting of the edges of M and N along this separation. However, once this splitting is performed, the definition will be a straightforward adaptation of the execution on graphs. Let us now delve into the technical details.

Now, let M, N be graphing representatives. We define the support Y_M as the subspace of $\mathbf{X}$ defined as $\cup_{e \in E^M} \mathrm{dom}(\alpha(m_e)) \cup \mathrm{im}(\alpha(m_e))$ as above. We can define Y_N in the same way. The subspace C is then defined as $Y_M \cap Y_N$. The issue is that given some $e \in E^M$, it may be true that $\mathrm{dom}(\alpha(m_e))$ is neither included in $Y_N \setminus C$ nor in C, but somehow stands in between[7]. A similar problem arises with $\mathrm{im}(\alpha(m_e))$: it can intersect non trivially both $Y_M \setminus C$ and C.

7: This is not possible in graphs because the underlying space is discrete.

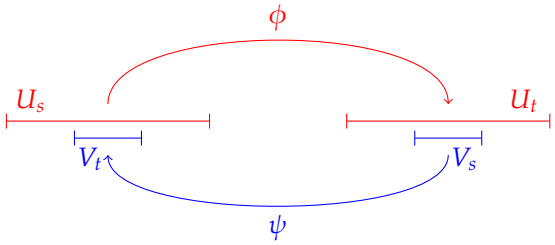


Figure 9.1.: Example of a plugging between graphings (over the real line)

We will therefore consider a new α-graphing representative $M^{\text{Ÿ}\mathbb{C}}$ which consists in splitting each edge e in four distinct edges $(e, \text{in}, \text{in})$, $(e, \text{in}, \text{out})$, $(e, \text{out}, \text{in})$, and $(e, \text{out}, \text{out})$. Those as defined as follows; we only describe the source sets as the weight and monoid element associated are those of the initial edge e:

- $(e, \text{in}, \text{in})$ has source $S_e \cap (Y_M \setminus C) \cap (\alpha(m_e)^{-1}(Y_M \setminus C))$;
- $(e, \text{in}, \text{out})$ has source $S_e \cap (Y_M \setminus C) \cap (\alpha(m_e)^{-1}(C))$;
- $(e, \text{out}, \text{in})$ has source $S_e \cap C \cap (\alpha(m_e)^{-1}(Y_M \setminus C))$;
- $(e, \text{out}, \text{out})$ has source $S_e \cap C \cap (\alpha(m_e)^{-1}(C))$.

This graphing representative is equivalent (for the full equational theory) to the initial graphing representative M.

We can define in a similar way the α-graphing representative $N^{\text{Ÿ}\mathbb{C}}$. It is then possible to define the α-graphing (representative) of alternating paths:

- its set of edges is defined as the set of alternating paths from $(Y_M \setminus C) \cup (Y_N \setminus S)$ to itself, i.e. sequence $\pi = e_0 e_1 \dots e_n$ such that:
  - $e_i \in M^{\text{Ÿ}\mathbb{C}}$ if and only if $e_{i+1} \in N^{\text{Ÿ}\mathbb{C}}$ for $i = 0, \dots, n-1$,
  - $e_0 = (e, \text{in}, \text{out})$,
  - $e_n = (e, \text{out}, \text{in})$,
  - $e_i = (e, \text{out}, \text{out})$ for $i = 1, \dots, n-1$,
  - $\alpha(m_{e_i})(S_{e_i}) \subseteq S_{e_{i+1}}$ for $i = 0, \dots, n-1$.
- for $\pi \in E^{\text{Ex}(M,N)}$, the monoid element m_π is equal to $m_{e_n} m_{e_{n-1}} \dots m_{e_0}$;
- for $\pi \in E^{\text{Ex}(M,N)}$, the source is the subspace of S_{e_0} containing the points that go through all edges, i.e.

$$S_\pi = S_{e_0} \cap \alpha(m_{e_0})^{-1}(S_{e_1}) \cap \dots \cap \alpha(m_{e_{n-1}} \dots m_{e_1} m_{e_0})^{-1}(S_{e_n}).$$

- for $\pi \in E^{\text{Ex}(M,N)}$, the weight (if one works with weighted graphings) ω_π is defined as $\omega_{e_n} \omega_{e_{n-1}} \dots \omega_{e_0}$.

We have thus defined a general notion of execution that captures all the examples above. Indeed, when adding restrictions, we recover the previous definitions:

- if the space is discrete, this corresponds to the execution of graphs, which generalises the execution of finite matrices;
- if the space is not discrete, we can associate operators to M and N and show that the execution of graphings corresponds to the execution of these operators – when the latter is defined.

Remark 9.2.1 One important remark here is related to dynamical systems. If M and N are deterministic, they correspond to partial dynamical

systems DynS(M) and DynS(N). Then the execution can be alternatively defined as the graphing of finite maximal orbits in

$$(\mathrm{DynS}(M) + 1_{Y^N \setminus Y^M})(\mathrm{DynS}(N) + 1_{Y^M \setminus Y^N}).$$

The main property about the execution is associativity, stated in the following theorem.

Theorem 9.2.1 (Associativity of execution) *Given three α-graphings M, N, and P with supports Y_M, Y_N and Y_P respectively. If*[8] *$Y_M \cap Y_N \cap Y_P = \emptyset$, then*

$$\mathrm{Ex}(\mathrm{Ex}(M, N).P) = \mathrm{Ex}(M, \mathrm{Ex}(N, P)).$$

8: In case of graphings over a measured space, the equality can be taken up to a negligible subset.

Before discussing the trefoil / 2-cocycle property, I discuss two results which are essential to define *submodels*.

Submodels

The following lemmas show that if one restrict to deterministic graphings, the execution is well defined. As a consequence, the linear realisability constructions considered in the next chapter can be performed on the set of deterministic graphings, defining a proper submodel.

Lemma 9.2.2 *The execution of two deterministic graphings is a deterministic graphing.*

It turns out that the notion of probabilistic graphing also behaves well under composition, i.e. there exists a *sub-probabilistic* submodel, namely the model of *sub-probabilistic graphings*. As explained below in the more general case of Markov processes (Remark 9.2.2), probabilistic graphings are *not* closed under composition.

Lemma 9.2.3 *The execution of two sub-probabilistic graphings is a sub-probabilistic graphing.*

Both lemmas are proved for measurable graphings in published work [135], and the proof is easily adapted to other cases.

[135]: Seiller (2022), *Zeta functions and the (linear) logic of Markov processes*

Sub-Markov kernels

As a generalisation of the previous case, and based on the identification of probabilistic graphings with discrete-image sub-Markov kernels (Proposition 4.2.2), we can generalise the framework and define an execution on (general) sub-Markov kernels [135]. The need to consider sub-Markov kernels and not only Markov kernels is explained by technical reasons we illustrate below (Remark 9.2.2).

[135]: Seiller (2022), *Zeta functions and the (linear) logic of Markov processes*

Notations 9.2.4. In the following we write Id the *identity kernel* on $\mathbf{X} \to \mathbf{X}$, i.e. the Dirac delta function $\mathrm{Id}(x, \dot{x}) = \delta(x, \dot{x})$ s.t. $\int_A \mathrm{Id}(x, \dot{x}) = 1$ if $x \in A$ and $\int_A \mathrm{Id}(x, \dot{x}) = 0$ otherwise.

We will now define the execution in a similar way as for the graph and graphing cases above: one computes all alternating paths between two objects. This computation can in fact be obtained in two steps: composing the two objects, and then computing the set of all paths of the result. We start by defining the latter operation, that computes paths of arbitrary length.

Definition 9.2.1 (Iterated kernel) *Let κ be a sub-Markov kernel on $\mathbf{X} \times \mathbf{Y}$. For $k > 1$, we define the k-th iterated kernel:*

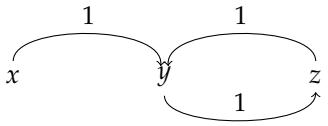
$$\kappa^{(k)}(x_0, \dot{x}_k) = \iint_{(x_1,\dots,x_{k-1}) \in (\mathbf{X} \cap \mathbf{Y})^{k-1}} \prod_{i=0}^{k-1} \kappa(x_i, \dot{x}_{i+1}).$$

By convention, $\kappa^{(1)} = \kappa$.

Definition 9.2.2 (Maximal paths – Execution kernel) *Let κ be a sub-Markov kernel on $\mathbf{X} \times \mathbf{Y}$. We define the* execution kernel *of κ as the map (in the formula, x_{n+1} is used as a notation for y):*

$$\begin{array}{rccl} \mathrm{tr}(\kappa) & : & X \backslash Y \times \mathcal{Y} \backslash \mathcal{X} & \to & [0,1] \\ & & (x,y) & \mapsto & \sum_{n \geqslant 1} \kappa^{(n)}(x,y). \end{array}$$

Remark 9.2.2 One could wonder why this is not defined on the whole space $X \times \mathcal{Y}$. The restriction is needed to define a sub-Markov kernel, something that can be understood on a very simple Markov chain:



On this figure, the partial sums of $\kappa^{(i)}(x, y)$ is a diverging series. This example also shows why the resulting kernel could be a sub-Markov kernel even when κ is a proper Markov kernel.

Lemma 9.2.5 *If κ is a sub-Markov kernel, $\mathrm{tr}^A(\kappa)$ is well-defined and a sub-Markov kernel.*

Now, the execution kernel just defined is the main operation for defining the execution of sub-Markov kernels.

Definition 9.2.3 *Given two sub-Markov kernels κ on $\mathbf{X} \times \mathbf{X}'$ and κ' on $\mathbf{Y} \times \mathbf{Y}'$, we define their execution $\kappa :: \kappa'$ as the kernel $\mathrm{tr}(\kappa \bullet \kappa')$ where:*

$$\kappa \bullet \kappa' = (\kappa + \mathrm{Id}_{\mathbf{Y} \backslash \mathbf{X}'}) \circ (\kappa' + \mathrm{Id}_{\mathbf{X}' \backslash \mathbf{Y}})$$

The reader with notions from *traced monoidal categories* [133, 136, 137] should not be surprised of this definition and the following properties[9].

Definition 9.2.4 *Three sub-Markov kernels κ on $\mathbf{X} \times \mathbf{X}'$, κ' on $\mathbf{Y} \times \mathbf{Y}'$, and κ'' on $\mathbf{Z} \times \mathbf{Z}'$ are said to be* in general position[10] *when the following condition is met:*

$$\mu(\mathbf{X}' \cap \mathbf{Y} \cap \mathbf{Z}) = \mu(\mathbf{Y}' \cap \mathbf{Z} \cap \mathbf{X}) = \mu(\mathbf{Z}' \cap \mathbf{X} \cap \mathbf{Y}) = 0,$$

[133]: Joyal et al. (1996), *Traced monoidal categories*
[136]: Hasegawa (1997), *Recursion from Cyclic Sharing: Traced Monoidal Categories and Models of Cyclic Lambda Calculi*
[137]: Haghverdi et al. (2006), *A categorical model for the geometry of interaction*

9: In fact, the execution kernel should define a trace in the categorical sense.

10: The reader will realise the terminology is inspired from algebraic geometry, but no formal connections should be expected.

$$\mu(\mathbf{X} \cap \mathbf{Y}' \cap \mathbf{Z}') = \mu(\mathbf{Y} \cap \mathbf{Z}' \cap \mathbf{X}') = \mu(\mathbf{Z} \cap \mathbf{X}' \cap \mathbf{Y}') = 0.$$

Note that if $\mathbf{X} = \mathbf{X}'$, $\mathbf{Y} = \mathbf{Y}'$ *and* $\mathbf{Z} = \mathbf{Z}'$, *the condition becomes* $\mu(\mathbf{X} \cap \mathbf{Y} \cap \mathbf{Z}) = 0$, *which is the condition of application of the associativity of execution and of the trefoil property in the graph case.*

Lemma 9.2.6 *Given three sub-Markov kernels* κ_0 *on* $\mathbf{X} \times \mathbf{X}'$, κ_1 *on* $\mathbf{Y} \times \mathbf{Y}'$, *and* κ_2 *on* $\mathbf{Z} \times \mathbf{Z}'$ *in general position:*

$$(\kappa_0 :: \kappa_1) :: \kappa_2 = \kappa_0 ::(\kappa_1 :: \kappa_2).$$

Lemma 9.2.7 *Given two sub-Markov kernels* κ *on* $\mathbf{X} \times \mathbf{X}'$ *and* κ' *on* $\mathbf{Y} \times \mathbf{Y}'$ *such that* $\mathbf{X} \cap \mathbf{Y} = \mathbf{X}' \cap \mathbf{Y}' = \emptyset$*:*

$$\kappa :: \kappa' = \kappa' :: \kappa.$$

This establishes the existence of a well-defined associative (and symmetric) execution, the first ingredient for constructing linear realisability models.

9.3. Trefoil Properties

We will now discuss the *trefoil* property, which I now also call the 2-cocycle property. This property relates a measurement of pairs of objects (graphs, matrices, graphings, operators, etc.) and the execution just defined. It states a sort of higher dimensional associativity, and is essential in the construction of linear realisability models (chapter 10).

Definition 9.3.1 *Let* (P, Ex) *be a set* P *together with an associative operation* $\mathrm{Ex} : P \times P \to P$. *A measurement* $[\![\cdot, \cdot]\!]_m : P \times P \to \Omega$ *satisfies the* trefoil *(or* 2-cocycle*) property if:*

$$[\![\mathrm{Ex}(A, B), C]\!]_m + [\![A, B]\!]_m = [\![A, \mathrm{Ex}(B, C)]\!]_m + [\![B, C]\!]_m.$$

Remark 9.3.1 When the measurement is symmetric, i.e. when $[\![A, B]\!]_m = [\![B, A]\!]_m$, this implies the original version of the property[17]:

[17]: Seiller (2016), *Interaction graphs: Additives*

$$\begin{aligned}[\![\mathrm{Ex}(A, B), C]\!]_m + [\![A, B]\!]_m &= [\![\mathrm{Ex}(C, A), B]\!]_m + [\![C, A]\!]_m \\ &= [\![\mathrm{Ex}(B, C), A]\!]_m + [\![B, C]\!]_m.\end{aligned}$$

This property was introduced when extending the Interaction graphs construction to additive connectives of linear logic [17]. It was in particular essential to define a quotient of the model that provided a solution to the usual failure of the geometry of interaction interpretation of additives. It turns out that it is in fact a generalisation of a property considered by Girard under the name *adjunction* (because it is in some way a low level equivalent of the duality $(A \otimes B^{\perp})^{\perp} = A \multimap B$, implying the monoidal closure of the corresponding categorical model of linear logic).

[17]: Seiller (2016), *Interaction graphs: Additives*

The introduction of the trefoil property also put into light the common general structure behind the different geometry of interaction constructions considered by Girard over the years. This will be explained later in this section.

Graphs and matrices

The first work in which the trefoil property was introduced [17] was written during my PhD and only concerned the case of graphs[11]. It was proved for the following measurement between graphs:

$$[\![F, G]\!]_m = \sum_{\pi \in \mathcal{C}(F,G)} m(\omega(\pi)),$$

where the sum was taken over 1-*circuits*, that is cycles[12] alternating between F and G that are not a proper power of another cycle, and m was an arbitrary map from circuits to $\mathbf{R}_+$. This is formalised in the following definitions.

I realised later on that the notion behind the terminology 1-circuit was in fact that of *prime closed path*, as considered when defining zeta functions of graphs for instance. I will therefore adopt from now on this more standard terminology.

Definition 9.3.2 *Given a graph G, a closed path π (called* circuit *in earlier work [16]) of length* $\mathrm{len}(\pi) = k$ *is a path $(e_i)_{i=0}^{k-1}$ such that $s^G(e_0) = t^G(e_{k-1})$ and considered up to cyclic permutations. A* prime closed path *(called* 1-*circuit in* IG*) is a closed path which is not a proper power of a smaller closed path. We denote by $\mathcal{C}(G)$ the set of prime closed paths in G.*

Definition 9.3.3 *Given graphs F, G, an* alternating closed path *π of length* $\mathrm{len}(\pi) = 2k$ *is a closed path $(e_i)_{0 \leqslant i \leqslant 2k-1}$ in $F \cup G$ such that for all $i \in \mathbf{Z}/2k\mathbf{Z}$, $e_i \in F$ if and only if $e_{i+1} \in G$. The set of prime alternating closed paths between F and G will be denoted $\mathcal{C}(F, G)$.*

Definition 9.3.4 *Let m be a map $\mathbf{C} \to \mathbf{R}_{\geqslant 0}$. For any two graphs F, G we define the measurement*

$$[\![F, G]\!]_m = \sum_{\pi \in \mathcal{C}(F,G)} m(\omega^{F \square G}(\pi)).$$

Now, I realised that any of these measurements satisfied the trefoil property, based on a very simple geometric argument. Under some mild hypothesis on the graphs, the trefoil property is obtained as a consequence of a geometric identity [9, 17]. We can establish that when $V^F \cap V^G \cap V^H = \emptyset$, there is weight-preserving bijection between the following sets of closed paths:

$$\mathcal{C}(F, \mathrm{Ex}(G, H)) \uplus \mathcal{C}(G, H) \equiv \mathcal{C}(G, \mathrm{Ex}(H, F)) \uplus \mathcal{C}(H, F). \qquad (9.2)$$

When restricting to the case $V^G \cap V^H = \emptyset$, $\mathrm{Ex}(G, H) = G + H$ is a disjoint union, and specific values of the measurement give the different adjunctions considered by Girard over the years:

[17]: Seiller (2016), *Interaction graphs: Additives*

11: In fact, it involved linear combinations of graphs, but this aspect can be overlooked here.

12: A cycle here is understood geometrically, i.e. we do not consider a fixed source vertex.

[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*

[9]: Seiller (2012), *Logique dans le facteur hyperfini : géometrie de l'interaction et complexité*

[17]: Seiller (2016), *Interaction graphs: Additives*

- if $m(x) = \infty$, then this implies that $F(G + H)$ is nilpotent if and only if $\mathrm{Ex}(F, H)G$ and FH are nilpotent (the adjunction used in the first three geometry of interaction constructions [13, 138, 139]);
- if $m(x) = 1$, this implies that $F(G + H)$ is cyclic if and only if $\mathrm{Ex}(F, H)G$ is cyclic when FH is nilpotent (the adjunction used in *multiplicatives* [140], the prequel to geometry of interaction);
- if $m(x) = -\log(1 - x)$, I showed how the measurement provides a combinatorial way to compute the determinant, recovering the adjunction based on

$$-\mathrm{ldet}(1 - F(G \oplus H))) = -\mathrm{ldet}(1 - \mathrm{Ex}(F, G)H)) - \mathrm{ldet}(1 - FG)),$$

  where ldet denotes the logarithm of the determinant, and which is used in the geometry of interaction in the hyperfinite factor[13]

[13]: Girard (1995), *Geometry Of Interaction III: Accommodating The Additives*
[138]: Girard (1989), *Geometry of Interaction I: Interpretation of System F*
[139]: Girard (1988), *Geometry of Interaction II: Deadlock-free Algorithms*
[140]: Girard (1987), *Multiplicatives*

In order to prove this last result, I relied on the following lemma [16].

Lemma 9.3.1 *Let A be a square matrix such that $\|A\| \leqslant 1$. Then, with the convention that $-\log(0) = \infty$,*

$$-\log(\det(1 - A)) = \sum_{k=1}^{\infty} \frac{\mathrm{Tr}(A^k)}{k}.$$

13: To be more precise, the latter uses the equivalent expression in which ldet denotes the logarithm of the *Fuglede-Kadison* determinant [10], but the result still holds [16].
[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*

I realised several years later that this result is a special case of the proof of rationality of the zeta function of graph. This lead me to establish a new measurement based on zeta functions presented below. However, before detailing these, I explain how the results on graphs can be extended to graphings.

Graphings cocycles

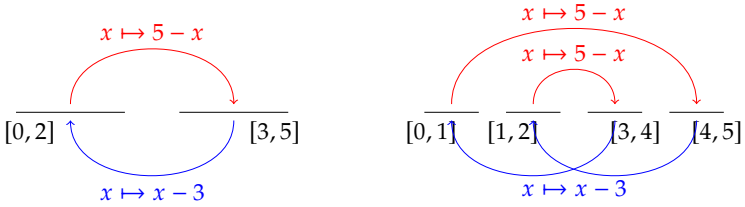
Now, the extension of the measurement to graphings will follow the identification of graphings as generalised graphs. In the same way we defined the execution of graphings by mimicking the execution of graphs, we will here define a measurement based on 1-circuits. We however encounter an issue, similar but more complex than that of splitting explained above.

More precisely, we want to define the measurement as a sum over 1-circuits, but those may behave in different ways with respect to 'splitting' edges, i.e. only accounting for cycles between two graphing representatives M and N will not lead to a measurement invariant w.r.t. the full equational theory. This is illustrated in Figure 9.2.

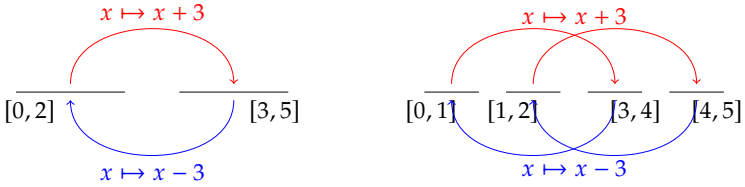
The solution is to take into account the action on the underlying space. Intuitively, one measures the cycles appearing in the underlying dynamical system instead of the graph-like representation. In the example, a given cycle becomes either one longer cycle or two disjoint cycles when splitting the space. But the reason behind this difference can be found in the maps realising the initial cycle. The measurement will thus split a 'formal' cycle, i.e. a cycle in the graph structure of the graphing representative, into several cycles depending on the length of the orbits.

Formally, I defined an abstract notion of *circuit-quantifying map* [18], i.e. a notion of measurement that behaves well with regards to the splitting

[18]: Seiller (2017), *Interaction Graphs: Graphings*



(a) A cycle doubling its length



(b) A cycle splitting in two cycles

Figure 9.2.: Examples of the evolution of a cycle when performing a refinement

$$\Sigma_{\pi=(e_i)_{i=1}^n \in \mathrm{Cy}(F,G)} \Sigma_{j=0}^{n} \int_{\mathrm{supp}(\pi)} \Sigma_{k=0}^{\rho_{\phi_\pi}(x)-1} \frac{m(\omega(\pi)^{\rho_{\phi_\pi}(\phi_\pi^k(x))})}{(n+1)\rho_{\phi_\pi}(x)\rho_{\phi_\pi}(\phi_\pi^k(x))} d(\phi_{e_n} \circ \phi_{e_{n-1}} \circ \cdots \circ \phi_{e_j})_* \lambda(x),$$

Figure 9.3.: Measurement for graphings

of cycles exemplified in Figure 9.2. I then defined a family of such maps in the restricted case of *bi-measurable* transformations on a *trefoil space*. More precisely, a trefoil space $\mathbf{X}$ is a second-countable Hausdorff space equipped with a Borel σ-algebra and a σ-additive Radon measure. A bimeasurable transformation on $\mathbf{X}$ is a measurable function $f : \mathbf{X} \to \mathbf{X}$ such that f maps open sets to open sets, and such that for all measurable set O, $\mu(f^{-1}(O)) = 0$ if and only if $\mu(O) = 0$.

In this case, one can define the measurement shown in Figure 9.3.

where $\mathrm{Cy}(F,G)$ is the set of prime cycles, $\mathrm{supp}(\pi)$ is the subspace on which the cycle is defined, $\rho_{\phi_\pi}(x)$ is the map associating to $x \in \mathrm{supp}(\pi)$ the length (possibly infinite) of the corresponding orbit, and $d\phi_*\lambda$ is the pushforward measure of λ along ϕ. While this expression is quite involved, it can be understood as computing a sum over all finite closed orbits (this point of view will be presented when explaining the connection with zeta functions later in this chapter).

This measurement can then be shown to satisfy the trefoil property [18, Theorem 35].

Theorem 9.3.2 *Let F, G, H be graphings such that no point x belongs to the source of edges from all three graphings (i.e. $V^F \cap V^G \cap V^H = \emptyset$ if one write V^A the set $x \in \mathbf{X}$ such that x belongs to the domain of at least one edge of the graphing A). Then:*

$$[\![F, \mathrm{Ex}(G,H)]\!]_m + [\![G,H]\!]_m = [\![\mathrm{Ex}(F,G), H]\!]_m + [\![F,G]\!]_m.$$

Moreover, the measurement is symmetric, i.e. $[\![F,G]\!]_m = [\![G,F]\!]_m$.

9.4. Zeta cocycles

Bowen-Lanford Zeta Functions

We first recall the definition and some properties of the zeta function of a directed graph. We refer to the book of Terras [141] for more details. We will later on continue with zeta functions for weighted directed graphs, and further with zeta functions for dynamical systems. The graph case is important as it provides intuitions about the later generalisations.

[141]: Terras (2010), *Zeta functions of graphs: a stroll through the garden*

In this subsection only, we consider non-weighted directed graphs (i.e. there is no weight map ω^G or, equivalently, this map is the constant map equal to 1) and suppose they are *simple*, i.e. that the map $E^G \mapsto V^G \times V^G; e \mapsto (s^G(e), t^G(e))$ is injective. Given such a graph, its *transition matrix* is defined as the $V^G \times V^G$ matrix whose coeficients are defined by $M_G(v, v') = 1$ if there is an edge $e \in E^G$ such that $s^G(e) = v$ and $t^G(e) = v'$, and $M_G(v, v') = 0$ otherwise. The following definition provides a clear parallel with the famous Euler zeta function.

Definition 9.4.1 *The Bowen-Lanford zeta function associated with the graph G is defined as:*

$$\zeta_G(z) = \prod_{\tau \in \mathcal{C}(G)} (1 - z^{\text{len}(\tau)})^{-1}$$

which converges provided $|z|$ is sufficiently small.

The two following lemmas are easy to establish (using the identity $\log(1-x) = \sum_{k=1}^{\infty} \frac{x^n}{n}$). The first is essential in our work, as it provides an alternative expression of the zeta function that we will be able to generalise later. Indeed, while the formal definition above uses the notion of prime closed paths, this one quantifies over all closed paths.

The second lemma is key to the representation of $\zeta_G(z)$ as a rational function. This relates the zeta function with the determinant of the adjacency matrix of G.

Lemma 9.4.1 *Let $N(n)$ denote the number of all possible strings $(v_1, \ldots, v_n)$ representing a closed path in G of length n. Then $\zeta_G(z) = \exp\left(\sum_{i=1}^{\infty} \frac{z^n}{n} N(n)\right)$.*

Lemma 9.4.2 *Let G be a graph and $M(G)$ its* transition matrix, *then $\text{tr}(M(G)^k) = N(k)$.*

The previous lemmas are standard results from the theory of Zeta functions. Together, they yield the following result.

Proposition 9.4.3 *Let G be a graph, $M(G)$ its* transition matrix*:*

$$\log(\zeta_G(z)) = -\log(\det(1 - z.M(G))),$$

for sufficiently small values of $|z|$.

Zeta functions of weighted directed graphs

Now, we consider weighted directed graphs, i.e. graphs with weights of the edges, and we will restrict to the case of complex numbers as weights. We write ω the weight function, as well as its extension to paths, using the product, i.e.

$$\omega(\pi) = \prod_{e \in \pi} \omega(e).$$

Similarly to the case of unweighted graphs, we define the *transition matrix* of a *simple* weighted graph as the $V^G \times V^G$ matrix with $M_G(v, v') = \omega(e)$ if there exists a (necessarily unique) edge $e \in E^G$ with $\langle s^G(e), t^G(e)\rangle = (v, v')$, and $M_G(v, v') = 0$ otherwise.

For a general (i.e. non-simple) weighted graph G, we write $G(v, v')$ the set $\{e \in E^G \mid s^G(e) = v, t^G(e) = v'\}$. One can then extend the definition of *transition matrix* by associating to G the $V^G \times V^G$ matrix with $M_G(v, v') = \sum_{e \in G(v,v')} \omega(e)$. Alternatively, this matrix can also be defined as $M_{\hat{G}}$ where $\hat{G}$ is the *simple collapse* of G, i.e. the simple graph defined as $\hat{G} = (V^G, \hat{E}^G, \hat{s}^G, \hat{t}^G, \hat{\omega}^G)$ with:

- $\hat{E}^G = \{(v, v') \in V^G \times V^G \mid G(v, v') \neq \emptyset\}$,
- $\hat{s}^G((v, v')) = v$,
- $\hat{t}^G((v, v')) = v'$,
- $\hat{\omega}^G((v, v')) = \sum_{e \in G(v,v')} \omega(e)$.

Note that I proved in earlier work [16] that the measurement defined from the function $m := \lambda x.\log(1 - x)$ satisfies $[\![F, G]\!]_m = [\![\hat{F}, \hat{G}]\!]_m$.

[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*

The zeta function of a weighted graph is defined as follows.

Definition 9.4.2 *The zeta function associated with the weighted graph G is defined as:*

$$\zeta_G(z) = \prod_{\pi \in \mathcal{C}(G)} (1 - \omega(\pi).z)^{-1}$$

which converges provided $|z|$ is sufficiently small.

Readers familiar with zeta functions of weighted graphs will notice that we take the product of the weights to define the weight ω of a path, while standard work on zeta functions for weighted graphs define the weight ν of a path as a sum. This is formally explained by taking a logarithm, i.e. $\omega = \log \circ \nu$, leading to multiplying here expressions $1 - \omega(\pi)z$ instead of $1 - z^{\nu(\pi)}$ in the standard definition.

Adapting the proof of the non-weighted case (Proposition 9.4.3), one obtains the following general result, which extends the combinatorial interpretation of the determinant $\det(1 - M(G))$ mentioned above.

Proposition 9.4.4 *Let G be a directed weighted graph, $M(G)$ its* transition matrix*:*

$$\log(\zeta_G(z)) = -\log(\det(1 - z.M(G))),$$

for sufficiently small values of $|z|$.

Taking the logarithm we obtain:

$$\log(\zeta_G(z)) = \sum_{\pi \in \mathcal{C}(G)} -\log(1 - \omega(\pi).z)),$$

an expression that appears in the definition of measurement in the previous section. This can be used to relate the measurement defined in interaction graphs for $m := \lambda x.\log(1-x)$ with the value of the zeta function at $z = 1$:

$$[\![F, G]\!]_{\lambda x.\log(1-x)} = \log(\zeta_{F \bullet G}(1))$$

where the $\bullet$ operation consists in composing (i.e. taking length-2 paths) the graphs $F + \mathrm{Id}_{V^F \setminus V^G}$ and $G + \mathrm{Id}_{V^G \setminus V^F}$.

Orthogonality in IG models is defined by $F \perp G$ as $[\![F, G]\!]_m \neq 0, \infty$, i.e. $-\log(\zeta_{F \bullet G}(1)) \neq 0, \infty$. Through this previous result, this is equivalent to the fact that $\zeta_{F \bullet G}(1) \neq 0, 1$. We will now build on this remark to extend the construction of IG models. This provides a new family of models using zeta functions to define the orthogonality.

Zeta, Execution and a Cocycle Property

The geometric identity (Equation 9.2) can be used to establish a general cocycle condition satisfied by zeta functions.

Theorem 9.4.5 *Suppose $V^F \cap V^G \cap V^H = \emptyset$. Then:*

$$\zeta_{F \bullet (G :: H)}(z).\zeta_{G \bullet H}(z) = \zeta_{G \bullet (H :: F)}(z).\zeta_{H \bullet F}(z). \tag{9.3}$$

Using the fact we noticed earlier that $[\![F, G]\!]_{\lambda x.\log(1-x)} = -\log(\zeta_{F \bullet G}(1))$, we recover the trefoil property for graphs exposed in the previous section (in the case of $m(x) = -\log(1-x)$) by taking $z = 1$.

Zeta Functions for dynamical systems

The Ruelle zeta function [23] is defined from a function $f : M \to M$ where M is a manifold and a function $\phi : M \to \mathfrak{M}_k$ a matrix-valued function. We write $\mathrm{Fix}(g)$ the set of fixed points of g. Then the Ruelle zeta function is defined as (we suppose that $\mathrm{Fix}(f^k)$ is finite for all k):

[23]: Ruelle (1976), *Zeta-functions for expanding maps and Anosov flows*

$$\zeta_{f,\Phi}(z) = \exp\left(\sum_{m \geqslant 1} \frac{z^m}{m} \sum_{x \in \mathrm{Fix}(f^m)} \mathrm{tr}\left(\prod_{i=0}^{m-1} \phi(f^i(x))\right)\right).$$

For $d = 1$ and $\phi = 1$ the constant function equal to 1, this is the Artin-Mazur [142] zeta function:

[142]: Artin et al. (1965), *On periodic points*

$$\zeta_{f,1}(z) = \exp\left(\sum_{m \geqslant 1} \frac{z^m}{m} \mathrm{Card}(\mathrm{Fix}(f^m))\right).$$

Since we work with measured spaces, we consider the following measured variant of Ruelle's zeta function (defined for measure-preserving maps). Suppose we work with a measured space $(M, \mathcal{B}, \mu)$ and that $\mathrm{Fix}(f^m)$ is of finite measure:

$$\zeta_{f,\Phi}(z) = \exp\left(\sum_{m \geqslant 1} \frac{z^m}{m} \int_{\mathrm{Fix}(f^m)} \mathrm{tr}\left(\prod_{i=0}^{m-1} \phi(f^i(x))\right) d\mu(x)\right)$$

For $d = 1$ and $\phi = 1$, this becomes:

$$\zeta_{f,1}(z) = \exp\left(\sum_{m \geqslant 1} \int_{\mathrm{Fix}(f^m)} \frac{z^m}{m}\right)$$

which we relate to the measurement on graphings defined above in Figure 9.3 in the case of non-singular measure-preserving maps (NSMP).

Proposition 9.4.6 *Given NSMP partial dynamical systems $f, g : \mathbf{X} \to \mathbf{X}$, for all constant c we have:*

$$[\![f, g]\!]_{\lambda x.c} = \log(\zeta_{g \circ f,1}(c)),$$

with $[\![\_, \_]\!]_m$ the standard measurement on graphings [18].

[18]: Seiller (2017), *Interaction Graphs: Graphings*

Proof. On one hand, we have

$$-\log(\zeta_{g \circ f,1}(1)) = \sum_{m \geqslant 1} \int_{\mathrm{Fix}((g \circ f)^m)} \frac{1}{m} \sum_{m \geqslant 1} \frac{\mu(\mathrm{Fix}((g \circ f)^m))}{m}.$$

On the other hand, the measurement $[\![f, g]\!]_m$ defined on general graphings [18, Definitions 37 and 57] is given by the formula shown in Figure 9.3 in which ρ_ϕ is a measurable map associating to each point the length of the orbit it belongs to [16, Corollary 45], $\mathrm{Cy}(f, g)$ denotes the set of prime closed paths alternating between f and g, and generally $h_*\mu$ denotes the pullback measure of μ along h.

This expression simplifies in the measure-preserving case [18, Proposition 52], and can be expressed as

$$[\![f, g]\!]_m = \sum_{\pi = e_0 \dots e_n \in \mathrm{Cy}(f,g)} \int_{\mathrm{supp}(\pi)} \frac{m(\omega(\pi)^{\rho_{\phi_\pi}(x)})}{\rho_{\phi_\pi}(x)}$$

Now, we can split this expression by considering the partition of $\mathrm{supp}(\pi)$ given by the preimage of ρ_ϕ. I.e. this partitions $\mathrm{supp}(\pi)$ into (measurable) subsets $S_i^\pi = \rho_\phi^{-1}(\mathrm{supp}(\pi))$ containing the points $x \in \mathrm{supp}(\pi)$ such that the orbit of x is of length i.

As the value of ρ_ϕ is constant on these sets, this gives:

$$[\![f, g]\!]_m = \Sigma_{\pi = e_0 \dots e_n \in \mathrm{Cy}(f,g)} \Sigma_{i=0}^{\infty} \int_{S_i^\pi} \frac{m(\omega(\pi)^i)}{i}$$

Now, we are considering the case where $m(x) = z$, and we know all weights in the graphing are equal to 1. Hence:

$$[\![f, g]\!]_m = \Sigma_{\pi \in \mathrm{Cy}(f,g)} \Sigma_{i=0}^{\infty} \int_{S_i^\pi} \frac{z}{i}.$$

On the other hand, we have that, writing $\mathrm{AltCycle}(f, g)_m$ the set of all alternating cycle between f and g of length m:

$$\begin{aligned} \log(\zeta_{g \circ f,1}(z)) &= \Sigma_{m \geq 1} \int_{\mathrm{Fix}((g \circ f)^m)} \frac{z}{m} \\ &= \Sigma_{m \geq 1} \Sigma_{\pi \in \mathrm{AltCycle}(F,G)_m} \int_{S_m^\pi} \frac{z}{m} \end{aligned}$$

since each fixpoint belongs to exactly one alternating cycle of length m between f and g (because the graphings are deterministic).

Now each alternating cycle of length m between f and g can be written uniquely as a product of alternating prime cycles, we deduce:

$$\begin{aligned}\log(\zeta_{g\circ f,1}(z)) &= \sum_{m\geq 1}\sum_{\pi\in \mathrm{Cy}(f,g)}\int_{S_m^\pi}\frac{z}{m}\\ &= \sum_{\pi\in \mathrm{Cy}(f,g)}\sum_{m\geq 1}\int_{S_m^\pi}\frac{z}{m}\\ &= [\![f,g]\!]_m\end{aligned}$$

This is the equality we wanted to prove. ❀

Zeta functions for sub-Markov processes

We now define the zeta function for sub-Markov kernels which I introduced [135]; to my knowledge, this was not studied elsewhere. For this, we first define a map which we call the 'zeta kernel'.

[135]: Seiller (2022), *Zeta functions and the (linear) logic of Markov processes*

Definition 9.4.3 (Finite orbits – Zeta kernel) *Let κ be a sub-Markov kernel on $\mathbf{X}\times\mathbf{Y}$. The* zeta kernel*, or kernel of finite orbits of κ is a kernel on $\mathbf{X}\times\mathbf{N}$ – where $\mathbf{N}$ denotes the set of natural numbers – defined as:*

$$\zeta_\kappa(x_0,\dot{x}_0,n) = \iint_{(x_1,\dots,x_{n-1})\in(\mathbf{X}\cap\mathbf{Y})^{n-1}}\prod_{i\in\mathbf{Z}/n\mathbf{Z}}\kappa(x_i,\dot{x}_{i+1}).$$

This expression computes the probability that a given point x_0 lies in an orbit of length n. It is a sub-Markov kernel for each fixed value of n, but the sum over $n\in\mathbf{Z}$ is not. The reason is simple: if a point x lies in a length 2 orbit with probability 1 (e.g. the point y in the example Markov chain in Remark 9.2.2), then it lies in a length $2k$ orbit with probability 1 as well. However, let us remark that the expression

$$\int_{x\in X\cap Y}\zeta_\kappa(x,\dot{x},n)$$

plays the role of the set $\mathrm{Fix}(f^n)$ that appears in dynamical and graph zeta functions.

Definition 9.4.4 (Zeta function) *We now define the Zeta function associated with a sub-Markov kernel κ on $\mathbf{X}\times\mathbf{Y}$:*

$$\zeta_\kappa(z) : z\mapsto \exp\left(\sum_{n=1}^{\infty}\frac{z^n}{n}\int_{x\in\mathbf{X}\cap\mathbf{Y}}\zeta_\kappa(x,\dot{x},n)\right)$$

We can then establish the trefoil property. Following what was presented in the first sections, we now define a zeta function associated to pairs of general sub-Markov processes, and show it satisfies the required cocycle property w.r.t. execution.

Definition 9.4.5 *Given two kernels κ,κ', we define their* zeta-measurement *$\zeta_{\kappa,\kappa'}$ as the function $\zeta_{\kappa\bullet\kappa'}(z)$.*

Proposition 9.4.7 (Cocycle) *Given three sub-Markov kernels* κ *on* $\mathbf{X} \times \mathbf{X}'$, κ' *on* $\mathbf{Y} \times \mathbf{Y}'$, *and* κ'' *on* $\mathbf{Z} \times \mathbf{Z}'$ *in general position:*

$$\zeta_{\kappa,\kappa'}(z)\zeta_{\kappa :: \kappa',\kappa''}(z) = \zeta_{\kappa' :: \kappa'',\kappa}(z)\zeta_{\kappa',\kappa''}(z)$$

In the next chapter, we will explain how to construct linear realisability models from a set of machines equipped with an associative execution and a measurement satisfying the trefoil property. We will then explain how all the above (and other) situations give rise to models of (fragments of) linear logic.

10. Linear realisability

In this chapter, we explain how the properties presented in the previous chapter can be used to obtain models of linear logic. Linear logic is a refinement of Intuitionistic and classical logic in which the implication $A \Rightarrow B$ is decomposed as $!A \multimap B$, where $\multimap$ is a linear implication (using exactly once its argument) and ! a modality allowing to duplicate a formula, i.e. allowing to deduce $!A \otimes !A$ from $!A$.

One important aspect is that this decomposition provides a finer control on the management of 'resources', i.e. what proof theory calls *structural rules*. These rules usually translate trivial principles of logic, such as weakening: if A implies C, then $A \wedge B$ imply C. Since linear logic allows these rules only on formulas with the ! modality (or its dual ?), then some basic properties of proof systems are no longer satisfied. In particular, two non-equivalent conjunctions arise: one verifying that $A \Rightarrow C$ and $B \Rightarrow C$ implies that $A \wedge B \Rightarrow C$ – the additive conjunction & corresponding categorically to a cartesian product – and the other satisfying that $A \Rightarrow C$ and $B \Rightarrow D$ implies $A \wedge B \Rightarrow C \wedge D$ – the multiplicative conjunction $\otimes$ corresponding categorically to a (symmetric) monoidal product. While the two conjunctions are equivalent in intuitionistic and classical logic, through structural rules, they are no longer equivalent once one restricts to linear implications. The modality !, which transforms the additive conjunction into a multiplicative one since it satisfies $!(A \,\&\, B) \cong !A \otimes !B$, is called an *exponential* connective.

10.1. The general setting

The technique behind the construction of geometry of interaction, ludics, interaction graphs, are based upon a general method. Here we abstract away the mathematical underpinnings. All these constructions are based upon a notion of abstract programs sharing the common structure exposed in the previous chapter. We explain in a systematic and axiomatic way now how this structure can be exploited to define linear realisability models for second-order multiplicative additive linear logic (MALL). The interpretation of larger fragments (namely exponentials of linear logic, or variants of linear logic) requires some additional assumptions, and is possible in specific models (most of them as it turns out).

Definition 10.1.1 *A multiplicative linear realisability situation is a tuple* $(P, \mathrm{Ex}, [\![\cdot,\cdot]\!]_m)$ *where* P *is a set,* $\mathrm{Ex} : P \times P \to P$ *is an abstract notion of execution,* $[\![\cdot,\cdot]\!]_m : P \times P \to \Theta$ *is a measurement in a commutative group* Θ*, such that:*

- *Associativity of execution:*

$$\mathrm{Ex}(\mathrm{Ex}(p_1, p_2), p_3) = \mathrm{Ex}(p_1, \mathrm{Ex}(p_2, p_3)),$$

- *Trefoil property / 2-cocycle:*

$$[\![\mathrm{Ex}(p_1, p_2), p_3]\!]_m + [\![p_1, p_2]\!]_m = [\![p_1, \mathrm{Ex}(p_2, p_3)]\!]_m + [\![p_2, p_3]\!]_m.$$

Definition 10.1.2 *A project* $\mathfrak{p}$ *is a pair* (a, p) *where* $a \in \Theta$ *and* $p \in P$*. The measurement is extended to projects in the following way:*

$$[\![(a,p),(a,p')]\!]_m = a + a' + [\![p,p']\!]_m.$$

The execution is extended to projects by:

$$\mathrm{Ex}((a,p),(a,p')) = (a + a' + [\![p,p']\!]_m, \mathrm{Ex}(p,p')).$$

Remark 10.1.1 While writing this document, as part of a collaboration with Juan-Luis Gastaldi, Luc Pellissier and John Terrilla, I realised that an alternative construction can be performed. Indeed, if one defines the following extensions of measurement and execution:

$$\begin{aligned}[\![(a,p),(a,p')]\!]_m &= a + a' - [\![p,p']\!]_m \\ \mathrm{Ex}((a,p),(a,p')) &= (a + a' - [\![p,p']\!]_m, \mathrm{Ex}(p,p')),\end{aligned}$$

then it can be shown that the results exposed later in this section hold: these definitions give rise to a linear realisability model. This appeared while studying the model obtained when considering a set of programs P the strings over an alphabet Σ, equipped with concatenation for execution and a well-chosen measurement depending on a corpus. In this particular case, we believe that the notion of type can account for syntactic (and maybe semantic) constructs of natural language, extending and generalising preliminary investigation of Bradley, Gastaldi, and Terrilla [143].

[143]: Bradley et al. (2024), *The structure of meaning in language*

As a consequence of these definitions, the trefoil (or 2-cocycle) property simplifies to a simple cyclic property. We note that commutativity of Θ is essential in the following proof.

Proposition 10.1.1 *Let* $\mathfrak{p}, \mathfrak{q}, \mathfrak{r}$ *be three projects. Then*

$$[\![\mathrm{Ex}(\mathfrak{p},\mathfrak{q}),\mathfrak{r}]\!]_m = [\![\mathfrak{p},\mathrm{Ex}(\mathfrak{q},\mathfrak{r})]\!]_m.$$

Proof. This is simple calculation. We write $\mathfrak{p} = (a,p)$, $\mathfrak{q} = (b,q)$, $\mathfrak{r} = (c,r)$. Then:

$$\begin{aligned}[\![\mathrm{Ex}(\mathfrak{p},\mathfrak{q}),\mathfrak{r}]\!]_m &= [\![(a + b + [\![p,q]\!]_m, \mathrm{Ex}(p,q)),(c,r)]\!]_m \\ &= a + b + c + [\![p,q]\!]_m + [\![\mathrm{Ex}(p,q),r]\!]_m \\ &= a + b + c + [\![q,r]\!]_m + [\![p,\mathrm{Ex}(q,r)]\!]_m \\ &= [\![(a,p),(b + c + [\![q,r]\!]_m, \mathrm{Ex}(q,r))]\!]_m \\ &= [\![\mathfrak{p},\mathrm{Ex}(\mathfrak{q},\mathfrak{r})]\!]_m.\end{aligned}$$

❀

Definition 10.1.3 (Antipode) *An* antipode *is a subset* $\aleph$ *of* Θ*.*

Definition 10.1.4 *Two projects* $\mathfrak{p}$ *and* $\mathfrak{q}$ *are* orthogonal w.r.t. the antipode $\aleph$*, noted* $\mathfrak{p} \perp_\aleph \mathfrak{q}$ *when* $[\![\mathfrak{p},\mathfrak{q}]\!]_m \in \aleph$*.*

Remark 10.1.2 In most constructions, this definition is coherent, though weaker, than the definition of pole in classical realisability. Indeed, it is true in general that the orthogonality can be equivalently defined by a set of specific programs $P_\aleph$ as follows: $\mathfrak{p} \perp_\aleph \mathfrak{q}$ if and only if $\mathrm{Ex}(\mathfrak{p},\mathfrak{q}) \in P_\aleph$.

When this is the case, the set of pairs of programs such that $\mathfrak{p} \perp_{\aleph} \mathfrak{p}'$ is a pole in the sense of Krivine: it is a set of "processes" (here: a program $\mathfrak{p}$ and a single-element stack $\mathfrak{p}'$) which is closed under anti-reduction[1].

1: We do not go into details here, but the reduction is a partial execution, i.e. execution w.r.t. a subset of the cut formulas.

We now define the notion of type based on this orthogonality relation. Again, this is close to Krivine's definition in terms of *truth values* and *falsity values*, but as an effect of linearity the stacks contain exactly one term, leading to an identification of stacks and terms. This, together with the fact that (in most cases) the notion of orthogonality is symmetric, implies that we have a single notion of type accounting for both truth and falsity values.

We note however that in the general framework, I have not explicitly asked for symmetry. As a consequence, one should distinguish between left- and right- types. This will not be detailed below, but all statements can ba adapted to distinguish between those.

Definition 10.1.5 *A type is a set* $\mathbf{A}$ *of projects such that* $\mathbf{A} = T^{\perp}$ *for some set of projects* T*. We will often refer to such a set* T *(which is not unique in general) as* a set of tests for $\mathbf{A}$.

Remark 10.1.3 A standard argument establishes that a set is a type if and only if $\mathbf{A} = \mathbf{A}^{\perp\perp}$.

While this is not detailed here, it is important to note that this notion of types as descriptors differs in nature from the standard understanding as types as constraints [61]. It is moreover related to work by Birkhoff on lattice theory, and the related notion of formal concepts introduced in computer science by Wille [63], but extends those by taking into account the dynamic aspect of computation.

[61]: Joinet et al. (2021), *From abstraction and indiscernibility to classification and types: revisiting Hermann Weyl's theory of ideal elements*

[63]: Wille (1982), *Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts*

10.2. Multiplicative-Additive linear logic

Implicative Linear Logic

We note that in this general setting, we do not impose that $[\![\cdot,\cdot]\!]_m$ is symmetric. While this is the case of all examples below, it turns out the assumption is not needed. A non-symmetric measurement however imposes a distinction between right and left types. As a consequence, one would have to carefully distinguish between $A^{\perp\perp}$, ${}^{\perp}(A^{\perp})$, $({}^{\perp}A)^{\perp}$, and ${}^{\perp\perp}A$. As mentioned above, we leave it to the reader to adapt the statements in this general setting and state, for simplicity, the results without making this distinction.

Definition 10.2.1 *Given types* $\mathbf{A}$ *and* $\mathbf{B}$ *we define:*

$$\mathbf{A} \multimap \mathbf{B} = \{\mathfrak{p} \mid \forall \mathfrak{a} \in \mathbf{A}, \mathrm{Ex}(\mathfrak{p}, \mathfrak{a}) \in \mathbf{B}\},$$
$$\mathbf{A} :: \mathbf{B} = \{\mathrm{Ex}(\mathfrak{a}, \mathfrak{b}) \mid \forall \mathfrak{a} \in \mathbf{A}, \forall \mathfrak{b} \in \mathbf{B}\}^{\perp\perp}.$$

Proposition 10.2.1 *For all types* $\mathbf{A}$ *and* $\mathbf{B}$*, the set* $\mathbf{A} \multimap \mathbf{B}$ *is a type, and:*

$$\mathbf{A} \multimap \mathbf{B} = (\mathbf{A} :: \mathbf{B}^{\perp})^{\perp}.$$

Proof. Take $\mathfrak{p} \in (\mathbf{A} :: \mathbf{B}^\perp)^\perp$. Since $\mathbf{A}^{\perp\perp\perp} = \mathbf{A}^\perp$, this implies that $\mathfrak{p} \in \{\mathrm{Ex}(\mathfrak{a}, \mathfrak{b}') \mid \forall \mathfrak{a} \in \mathbf{A}, \forall \mathfrak{b}' \in \mathbf{B}^\perp\}^\perp$. Hence, for all $\mathfrak{a} \in \mathbf{A}$ and all $\mathfrak{b}' \in \mathbf{B}^\perp$, $[\![\mathfrak{p}, \mathrm{Ex}(\mathfrak{a}, \mathfrak{b}')]\!]_m \in \perp\!\!\!\perp$. By the cyclic property:

$$[\![\mathfrak{p}, \mathrm{Ex}(\mathfrak{a}, \mathfrak{b}')]\!]_m \in \perp\!\!\!\perp \Leftrightarrow [\![\mathrm{Ex}(\mathfrak{p}, \mathfrak{a}), \mathfrak{b}']\!]_m \in \perp\!\!\!\perp.$$

Hence $\mathrm{Ex}(\mathfrak{p}, \mathfrak{a}) \perp \mathfrak{b}'$ for all $\mathfrak{a} \in \mathbf{A}$ and all $\mathfrak{b}' \in \mathbf{B}^\perp$. I.e. $\mathfrak{p} \in \mathbf{A} \multimap \mathbf{B}$. ❀

Additive conjunction

Defining additive connectives requires an extension of the framework in which we assume that Θ is a semiring.

Definition 10.2.2 *A general linear realisability situation is defined as a tuple* $(P, \mathrm{Ex}, [\![\cdot,\cdot]\!]_m, \mathbf{1})$ *where* P *is a set,* $\mathrm{Ex} : P \times P \to P$ *is an abstract notion of execution,* $[\![\cdot,\cdot]\!]_m : P \times P \to \Theta$ *is a measurement in a semiring* Θ*, and* $\mathbf{1} : P \to \Theta$ *satisfies* $\mathbf{1}_{\mathrm{Ex}(p,p')} = \mathbf{1}_p \mathbf{1}_{p'}$*, and such that:*

- *Associativity of execution:*

$$\mathrm{Ex}(\mathrm{Ex}(p_1, p_2), p_3) = \mathrm{Ex}(p_1, \mathrm{Ex}(p_2, p_3)),$$

- *Trefoil property / 2-cocycle:*

$$[\![\mathrm{Ex}(p_1, p_2), p_3]\!]_m + [\![p_1, p_2]\!]_m \mathbf{1}_{p_3} = [\![p_1, \mathrm{Ex}(p_2, p_3)]\!]_m + \mathbf{1}_{p_1} [\![p_2, p_3]\!]_m.$$

We can define a general linear realisability situation from any linear realisability situation over a commutative[2] semiring Θ as follows. Instead of simply considering elements of P, we work with formal linear combinations of elements of P, i.e. in $\Theta[P]$, the free Θ-module over P. The execution and measurement extend to $\Theta[P]$ by letting:

- $\mathrm{Ex}^\Theta(\sum_i \alpha_i p_i, \sum_j \beta_j q_j) = \sum_{i,j} \alpha_i \beta_j \mathrm{Ex}(p_i, q_j)$,
- $[\![\sum_i \alpha_i p_i, \sum_j \beta_j q_j]\!]_m^\Theta = \sum_{i,j} \alpha_i \beta_j [\![p_i, q_j]\!]_m$.

We define $\mathbf{1}_{\sum_i \alpha_i p_i} = \sum_i \alpha_i$.

2: Commutativity of the product is required to obtain Lemma 10.2.4, but not necessary to define non-reversible connectives.

Lemma 10.2.2 *If* $(P, \mathrm{Ex}, [\![\cdot,\cdot]\!]_m)$ *is a multiplicative linear realisability situation, the tuple* $(\Theta[P], \mathrm{Ex}^\Theta, [\![\cdot,\cdot]\!]_m^\Theta)$ *is a general linear realisability situation.*

Proof. We first check that the extended notion of execution is associative.

$$\begin{aligned}
&\mathrm{Ex}^\Theta(\mathrm{Ex}^\Theta(\sum_i \alpha_i p_i, \sum_j \beta_j q_j), \sum_k \gamma_k r_k) \\
&\quad = \mathrm{Ex}^\Theta(\sum_{i,j} \alpha_i \beta_j \mathrm{Ex}(p_i, q_j), \sum_k \gamma_k r_k) \\
&\quad = \sum_{i,j,k} \alpha_i \beta_j \gamma_k \mathrm{Ex}(\mathrm{Ex}(p_i, q_j), r_k) \\
&\quad = \sum_{i,j,k} \alpha_i \beta_j \gamma_k \mathrm{Ex}(p_i, \mathrm{Ex}(q_j, r_k)) \\
&\quad = \mathrm{Ex}^\Theta(\sum_i \alpha_i p_i, \sum_{j,k} \beta_j \gamma_k \mathrm{Ex}(q_j, r_k)) \\
&\quad = \mathrm{Ex}^\Theta(\sum_i \alpha_i p_i, \mathrm{Ex}^\Theta(\sum_j \beta_j q_j, \sum_k \gamma_k r_k)).
\end{aligned}$$

We then verify that the trefoil property holds.

$$\begin{aligned}
&[\![\mathrm{Ex}^{\Theta}(\sum_i \alpha_i p_i, \sum_j \beta_j q_j), \sum_k \gamma_k r_k]\!]^{\Theta}_m + [\![\sum_i \alpha_i p_i, \sum_j \beta_j q_j]\!]^{\Theta}_m \mathbf{1}_\gamma \\
&\quad = [\![\sum_{i,j} \alpha_i\beta_j \mathrm{Ex}(p_i, q_j), \sum_k \gamma_k r_k]\!]^{\Theta}_m + (\sum_{i,j} \alpha_i\beta_j [\![p_i, q_j]\!]_m)\mathbf{1}_\gamma \\
&\quad = \sum_{i,j,k} \alpha_i\beta_j\gamma_k([\![\mathrm{Ex}(p_i, q_j), r_k]\!]_m + [\![p_i, q_j]\!]_m) \\
&\quad = \sum_{i,j,k} \alpha_i\beta_j\gamma_k([\![p_i, \mathrm{Ex}(q_j, r_k)]\!]_m + [\![q_j, r_k]\!]_m) \\
&\quad = [\![\sum_i \alpha_i p_i, \sum_{j,k} \beta_j\gamma_k \mathrm{Ex}(q_j, r_k)]\!]^{\Theta}_m + (\sum_i \alpha_i)(\sum_{j,k} \beta_j\gamma_k [\![q_j, r_k]\!]_m) \\
&\qquad = [\![\sum_i \alpha_i p_i, \mathrm{Ex}^{\Theta}(\sum_j \beta_j q_j, \sum_k \gamma_k r_k)]\!]^{\Theta}_m + (\sum_i \alpha_i)[\![\sum_j \beta_j q_j, \sum_k \gamma_k r_k)]\!]^{\Theta}_m.
\end{aligned}$$

❀

We can now define additive projects following the constructions from the previous section.

Definition 10.2.3 *An additive project* $\mathfrak{p}$ *is a pair* $(a, \sum_i \alpha_i p_i)$ *of an element of* Θ *and an element of* $\Theta[P]$. *Execution and measurement are defined as follows on additive projects:*

$$\begin{aligned}
&\mathrm{Ex}((a, \sum_i \alpha_i p_i), (b, \sum_j \beta_j q_j)) \\
&\qquad = (a\mathbf{1}_\beta + \mathbf{1}_\alpha b + \sum_{i,j} \alpha_i\beta_j [\![p_i, q_j]\!]_m, \mathrm{Ex}(\sum_i p_i, \sum_j q_j)), \\
&[\![(a, \sum_i \alpha_i p_i), (b, \sum_j \beta_j q_j)]\!]_m \\
&\qquad = a\mathbf{1}_\beta + \mathbf{1}_\alpha b + \sum_{i,j} \alpha_i\beta_j [\![p_i, q_j]\!]_m.
\end{aligned}$$

A consequence of these definitions is the cyclic property on additive projects, generalising Proposition 10.1.1.

Proposition 10.2.3 *Given three additive projects* $\mathfrak{p}$, $\mathfrak{q}$, *and* $\mathfrak{r}$, *we have:*

$$[\![\mathrm{Ex}(\mathfrak{p}, \mathfrak{q}), \mathfrak{r}]\!]_m = [\![\mathfrak{p}, \mathrm{Ex}(\mathfrak{q}, \mathfrak{r})]\!]_m$$

This allows us to define the multiplicative connectives as above:

$$\begin{aligned}
\mathbf{A} \multimap \mathbf{B} &= \{\mathfrak{f} \mid \forall \mathfrak{a} \in \mathbf{A}, \mathrm{Ex}(\mathfrak{f}, \mathfrak{a}) \in \mathbf{B}\}, \\
\mathbf{A} :: \mathbf{B} &= \{\mathrm{Ex}(\mathfrak{a}, \mathfrak{b}) \mid \mathfrak{a} \in \mathbf{A}, \mathfrak{b} \in \mathbf{B}\}^{\perp\perp},
\end{aligned}$$

and establish that $\mathbf{A} \multimap \mathbf{B} = (\mathbf{A} \otimes \mathbf{B}^{\perp})^{\perp}$.

It is then possible to define new connectives based on the sum of additive projects:

$$\mathfrak{p} + \mathfrak{q} = (a, \sum_i \alpha_i p_i) + (b, \sum_j \beta_j q_j) = (a + b, \sum_i \alpha_i p_i + \sum_j \beta_j q_j).$$

Note this is just the sum in $\Theta[P]$.

Definition 10.2.4 *Given types* $\mathbf{A}$ *and* $\mathbf{B}$*, one defines:*

$$\mathbf{A} \,\&\, \mathbf{B} = \{\mathfrak{a} + \mathfrak{b} \mid \mathfrak{a} \in \mathbf{A}, \mathfrak{b} \in \mathbf{B}\}^{\perp\perp},$$
$$\mathbf{A} \oplus \mathbf{B} = (\mathbf{A}^{\perp} \,\&\, \mathbf{B}^{\perp})^{\perp}.$$

We can then establish the following lemma, which is the *low-level* formulation of the universal property of the cartesian product.

Lemma 10.2.4

$$\mathrm{Ex}(\mathfrak{f} + \mathfrak{g}, \mathfrak{a}) = \mathrm{Ex}(\mathfrak{f}, \mathfrak{a}) + \mathrm{Ex}(\mathfrak{g}, \mathfrak{a})$$

Proof. Here again, the proof is a simple calculation.

$$\begin{aligned}
&\mathrm{Ex}(\mathfrak{f} + \mathfrak{g}, \mathfrak{a}) \\
&= \mathrm{Ex}((a + b, \sum_i \alpha_i p_i + \sum_j \beta_j q_j), (c, \sum_k \gamma_k r_k)) \\
&= \left((a+b)\mathbf{1}_\gamma + (\mathbf{1}_\alpha + \mathbf{1}_\beta)c + [\![\sum_i \alpha_i p_i + \sum_j \beta_j q_j, \sum_k \gamma_k r_k]\!]_m, \right. \\
&\qquad \left. \sum_{i,k} \alpha_i \gamma_k \mathrm{Ex}(p_i, r_k) + \sum_{j,k} \beta_j \gamma_k \mathrm{Ex}(q_j, r_k) \right) \\
&= \left(a\mathbf{1}_\gamma + b\mathbf{1}_\gamma + \mathbf{1}_\alpha c + \mathbf{1}_\beta c + [\![\sum_i \alpha_i p_i, \sum_k \gamma_k r_k]\!]_m + [\![\sum_j \beta_j q_j, \sum_k \gamma_k r_k]\!]_m, \right. \\
&\qquad \left. \sum_{i,k} \alpha_i \gamma_k \mathrm{Ex}(p_i, r_k) + \sum_{j,k} \beta_j \gamma_k \mathrm{Ex}(q_j, r_k) \right) \\
&= \left(a\mathbf{1}_\gamma + \mathbf{1}_\alpha c + [\![\sum_i \alpha_i p_i, \sum_k \gamma_k r_k]\!]_m, \sum_{i,k} \alpha_i \gamma_k \mathrm{Ex}(p_i, r_k) \right) \\
&\qquad + \left(b\mathbf{1}_\gamma + \mathbf{1}_\beta c + [\![\sum_j \beta_j q_j, \sum_k \gamma_k r_k]\!]_m, \sum_{j,k} \beta_j \gamma_k \mathrm{Ex}(q_j, r_k) \right) \\
&= \mathrm{Ex}(\mathfrak{f}, \mathfrak{a}) + \mathrm{Ex}(\mathfrak{g}, \mathfrak{a})
\end{aligned}$$

❀

This lemma can now be used to establish the following theorem which ensures that additive connectives are properly interpreted, i.e. that the connective & satisfies the universal property of the cartesian product.

Theorem 10.2.5

$$(\mathbf{A} \multimap \mathbf{B}) \,\&\, (\mathbf{A} \multimap \mathbf{C}) \subseteq \mathbf{A} \multimap (\mathbf{B} \,\&\, \mathbf{C})$$

Proof. Let $\mathfrak{f} \in \mathbf{A} \multimap \mathbf{B}$ and $\mathfrak{g} \in \mathbf{A} \multimap \mathbf{C}$. For all $\mathfrak{a} \in \mathbf{A}$, by the previous lemma, $\mathrm{Ex}(\mathfrak{f}+\mathfrak{g}, \mathfrak{a}) = \mathrm{Ex}(\mathfrak{f}, \mathfrak{a}) + \mathrm{Ex}(\mathfrak{g}, \mathfrak{a})$. By definition of $\mathbf{A} \multimap \mathbf{B}$, $\mathrm{Ex}(\mathfrak{f}, \mathfrak{a}) = \mathfrak{b} \in \mathbf{B}$. Similarly, $\mathrm{Ex}(\mathfrak{g}, \mathfrak{a}) = \mathfrak{c} \in \mathbf{C}$. Hence $\mathrm{Ex}(\mathfrak{f} + \mathfrak{g}, \mathfrak{a}) \in \mathbf{A} + \mathbf{B}$ for all $\mathfrak{a} \in \mathbf{A}$, i.e. $\mathfrak{f} + \mathfrak{g} \in \mathbf{A} \multimap (\mathbf{B} \,\&\, \mathbf{C})$, showing the inclusion. ❀

Quantifiers

Lastly, on can easily define second-order connectives. This is one of the main outcomes of this definition of types: second-order quantification, and polymorphism, are obtained for free. Note however that interpreting rules for second-order quantification requires some additional work [18].

[18]: Seiller (2017), *Interaction Graphs: Graphings*

Definition 10.2.5 *Suppose given a family* $\mathbf{A}(\mathbf{X})$ *of types, indexed by the type* $\mathbf{X}$*. We define:*

$$\forall \mathbf{X}\ \mathbf{A}(\mathbf{X}) = \bigcap_{\mathbf{X}\ type} \mathbf{A}(\mathbf{X})$$

$$\exists \mathbf{X}\ \mathbf{A}(\mathbf{X}) = \left(\bigcup_{\mathbf{X}\ type} \mathbf{A}^{\perp}(\mathbf{X}) \right)^{\perp}$$

Exponentials

Lastly, exponential connectives can be defined in some constructions. The underlying principle is that exponentials ensure the possibility of duplicating formulas. As such, an exponential connective should construct from a type $\mathbf{A}$ a new type $!\mathbf{A}$ such that the following *contraction principle* is satisfied: $!\mathbf{A} \multimap (!\mathbf{A} \otimes !\mathbf{A})$. Obviously, the meaning of *the following principle is satisfied* should be detailed: there should not only be a program realising this type, but this program should be *proof-like* (in the terminology of Krivine) or *successful* (in the terminology of game semantics).

In the general framework of graphings, this can be ensured by mapping abstract machines to *stateless machines*[3] This is based on the following observation: if A has only one state, then after A is run, it ends up in its initial state. This ensures[4] that there exists a natural implementation of the contraction principle. Indeed, one can consider the machine that takes a function $\mathfrak{a} \in \mathbf{A}$ as input and uses it twice by doing the following: it starts in a state s_0 and applies a while moving to a second state s_1, and then applies a again from state s_1. This works correctly if and only if the second application of a is 'correct' in the sense that its execution is performed *as if* it was applied for the first time. If the machine a ends up after the first application in a state which is different from the initial state, then the second application would not follow the right computation as it would start from the end state of the first execution. But if a terminates in its initial state, it can be run again a second time.

3: This is the same notion of stateless as discussed in Part 1. However, we here consider that $\mathbf{initial}(M) = \mathbf{terminal}(M)$, which is simply another convention – although less natural – as long as the machine M possess an 'output space', i.e. a subspace in which the output of the computation is read.

4: This is not a necessary condition: the construction in the paragraph hints at a more general definition in which a machine is mapped to a *cyclic* machine, i.e. a machine with ends its computation in the state it started in.

Note that applying the contraction principle uses states. Moreover, there are many ways to extract a stateless machine $!M$ from a machine M: for instance a naive approach would be to identify all states (which in general produces a non-deterministic machine) or project onto one state, i.e. removing anything that involves another state. While those define a notion of exponential, they will not satisfy the other principles that are expected from a proper interpretation of linear logic exponentials: functorial promotion and weakening, and possibly[5] digging and dereliction.

5: Those principles are not required to model Elementary linear logic [144].

10.3. Localised models

The models defined above do not possess low-level constructions corresponding to non-reversible connectives, i.e. $\otimes$ and $\oplus$. This is corrected by considering *localised linear realisability models*. As we will illustrate in the next sections, all the cocycles presented in the previous chapter give rise to localised models.

Multiplicatives

In the following, we write $A \veebar B$ the *symmetric difference* of A and B.

Definition 10.3.1 (A localised linear realisability situation) *A localised (multiplicative) linear realisability situation is given by a boolean algebra $\mathcal{B}$, and a tuple $(P, \phi, \mathrm{Ex}, [\![\cdot,\cdot]\!]_m)$ where P is a set, ϕ is a map $P \to \mathcal{B}$, the execution satisfies*[6] *$\mathrm{Ex} : P_A \times P_B \to P_{A \veebar B}$ is an abstract notion of execution, and $[\![\cdot,\cdot]\!]_m : P \times P \to \Theta$ is a measurement in a commutative group Θ, such that:*

- *(Associativity of execution) When defined,*

$$\mathrm{Ex}(\mathrm{Ex}(p_1, p_2), p_3) = \mathrm{Ex}(p_1, \mathrm{Ex}(p_2, p_3)).$$

- *(Trefoil – or 2-cocycle – Property) When defined:*

$$[\![\mathrm{Ex}(p_1, p_2), p_3]\!]_m + [\![p_1, p_2]\!]_m = [\![p_1, \mathrm{Ex}(p_2, p_3)]\!]_m + [\![p_2, p_3]\!]_m.$$

6: We write here P_A the fiber above A, i.e. the subset P_A of P such that $p \in P_A$ implies $\phi(p) \in A$.

In this case, the above constructions can be followed by taking *locations* – the element $\phi(p) \in \mathcal{B}$ associated to a program p – into account.

Definition 10.3.2 *A* localised project of support $A \in \mathcal{B}$ *is a pair $\mathfrak{p} = (a, p)$ where $a \in \Theta$ and $p \in P_A$.*

Execution and measurement are then extended to localised project as before:

$$\mathrm{Ex}(a, p), (b, q)) = (a\mathbf{1}_q + \mathbf{1}_p b + [\![p, q]\!]_m, \mathrm{Ex}(p, q))$$
$$[\![(a, p), (b, q)]\!]_m = a\mathbf{1}_q + \mathbf{1}_p b + [\![p, q]\!]_m$$

Definition 10.3.3 *Orthogonality is defined on fibers. I.e. the projets $\mathfrak{p}$ and $\mathfrak{p}'$ are orthogonal when they have the same support and $[\![\mathfrak{p}, \mathfrak{p}']\!]_m \in \aleph$.*

Definition 10.3.4 *A* localised type *of support $A \in \mathcal{B}$ is a subset $\mathbf{A}$ of P_A such that there exists $T \subset P_A$ satisfying $\mathbf{A} = T^{\perp}$. Equivalently, $\mathbf{A} = \mathbf{A}^{\perp\perp}$.*

As above, we define the following two constructors on localised types:

$$\mathbf{A} \multimap \mathbf{B} = \{\mathfrak{p} \mid \forall \mathfrak{a} \in \mathbf{A}, \mathrm{Ex}(\mathfrak{p}, \mathfrak{a}) \in \mathbf{B}\},$$
$$\mathbf{A} :: \mathbf{B} = \{\mathrm{Ex}(\mathfrak{a}, \mathfrak{b}) \mid \mathfrak{a} \in \mathbf{A}, \mathfrak{b} \in \mathbf{B}\}^{\perp\perp}$$

We note that in this case, the definition of $\mathbf{A} \multimap \mathbf{B}$ requires that $\mathbf{A}$ and $\mathbf{B}$ have disjoint locations since $\phi(\mathrm{Ex}(p, a)) \cap \phi(A) = \emptyset$. We will therefore specialise the definition of $\mathbf{A} :: \mathbf{B}$ to this particular case.

Definition 10.3.5 *When $A \cap B = \emptyset$, we define the tensor product of $\mathfrak{p}$ of support A and $\mathfrak{p}'$ of support B as $\mathfrak{p} \otimes \mathfrak{p}' = \mathrm{Ex}(\mathfrak{p}, \mathfrak{p}')$, and for types $\mathbf{A}$ and $\mathbf{B}$ of respective supports A and B, we define:*

$$\mathbf{A} \otimes \mathbf{B} = \{\mathfrak{a} \otimes \mathfrak{b} \mid \mathfrak{a} \in \mathbf{A}, \mathfrak{b} \in \mathbf{B}\}^{\perp\perp}$$

Note that $\mathbf{A} \otimes \mathbf{B}$ is a notation for the type $\mathbf{A} :: \mathbf{B}$ when $\mathbf{A}, \mathbf{B}$ have disjoint supports. The following proposition is the equivalent of Proposition 10.2.1. Its corollary is that localised types interpret multiplicative connectives soundly.

Corollary 10.3.1 *Let $\mathbf{A}, \mathbf{B}$ be localised types of disjoint supports, then*

$$\mathbf{A} \multimap \mathbf{B} = (\mathbf{A} \otimes \mathbf{B}^{\perp})^{\perp}.$$

Additives

Following the previous sections, we extend the notion of linear realisability situation to accommodate for additive connectives. We assume once again that Θ is a semiring.

Definition 10.3.6 (A localised general linear realisability situation) *A localised general linear realisability situation is given by a boolean algebra $\mathcal{B}$, and a tuple $(P, \phi, \mathrm{Ex}, [\![\cdot,\cdot]\!]_m, \mathbf{1})$ where:*

- *P is a set,*
- *ϕ is a map $P \to \mathcal{B}$,*
- *Ex is an abstract notion of execution verifying[7] $\mathrm{Ex} : P_A \times P_B \to P_{A \veebar B}$;*
- *$[\![\cdot,\cdot]\!]_m : P \times P \to \Theta$ is a measurement in a commutative group Θ;*
- *$\mathbf{1}.P \to \Theta$ satisfies $\mathbf{1}_{\mathrm{Ex}(p,p')} = \mathbf{1}_p \mathbf{1}_{p'}$;*

7: We write here P_A the fiber above A, i.e. the subset P_A of P such that $p \in P_A$ implies $\phi(p) \in A$.

such that:

- *(Associativity of execution) When defined,*

$$\mathrm{Ex}(\mathrm{Ex}(p_1, p_2), p_3) = \mathrm{Ex}(p_1, \mathrm{Ex}(p_2, p_3)).$$

- *(Trefoil – or 2-cocycle – Property) When defined:*

$$[\![\mathrm{Ex}(p_1, p_2), p_3]\!]_m + [\![p_1, p_2]\!]_m = [\![p_1, \mathrm{Ex}(p_2, p_3)]\!]_m + [\![p_2, p_3]\!]_m.$$

We can then define (localised) additive projects in the same way as above. Following the non-localised construction, one can define a localised general linear realisability situation from a multiplicative one by simply considering formal combinations of programs on a given location. I.e. from $(P, \phi, \mathrm{Ex}, [\![\cdot,\cdot]\!]_m, \mathbf{1})$ we define $(P', \phi', \mathrm{Ex}^{\Theta}, [\![\cdot,\cdot]\!]_m^{\Theta}, \mathbf{1})$ where:

- P' is defined as $\cup_{A \in \mathcal{B}} \Theta[P_A]$ is the set of formal linear combinations of programs having the same location;
- ϕ' is defined as $\phi'(p) = A$ for $p \in \Theta[P_A]$;
- Ex^{Θ}, $[\![\cdot,\cdot]\!]_m^{\Theta}$, and $\mathbf{1}$ are defined as in the non-localised case.

We will now moreover suppose that:

- for all $A \in \mathcal{A}$ there exists a program noted 0_A such that

$$\forall p \in P_A, [\![0_A, p]\!]_m = 0;$$

- for all $p, p' \in P_A$ and $q, q' \in P_B$ with $A \cap B = \emptyset$,

$$[\![\mathrm{Ex}(p,q), \mathrm{Ex}(p',q')]\!]_m = [\![p,p']\!]_m + [\![q,q']\!]_m.$$

We note that these properties are satisfied by the examples of linear realisability situation presented in the previous chapter.

In that case, one can prove that there exists an interpretation of the additive disjunction, i.e. the connective $\oplus$ dual to &, when restricting to a specific class of types called *behaviours*. In the following, we define $\mathfrak{o}_A = (0, 0_A)$ and $\mathbf{0}_A$ as the type $\{\mathfrak{o}_A\}^{\perp\perp}$.

Definition 10.3.7 *A* behaviour *is a localised type* $\mathbf{A}$ *of support* A *satisfying:*

- *for all* $\mathfrak{a} \in \mathbf{A}$ *and all* $\lambda \in \Theta$, $\mathfrak{a} + \lambda\mathfrak{o}_A \in \mathbf{A}$*;*
- *for all* $\mathfrak{a} \in \mathbf{A}^{\perp}$ *and all* $\lambda \in \Theta$, $\mathfrak{a} + \lambda\mathfrak{o}_A \in \mathbf{A}^{\perp}$.

We can then define on behaviours the following connectives, for localised types $\mathbf{A}, \mathbf{B}$ of respective disjoint supports A and B:

$$\begin{aligned}
\mathbf{A} \multimap \mathbf{B} &= \{\mathfrak{p} \mid \forall \mathfrak{a} \in \mathbf{A}, \mathrm{Ex}(\mathfrak{p}, \mathfrak{a}) \in \mathbf{B}\},\\
\mathbf{A} :: \mathbf{B} &= \{\mathrm{Ex}(\mathfrak{a}, \mathfrak{b}) \mid \mathfrak{a} \in \mathbf{A}, \mathfrak{b} \in \mathbf{B}\}^{\perp\perp}\\
\mathbf{A} \,\&\, \mathbf{B} &= \{\mathfrak{a} \otimes \mathfrak{o}_B + \mathfrak{o}_A \otimes \mathfrak{b} \mid \mathfrak{a} \in \mathbf{A}, \mathfrak{b} \in \mathbf{B}\}^{\perp\perp},\\
\mathbf{A} \oplus \mathbf{B} &= ((\mathbf{A} \otimes \mathbf{0}_B) \cup (\mathbf{0}_A \otimes \mathbf{B}))^{\perp\perp}
\end{aligned}$$

We can check that the properties proven above still hold, and that for all $\mathfrak{a}' \in \mathbf{A}$, all $\mathfrak{a} \in \mathbf{A}$ and all $\mathfrak{b} \in \mathbf{B}$ of support B disjoint from that of $\mathbf{A}$:

$$\begin{aligned}
&[\![\mathfrak{a}' \otimes \mathfrak{o}_B, \mathfrak{a} \otimes \mathfrak{o}_B + \mathfrak{o}_A \otimes \mathfrak{b}]\!]_m\\
&\qquad = [\![\mathfrak{a}' \otimes \mathfrak{o}_B, \mathfrak{a} \otimes \mathfrak{o}_B]\!]_m + [\![\mathfrak{a}' \otimes \mathfrak{o}_B, \mathfrak{o}_A \otimes \mathfrak{b}]\!]_m\\
&\qquad = [\![\mathfrak{a}', \mathfrak{a}]\!]_m + [\![\mathfrak{o}_B, \mathfrak{o}_B]\!]_m + [\![\mathfrak{a}', \mathfrak{o}_A]\!]_m + [\![\mathfrak{o}_B, \mathfrak{b}]\!]_m\\
&\qquad = [\![\mathfrak{a}', \mathfrak{a}]\!]_m.
\end{aligned}$$

As a consequence, we have the following theorem.

Theorem 10.3.2 *Given behaviours* $\mathbf{A}$ *and* $\mathbf{B}$ *of disjoint supports,*

$$\mathbf{A} \,\&\, \mathbf{B} = (\mathbf{A}^{\perp} \oplus \mathbf{B}^{\perp})^{\perp}.$$

Remark 10.3.1 The above results hold for general types and not only behaviours. Behaviours are considered more generally in the context of larger models including exponentials. They are at least important in that they are are fully *linear*: one can show they do not satisfy the weakening principle (something that can be true for certain types which are not behaviours).

Localised quantifiers

The construction of second-order connectives can also be adapted. A general intersection would define an empty set because of the constraints on locations, and we therefore consider *localised quantifiers*.

Definition 10.3.8 *Suppose given a family* $\mathbf{A}(\mathbf{X})$ *of types indexed by a type* $\mathbf{X}$ *of support* X*. We define:*

$$\forall \mathbf{X}\ \mathbf{A}(\mathbf{X}) = \bigcap_{\mathbf{X}\ \textit{type of support}\ X} \mathbf{A}(\mathbf{X}),$$

$$\exists \mathbf{X}\ \mathbf{A}(\mathbf{X}) = \left(\bigcup_{\mathbf{X}\ \textit{type of support}\ X} \mathbf{A}^{\perp}(\mathbf{X}) \right)^{\perp}.$$

10.4. Examples of linear realisability models

Graphs models

Graphs allow to define models of MALL. While second order quantifiers can be formally defined at the level of types, finite graphs would not allow for an adequate interpretation of second-order sequent calculus rules. Indeed this would require some kind of rescaling to substitute any formula with a type on a given location. Similarly, while some notion of exponential may be considered, these require erasing some information in the setting of finite graphs since graphs with an arbitrary large number of states (hence an arbitrary large number of vertices) needs to be mapped to graphs on a fixed, finite, number of vertices; the consequence is that one could not expect some important rules – such as the (functorial) promotion rule – to be interpreted correctly.

Allowing for infinite graphs solves these problems; one can then simply translate the first geometry of interaction constructions.

Directed multigraphs. This model was the first I introduced, as part of my PhD thesis. We fix a set of weights Ω. The boolean algebra is the set of subsets of the natural numbers (for instance) $2^{\mathbf{N}}$, and the set of programs P_A for $A \in 2^{\mathbf{N}}$ is the set of Ω-weighted directed multi-graph (with countable set of edges) whose set of vertices is A. The measurement is parametrised by a map $m : \Omega \to \mathbf{R}_{\geqslant 0}$ and defined as

$$[\![F, G]\!]_m = \sum_{\pi \in \mathrm{AltCycle}(F,G)} m(\pi).$$

Together, we obtain the following theorem, based on the execution and measurement defined on graphs in section 9.2 and section 9.3.

Theorem 10.4.1 *For any monoid of weights* Ω*, any map* $m : \Omega \to \mathbf{R}_{\geqslant 0}$*, and any antipode* $\aleph \subset \mathbf{R}_{\geqslant 0}$*, linear realisability constructions on directed multigraphs define a model of Multiplicative Additive Linear Logic (MALL).*

For specific values of Ω and m, the different GoI constructions of Girard can be recovered as special cases of these models:

- if one takes $m(x) = 1$ with the antipode $\{1\}$, orthogonality coincides with the existence of a unique cycle between two graphs; this generalises [17] the first geometry of interaction construction (sometimes called *GoI0*): *multiplicatives* [140];

[17]: Seiller (2016), *Interaction graphs: Additives*
[140]: Girard (1987), *Multiplicatives*

- if one takes $m(x) = \infty$ with the antipode $\{0\}$ or $\mathbf{R}_{\geqslant 0}$, then orthogonality coincides with nilpotency; the resulting model is a combinatorial version of early models of geometry of interaction [13, 138, 139];
- if one takes $m(x) = -\log(1-x)$ with the antipode $\mathbf{R}_{\geqslant 0} \setminus \{0\}$, then orthogonality of F and G corresponds to the fact that $\det(1 - M_F M_G) \neq 0, 1$ [16] (based on results presented in the previous chapter); the resulting model is a combinatorial version of the geometry of interaction in the hyperfinite factor [7].

[13]: Girard (1995), *Geometry Of Interaction III: Accommodating The Additives*
[138]: Girard (1989), *Geometry of Interaction I: Interpretation of System F*
[139]: Girard (1988), *Geometry of Interaction II: Deadlock-free Algorithms*
[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*
[7]: Girard (2011), *Geometry of Interaction V: Logic in the Hyperfinite Factor.*

Coherent graphs. This model is a variant of the directed multigraph model above. It was initially introduced in my PhD thesis, as a way to work with only finite graph. Indeed, in this construction the execution $\mathrm{Ex}(F, G)$ of two graphs is an infinite graph as soon as a cycle appears between F and G. To avoid this while still allowing cycles (which are essential for orthogonality, equivalently to account for correction), I introduced *elementary execution*, i.e. computing execution as the set of *elementary paths*, that is paths that do not use the same edge twice. The naive approach, i.e. considering directed graphs with elementary execution and measuring elementary cycles for defining orthogonality, does not work however. For instance, Figure 10.1 shows a counterexample to the adjunction (a special case of the trefoil cocycle in which on one side the graphs do not share vertices): in Figure 10.1a one can count only two elementary cycles: $deag$ and $cebf$, but in Figure 10.1b there are now three elementary cycles: $deag$, $cebf$, and $debfceag$. This mismatch comes from the fact that the paths deb and cea both use the edge e and therefore should not be both used in an elementary cycle. One therefore has to bookkeep some information to avoid using twice an edge once execution is performed. The technical solution is to define a symmetric relation on the set of edges keeping track of *coherent* edges, i.e. edges that can be used in the same path. More precisely, if F, G are graphs endowed with symmetric binary relations $\frown_F$ and $\frown_G$ on E^F and E^G respectively, then one endows $\mathrm{Ex}(F, G)$ with the symmetric binary relation:

$$e_1 e_2 \dots e_n \frown_{\mathrm{Ex}(F,G)} f_1 f_2 \dots f_m \Leftrightarrow \forall i \in [1, n], \forall j \in [1, m], e_i \frown f_j,$$

where $a \frown b$ on the right-hand side is identified with $\frown_F$ (resp. $\frown_G$) whenever a and b both belong to F (resp. to G), and true in other cases. If one defines this relation to be anti-reflexive, then coherent paths are elementary; moreover anti-reflexivity is preserved by execution. It is then possible to measure only *coherent paths*, i.e. we define for any map $m : \Omega \to \mathbf{R}_{\geqslant 0}$:

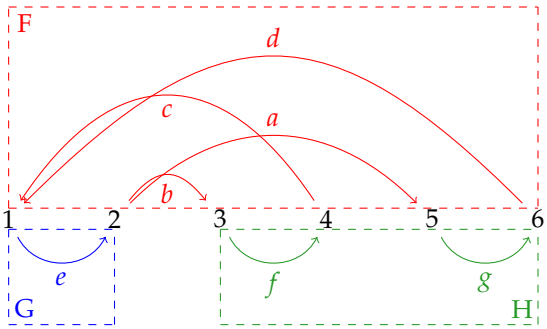
$$[\![F, G]\!]_m^{\frown} = \sum_{\pi \in \mathrm{Cy}^{\frown}(F,G)} m(\pi),$$

and we get the following theorem.

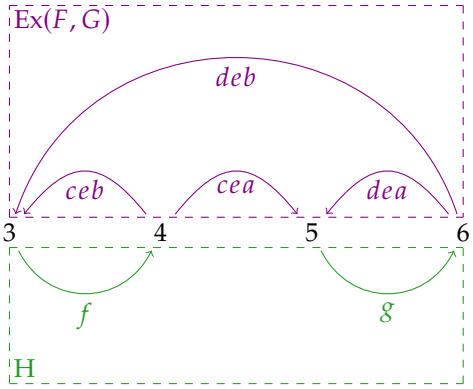
Theorem 10.4.2 *For any monoid of weights Ω, any map $m : \Omega \to \mathbf{R}_{\geqslant 0}$, and any antipode $\aleph \subset \mathbf{R}_{\geqslant 0}$, linear realisability constructions on coherent graphs, endowed with coherent execution and measuring coherent cycles, defines a model of Multiplicative Additive Linear Logic (MALL).*

One of these models was further studied with Lê Thành Dũng Nguyen [31], in particular from the perspective of the correspondance between

[31]: Nguyên et al. (2018), *Coherent Interaction Graphs*



(a) Cycles between F and $\mathrm{Ex}(G, H)$



(b) Cycles between $\mathrm{Ex}(F, G)$ and H

Figure 10.1.: Counter-example of the adjunction for elementary paths and cycles

proof nets and co-graphs established by Rétoré [145].

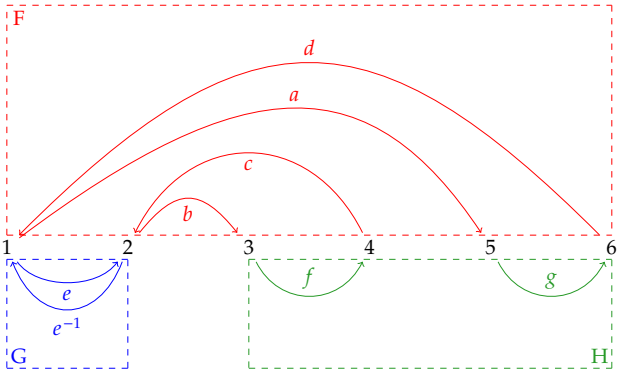
[145]: Retoré (2003), *Handsome proof-nets: perfect matchings and cographs*

Simple paths. This model was introduced together with the coherent graphs version, and considers *simple paths* rather than elementary paths, that is paths that do not go through the same vertex twice. As in the case of coherent graphs, the naive approach fails, and one can easily find a counterexample to the adjunction / trefoil cocycle as illustrated in Figure 10.2: here in Figure 10.2a there are no alternating simple paths, while in Figure 10.2b there exists one. Once again, the mismatch comes from the fact that execution forgets about some information, namely that two edges come from paths going through the same vertex. Following what was done for elementary paths, the solution is the introduction of a coherence relation on edges; the difference being in how the coherence in the execution $\mathrm{Ex}(G, H)$ is defined from the coherences[8] in G and H. Here, if F, G are graphs endowed with symmetric binary relations, then one endows $\mathrm{Ex}(F, G)$ with the symmetric binary relation defined by: two paths are coherent if and only if they are composed of pairwise coherent edges (as in the case of elementary paths) *and* their sets of inner vertices (i.e. vertices visited, except the source and target) are disjoint. From this notion of *simple execution*, and measuring simple coherent cycles, one gets the following theorem.

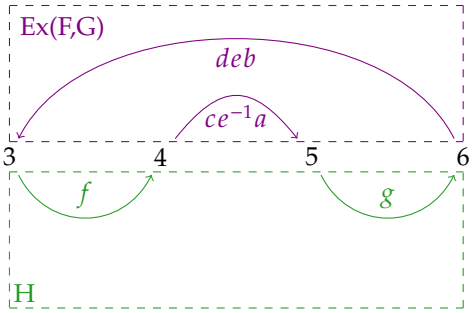
8: This definition is less natural and pleasing than in the coherent graphs case, since the definition implies some reasoning on the vertices visited by a path.

Theorem 10.4.3 *For any monoid of weights Ω, any map $m : \Omega \to \mathbf{R}_{\geqslant 0}$, and any antipode $\aleph \subset \mathbf{R}_{\geqslant 0}$, linear realisability constructions on coherent graphs, endowed with simple execution and the measurement of simple cycles, defines a model of Multiplicative Additive Linear Logic (MALL).*

These models were not studied further.



(a) Cycles between F and $\mathrm{Ex}(G, H)$



(b) Cycles between $\mathrm{Ex}(F, G)$ and H

Figure 10.2.: Counterexample to the adjunction for simple paths and cycles

Remark 10.4.1 The introduction of the coherence relation on edges was never pushed to the more general setting of graphings. It should, however, generalise properly and extend the models mentioned in this section to larger fragments of linear logic (i.e. interpreting properly quantifiers and/or exponentials).

Directed multigraphs: the zeta construction. The last construction based on graphs is to consider the zeta cocycle for graphs presented in section 9.4. This leads to the definition of a new family of models of MALL (and more if one allows for infinite graphs). While I did not study these models in details, one can note that a special case of the zeta measurement corresponds to taking the Euler characteristic; indeed, the Euler characteristic of a graph appears as an analytical property of its Zeta function[9].

9: The Ihara determinant formula establishes that $\zeta_G(z)^{-1} = \det(1 - zA + z^2Q)(1-u^2)^{-\chi}$, where A is the adjacency matrix, $Q + 1$ the valency matric, and χ the Euler characteristic of G.

Theorem 10.4.4 *For any monoid of weights Ω, any map $m : \Omega \to \mathbf{R}_{\geqslant 0}$, and any antipode $\aleph \subset \mathbf{C} \to \mathbf{C}$, linear realisability constructions on directed multigraphs, endowed with execution and the zeta measurement, defines a model of Multiplicative Additive Linear Logic (MALL).*

Graphings models

Graphings models generalise the graph models above, both the initial case of measuring one-cycles and the construction based on zeta cocycles. The consideration of graphings, and in particular graphings induced by an action on the real numbers allowing for *rescaling*, allowed me to prove that one can properly interpret second order quantifiers [18]. It was shown

[18]: Seiller (2017), *Interaction Graphs: Graphings*

in another work [14] that one can interpret Elementary Linear Logic [144], a variant of linear logic which characterises [146] the complexity class Elementary of elementary functions [147]. Lastly, I showed [30] that allowing for *continuous* sets of states, i.e. sets of states isomorphic to the real interval $[0,1]$ can lead to a model of standard exponential connectives. While, as mentioned above, standard exponentials can also be interpreted in infinite discrete graphs, the approach is much different here and, to my knowledge, do not correspond to previously considered interpretations of exponential connectives (in geometry of interaction or game semantics).

[14]: Seiller (2019), *Interaction Graphs: Exponentials*

[144]: Girard (1995), *Light Linear Logic*

[146]: Danos et al. (2003), *Linear logic and elementary time*

[147]: Kalmar (1943), *Egyszerii pelda eldonthetetlen aritmetikai problemara*

[30]: Seiller (2016), *Interaction Graphs: Full Linear Logic*

Submodels

One can then check that the interpretations of proofs by graphings in the above models are all deterministic. As a corollary of Lemma 9.2.2, this implies that the above models can be restricted to deterministic graphs and graphings. In particular, one obtains models of linear logic by realisability constructions on the set of all partial measured dynamical systems whose graph is included in $\mathcal{P}(\alpha)$, based on the identification of graphings with dynamical systems [135], whose orthogonality is defined based on Ruelle zeta functions.

[135]: Seiller (2022), *Zeta functions and the (linear) logic of Markov processes*

Similarly, Lemma 9.2.3 shows that the above constructions yield subprobabilistic submodels. In particular, the probabilistic submodel of graphings corresponds to a linear realisability construction on discrete-image sub-Markov processes.

Markov processes

We build on the execution and the zeta function cocycle on sub-Markov processes defined in section 9.2 and section 9.4.

Here abstract programs are defined as pairs of two measured spaces written as $\mathbf{X} \to \mathbf{Y}$, together with a sub-Markov kernel κ from $\mathbf{X}$ to $\mathbf{Y}$, that is a kernel $(\mathbf{X} \cdot [0,1]) \times (\mathbf{Y} \cdot [0,1]) \to [0,1]$, where $\cdot$ denotes a product that will be treated in specific manner when defining the execution and orthogonality.

We need to define an operation corresponding to extending the kernels to $(\mathbf{X} \cdot [0,1] \times [0,1]) \times (\mathbf{Y} \cdot [0,1] \times [0,1]) \to [0,1]$. This can be done in two different ways: extending by the identity on the first copy of $[0,1]$, or extending by the identity on the second copy of $[0,1]$:

$$\begin{aligned} \kappa^{\dagger} : &\quad ((x,e,f),(\dot{x},\dot{y},\dot{f})) \mapsto \kappa((x,e),(\dot{x},\dot{e}))\mathbf{1}(f,\dot{f}) \\ \kappa^{\ddagger} : &\quad ((x,e,f),(\dot{x},\dot{y},\dot{f})) \mapsto \kappa((x,f),(\dot{x},\dot{f}))\mathbf{1}(e,\dot{e}). \end{aligned}$$

Given two abstract objects $(\mathbf{X} \to \mathbf{Y}, \kappa_1)$ and $(\mathbf{X}' \to \mathbf{Y}', \kappa_2)$, we define the execution as the abstract program

$$\left((\mathbf{X} \cup \mathbf{X}') \setminus (\mathbf{Y} \cup \mathbf{Y}') \to (\mathbf{Y} \cup \mathbf{Y}') \setminus (\mathbf{X} \cup \mathbf{X}'), \mathrm{Ex}(\kappa_1^{\dagger} \bullet \kappa_2^{\ddagger})\right),$$

which is then written as a kernel $(\mathbf{X} \cdot [0,1]) \times (\mathbf{Y} \cdot [0,1]) \to [0,1]$ through a fixed map $[0,1]^2 \to [0,1]$.

The measurement is then defined using the zeta function:

$$[\![\kappa_1, \kappa_2]\!]_m = \zeta_{\kappa_1^\dagger \bullet \kappa_2^\ddagger}(z).$$

This defines a localised general linear realisability situation, which can be shown to model second-order linear logic [135].

[135]: Seiller (2022), *Zeta functions and the (linear) logic of Markov processes*

von Neumann algebras

Hyperfinite GoI (5.0). This is the first iteration of the 'hyperfinite geoemtry of interaction'. The construction does not appear in any published work (although it is somehow sketched in my work about the classification of maximal abelian sub-algebras [11] detailed in chapter 11). Some background material about von Neumann algebras can be found in the appendix.

[11]: Seiller (2018), *A Correspondence between Maximal Abelian Sub-Algebras and Linear Logic Fragments*

Here one works within the hyperfinite factor $\mathfrak{R}_{0,1}$ of type II_∞. An abstract program is defined by a tuple $(p, \mathfrak{A}, \alpha, A)$ where $p \in \mathfrak{R}_{0,1}$ is a *finite projection*, $\mathfrak{A}$ is a finite von Neumann algebra of type I, A is a self-adjoint operator of norm at most 1 in $p\mathfrak{R}_{0,1}p \otimes \mathfrak{A}$ – equivalently an operator $A \in \mathfrak{R}_{0,1} \otimes \mathfrak{A}$ such that $A = pAp$ –, and α is a *pseudo-trace* on $\mathfrak{A}$, that is a a hermitian ($\alpha(u) = \alpha(u^*)$), tracial ($\alpha(uv) = \alpha(vu)$), faithful and normal (i.e. σ-weakly continuous) linear form.

Remark 10.4.2 This seems quite abstract, but the von Neumann algebra $\mathfrak{A}$ can always be written as a direct sum of matrix algebras, i.e.

$$\mathfrak{A} = \bigoplus_{i=1}^{k} \mathfrak{M}_{n_i}(\mathbf{C}),$$

and the pseudo-trace corresponds to a real linear combination of the normalised[10] traces on each $\mathfrak{M}_{n_i}$, i.e. there exists real numbers $\lambda_1, \dots, \lambda_k$ such that:

$$\alpha(\bigoplus_{i=1}^{k} a_i) = \sum_{i=1}^{k} \lambda_i \mathrm{tr}(a_i).$$

10: Recall that the normalised trace is the usual trace of the matrix divided by the dimension of the underlying Hilbert space, so that the trace of the identity equals 1.

This will be used to factor the construction through a multiplicative localised linear realisability situation.

The execution between two abstract programs $(p, \mathfrak{A}, \alpha, A)$ and $(q, \mathfrak{B}, \beta, B)$ is defined as the solution to the feedback equation (Equation 9.1) involving $A^\dagger$ and $B^\ddagger$ which are the operators on $\mathfrak{R}_{0,1} \otimes \mathfrak{A} \otimes \mathfrak{B}$, i.e. writing τ the operator $\mathfrak{B} \otimes \mathfrak{A} \to \mathfrak{A} \otimes \mathfrak{B}$ defined from $b \otimes a \mapsto a \otimes b$ on simple tensors:

$$A^\dagger = A \otimes 1_{\mathfrak{B}}$$
$$B^\ddagger = (1_{\mathfrak{R}_{0,1}} \otimes \tau)(B \otimes 1_{\mathfrak{A}}).$$

The result of the execution is then the abstract program $(p \cap q, \mathfrak{A} \otimes \mathfrak{B}, \alpha \otimes \beta, \mathrm{Ex}(A^\dagger, B^\ddagger))$.

The measurement is then defined as follows. First, define

$$\mathrm{ldet}_\alpha(1 - A) = \sum_{i=1}^{\infty} \frac{\mathrm{tr}_{\mathfrak{R}_{0,1}} \otimes \alpha(A^k)}{k},$$

which produces from an abstract program a real number. Then the measurement between abstract programs is defined as:

$$[\![(p, \mathfrak{A}, \alpha, A), (q, \mathfrak{B}, \beta, B)]\!]_m = \mathrm{ldet}_{\alpha\otimes\beta}(1 - A^\dagger B^\ddagger).$$

Note the expression is named ldet since it can be related to the logarithm of the (Fuglede-Kadison) determinant.

Now, following Remark 10.4.2, we can define the following localised multiplicative linear realisability situation and understand abstract programs as real linear combinations of elements of this underlying multiplicative setting. The notion of abstract program is here a tuple (p, a, A) where $p \in \mathfrak{R}_{0,1}$ is a finite projection, $a \in \mathbf{Z}^*$ is a non-zero natural number, and A is a self-adjoint operator in $p\mathfrak{R}_{0,1}p \otimes \mathfrak{M}_a(\mathbf{C})$. Here again the execution is defined as the solution of the feedback equation involving $A^\dagger$ and $B^\ddagger$, and the measurement is defined as above using the normalised trace on $\mathfrak{M}_a(\mathbf{C})$:

$$[\![(p, a, A), (q, b, B)]\!]_m = \mathrm{ldet}_{\mathrm{tr}\otimes\mathrm{tr}}(1 - A^\dagger B^\ddagger).$$

Girard's construction then picked the specific antipode $\aleph = \mathbf{R}^*_{\geqslant 0}$ of non-zero positive reals.

This factoring of the general setting was essential in my early work showing how interaction graphs can provide a combinatorial version of the hyperfinite geometry of interaction. This construction provides a model of multiplicative and additive connectives following the abstract constructions above, as well as constrained exponential connectives from Elementary Linear Logic [144, 146].

[144]: Girard (1995), *Light Linear Logic*
[146]: Danos et al. (2003), *Linear logic and elementary time*

Hyperfinite GoI (5.1). The hyperfinite geometry of interaction in its official (published) form [7] makes use of the Fuglede-Kadison determinant [10] instead of the expression ldet considered previously. I.e. the abstract programs are defined in the same way, the execution is defined similarly, but the measurement is defined using $-\log(\det_{\mathrm{FK}}(1 - A))$ instead of the infinite series $\mathrm{ldet}(1 - A)$ used above.

[7]: Girard (2011), *Geometry of Interaction V: Logic in the Hyperfinite Factor.*
[10]: Fuglede et al. (1952), *Determinant theory in finite factors*

This provides a model of Elementary Linear logic (including additives), as for the GoI0 construction above. While this difference does not impact much the constructions, they lead to an important change in the definition of *successful* projects, which are understood as the interpretations of proofs. This is explained in chapter 11.

Ludics, transcendental syntax, nets

Some alternative models of linear logic are constructed based on biorthogonality methods, namely *Ludics* [148, 149] and the more recent *transcendental syntax* [34, 150]. Those are more directly based on a syntactic notion of abstract programs.

[148]: Girard (2001), *Locus solum: From the rules of logic to the logic of rules*
[149]: Terui (2011), *Computational ludics*
[34]: Eng et al. (2022), *Multiplicative linear logic from a resolution-based tile system*
[150]: Girard (2017), *Transcendental syntax I: deterministic case*

In both cases, the model can be understood as a linear realisability situation of a particular kind. In these models, one starts from a set of abstract programs P and a notion of execution Ex which is associative, and *defines orthogonality* directly by saying that two programs p, q are *orthogonal* whenever $\mathrm{Ex}(p, a)$ leads to a normal form (hence the computation – or cut-elimination procedure – terminates). Indeed, the execution $\mathrm{Ex}(p, q)$

may be undefined in these models. The setting then corresponds to defining the following measurement $[\![p, q]\!]_m = 0$ when $\mathrm{Ex}(p, q)$ is defined, and $[\![p, q]\!]_m = \infty$ otherwise. The trefoil property is then

$$[\![\mathrm{Ex}(p, q), r]\!]_m + [\![p, q]\!]_m = [\![p, \mathrm{Ex}(q, r)]\!]_m + [\![q, r]\!]_m,$$

and states that if $\mathrm{Ex}p, q$ and $\mathrm{Ex}(\mathrm{Ex}(p, q), r)$ are well-defined if and only if $\mathrm{Ex}(q, r)$ and $\mathrm{Ex}(p, \mathrm{Ex}(q, r))$ are well-defined: this is in fact the associativity of execution.

In this specific setting, one does not requires to define *projects*; they would be of the form $(0, p)$ or (∞, p) but defining execution partially avoids the need for (∞, p) elements.

I took a similar approach with Adrien Ragot and Lorenzo Tortora de Falco recently [32, 33] to define a linear realisability model directly on an abstract notion of *nets*, a generalised and untyped version of linear logic's *proof nets*.

[32]: Ragot et al. (2023), *Linear Realisability Over Nets and Second Order Quantification (short paper)*
[33]: Ragot et al. (2024), *Linear realisability on untyped nets*

Semantic complexity 11.

11.1. von Neumann algebras and expressivity

One of my own favorite results is published under the name "A Correspondence between Maximal Abelian Sub-Algebras and Linear Logic Fragments". While it has been published much later, it is in fact the first result of my PhD. At the time, I observed that the result exposed below held in the setting of the geometry of interaction in the hyperfinite factor. This work, done during the summer, hit a rock when I realised that Girard had changed the constructions of the latter during that same period of time. It then took me some time to adapt the results and write them properly: the results are more natural in the setting of GoI 5.0 (described above and in one of my papers [11], but never published by Girard) but can be expressed in GoI5.1 with some additional work.

[11]: Seiller (2018), *A Correspondence between Maximal Abelian Sub-Algebras and Linear Logic Fragments*

This result exhibits a remarkable correspondence between a classification of maximal abelian sub-algebras and fragments of linear logic. The maximal abelian sub-algebras arise naturally when one wishes to define the notion of *successful* projects, i.e. projects that are the interpretations of proofs. Since these definitions and the main theorem require a number of definitions and results about maximal sub-algebras, we will present below some results to provide a minimal background together with a number of intuitions that should help the reader to grasp some subtleties of the theory. After defining what exactly is a maximal abelian sub-algebra, we will start by explaining the classification of such in type I factors, the simpler case. We will then go on with the case of type II algebras which is more involved. Note that an appendix also covers some background on von Neumann algebras.

Truth and successful projects

The following definition should be clear from the discussion in chapter 10 on operator algebra models.

Definition 11.1.1 *A* hyperfinite project *is a tuple* $\mathfrak{a} = (p, a, \mathfrak{A}, \alpha, A)$*, where:*

- p *is a finite projection in* $\mathfrak{R}_{0,1}$*, the* carrier *of* $\mathfrak{a}$ *;*
- $a \in \mathbf{R} \cup \{\infty\}$ *is called the* wager *of* $\mathfrak{a}$*;*
- $\mathfrak{A}$ *is a finite von Neumann algebra of type I, the* dialect *of* $\mathfrak{a}$*;*
- α *is a pseudo-trace on* $\mathfrak{A}$*;*
- $A \in p\mathfrak{R}_{0,1}p \otimes \mathfrak{A}$ *is a hermitian operator of norm at most* 1*.*

Using Girard's notation, we will write $\mathfrak{a} = a \cdot + \cdot \alpha + A$*. When the dialect is equal to* $\mathbf{C}$*, we will denote by* $1_{\mathbf{C}}$ *the 'trace'* $x \mapsto x$*.*

We first recall the notion of truth considered by Girard [7] which is based on the notion of *successful hyperfinite project*. We will then explain this definition and propose a variant that can be used in the GoI5.0 construction. We then exhibit a correspondence between Girard's notion and the latter.

[7]: Girard (2011), *Geometry of Interaction V: Logic in the Hyperfinite Factor.*

Definition 11.1.2 (Viewpoint) *A* viewpoint *is a representation π of the algebra $\mathfrak{R}_{0,1}$ onto $L^2(\mathbf{R},\lambda)$ where λ is the Lebesgue measure, which satisfies the following conditions:*

- $L^\infty(\mathbf{R},\lambda) \subset \pi(\mathfrak{R}_{0,1})$;
- $\forall A \subset \mathbf{R}$, $\mathrm{tr}(\pi^{-1}(\chi_A)) = \lambda(A)$, *where χ_A is the characteristic function of A.*

A viewpoint if faithful *when the representation π is faithful.*

If $T : \mathbf{R} \to \mathbf{R}$ is a measure-preserving transformation, one can define the isometry $[T] \in \mathcal{L}(L^2(\mathbf{R},\lambda))$:

$$[T] : f \in L^2(\mathbf{R},\lambda) \mapsto f \circ T \in L^2(\mathbf{R},\lambda)$$

That $[T]$ is an isometry comes from the fact that T is measure-preserving:

$$\begin{aligned}
\langle [T]f, [T]g\rangle &= \int_{\mathbf{R}} ([T]f)(x)\overline{([T]g)(x)}d\lambda(x)\\
&= \int_{\mathbf{R}} f\circ T(x)\overline{g\circ T(x)}d\lambda(x)\\
&= \int_{\mathbf{R}} f\circ T(x)\overline{g\circ T(x)}d\lambda(x)\\
&= \int_{T(\mathbf{R})} f(x)\overline{g(x)}d\lambda(x)\\
&= \int_{\mathbf{R}} f(x)\overline{g(x)}d\lambda(x)\\
&= \langle f, g\rangle
\end{aligned}$$

Suppose now given $U : X \to Y$ a measure-preserving transformation, with $X, Y \subset \mathbf{R}$ measurable subsets. We define, for all map $f \in L^2(\mathbf{R},\lambda)$, $[U]f(x) = f \circ U(x)$ if $x \in X$ and $[U]f(x) = 0$ otherwise. The operator $[U]$ thus defined is a partial isometry. Indeed, if we write p the projection in $\mathcal{L}(L^2(\mathbf{R},\lambda))$ induced by the characteristic map of Y, then for all $f, g \in pL^2(\mathbf{R},\lambda)$,

$$\begin{aligned}
\langle [U]f, [U]g\rangle &= \int_{\mathbf{R}} ([U]f)(x)\overline{([U]g)(x)}d\lambda(x)\\
&= \int_{X} ([U]f)(x)\overline{([U]g)(x)}d\lambda(x)\\
&= \int_{X} f\circ U(x)\overline{g\circ U(x)}d\lambda(x)\\
&= \int_{Y} f(x)\overline{g(x)}d\lambda(x)\\
&= \int_{\mathbf{R}} f(x)\overline{g(x)}d\lambda(x)
\end{aligned}$$

Moreover, it is clear that for all $f, g \in (1-p)L^2(\mathbf{R},\lambda)$ one has $\langle [U]f, [U]g\rangle = 0$.

Definition 11.1.3 *A hyperfinite project $\mathfrak{a} = 0 \cdot + \cdot \alpha + A$ of carrier p is* successful *w.r.t. a viewpoint π when:*

- $\pi(p) \in L^\infty(\mathbf{R})$;
- *α is the normalised trace on $\mathfrak{A}$;*

- *there exists a basis* $e_1, \dots, e_n$ *of the dialect* $\mathfrak{A}$ *such that* $A = [f]$ *where* f *is a partial measure-preserving bijection of* $\mathbf{R} \times \{1, \dots, n\}$*;*
- *the set* $\{x \in \mathbf{R} \times \{1, \dots, n\} \mid f(x) = x\}$ *is of null measure.*

Remark 11.1.1 We added the last condition to the definition proposed by Girard [7]. This condition corresponds to the trace condition in our definition of promising projects (Definition 11.1.5), which can be explained intuitively (Remark 11.1.2).

Success and Bases

In geometry of Interaction, as in the theory of proof structures [4], in game semantics [28] or in classical realisability [78, 79], one needs to characterise those elements which correspond to proofs: proof nets (i.e. satisfying the correctness criterion), winning strategies, or proof-like terms. In GoI models, these 'proof-like terms', or *winning strategies* are called *successful projects*. In previous GoI models a successful project was defined as a partial symmetry. This definition was quite satisfying, but some of its important properties relied on the fact that the model depended on a chosen MASA $\mathfrak{A}$, i.e. it relied on the fact that the constructions were basis-dependent (i.e. operators are chosen in the normalising groupoid of $\mathfrak{A}$ only).

In Girard's hyperfinite model, constructions are no longer basis-dependent: the operators considered are no longer restricted to those elements that are in the normalising groupoid of a MASA, but can be any hermitian operator of norm at most 1. By going to this more general setting, defining successful projects as partial symmetries is no longer satisfying. The reason for this is quite easy to understand. Indeed, a satisfying notion of success should verify two essential properties. The first of these is that it should 'compose', i.e. the execution of two successful projects should be a successful project. The second is that it should be 'coherent', i.e. two orthogonal projects cannot be simultaneously successful.

Since we are no longer restricted to operators in a chosen normalising groupoid, the definition of successful projects as partial symmetries now lacks these two essential properties. This can be illustrated by easy examples on matrices (to obtain examples in the hyperfinite factor, use your favourite embedding). For instance, let us consider the following matrices:

$$u = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad v = \begin{pmatrix} 0 & \sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} \\ \sqrt{\frac{1}{2}} & 0 & 0 \\ -\sqrt{\frac{1}{2}} & 0 & 0 \end{pmatrix}$$

One can check that u, v are partial symmetries: it is obvious for u, and the following computation shows it for v.

$$vv^* = v^2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} \\ 0 & -\frac{1}{2} & \frac{1}{2} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} \\ 0 & -\frac{1}{2} & \frac{1}{2} \end{pmatrix}^2$$

However, their product is not a partial isometry (hence not a partial symmetry), which shows that the notion do not compose.

$$uv = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & \sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} \\ \sqrt{\frac{1}{2}} & 0 & 0 \\ -\sqrt{\frac{1}{2}} & 0 & 0 \end{pmatrix} = \begin{pmatrix} \sqrt{\frac{1}{2}} & 0 & 0 \\ 0 & \sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} \\ 0 & 0 & 0 \end{pmatrix}$$

Moreover, the computation of the determinant of $1 - uv$ shows that the notion is not coherent, since one can define from them two orthogonal projects.

$$\begin{vmatrix} 1 - \sqrt{\frac{1}{2}} & 0 & 0 \\ 0 & 1 - \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} \\ 0 & 0 & 1 \end{vmatrix} = (1 - \sqrt{\frac{1}{2}})^2 \neq 0, \infty$$

In order to obtain a good notion of successful project, we will have to restrict ourselves to a class of partial symmetries which is closed under sum and composition. Sums and products of partial isometries in the normalising groupoid of a MASA $\mathfrak{A}$ are again partial isometries* in the normalising groupoid of $\mathfrak{A}$. This will be enough to show that if u and v are partial symmetries in $\mathrm{G}(\mathfrak{A})$, then $u :: v$ is a partial symmetry in $\mathrm{G}(\mathfrak{A})$.

In the finite-dimensional case, this amounts to choosing a basis. Indeed, the complete classification of MASAs in $\mathcal{L}(\mathbb{H})$ (Theorem 11.1.6) shows that when $\mathbb{H}$ is of finite dimension the MASAs of $\mathcal{L}(\mathbb{H})$ are exactly the diagonal MASAs: the set of diagonal matrices in a fixed basis. One can therefore define a *subjective* notion of successful projects, i.e. a notion of success that depends on the choice of a basis. An operator is then *successful w.r.t.* $\mathcal{B}$ when it is a partial symmetry in the normalising groupoid of the algebra $\mathfrak{D}_{\mathcal{B}}$ of diagonal operators in the basis $\mathcal{B}$. The composition of such partial symmetries can be shown to be itself a partial symmetry in the normalising groupoid of $\mathfrak{D}_{\mathcal{B}}$ and the definition of success is therefore consistent with the execution. However, we are still unable to show the coherence of this definition: given two partial symmetries u, v in $\mathrm{G}(\mathfrak{D}_{\mathcal{B}})$, the logarithm of the determinant of $1 - uv$ is not necessarily equal to 0 or ∞. Once again, it is enough to consider matrices to illustrate this fact, and we will give an example with 2×2 matrices. Let u and v be the following matrices:

$$u = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \qquad v = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Then $\det(1 - uv) = 4$, i.e. $-\log(\det(1 - uv)) \neq 0, \infty$.

The issue here arises from the fact that one cannot distinguish between the identity and its opposite, i.e. the definition does not exclude negative coefficients. The solution proposed by Girard [7] is to consider a notion of success that depends on a representation of the algebra: a successful project will then have its operators u induced from a measure-preserving transformation on a measured space.

* In the case of the sum, one has to impose a condition on domains and codomains.

There is however another way to bypass the problem just exposed. It corresponds to an old version of Girard's hyperfinite GoI model, which we will refer to as the *matricial* GoI model, in which orthogonality is slightly modified. In this GoI model, defined as GoI5.0 in the previous chapter, it is possible to keep the notion of successful projects as partial symmetries in the normalising groupoid of a MASA $\mathfrak{A}$ since the change of orthogonality bypasses the issue with coherence. This GoI model will be related to the hyperfinite GoI model later on.

The matricial GoI model is based on the same notion of projects as the hyperfinite GoI model. The two constructions essentially differ on the measurement $[\![\cdot,\cdot]\!]_m$ which is used to defines the orthogonality relation. Notice that all constructions on projects are the same in both models.

We are now ready to define a notion of success for the matricial GoI model. To avoid an overlap of terminology, we call here *outlooks* the equivalent of Girard's *viewpoints*, and *promising projects* the equivalent of Girard's *successful projects*. We will first show how this notion of success satisfies coherence and compositionality, and we will then explain its relation to Girard's notion of success.

Definition 11.1.4 (Outlook) *An* outlook *is a MASA in* $\mathfrak{R}_{0,1}$.

Definition 11.1.5 (Promising Project) *An hyperfinite project* $\mathfrak{a} = a\cdot + \cdot\alpha + A$ *is* promising w.r.t the outlook $\mathfrak{P}$ *if:*

- ***Dialect.*** *The dialect* $\mathfrak{A}$ *is a finite factor, i.e. a matrix algebra;*
- ***Pseudo-Trace.*** α *is the normalised trace on* $\mathfrak{A}$*;*
- ***Wager.*** $\mathfrak{a}$ *is wager-free:* $a = 0$*;*
- ***Symmetry.*** *A is a partial symmetry in the normalising groupoid* $\mathfrak{P}\otimes\mathfrak{Q}$*, where* $\mathfrak{Q}$ *is a MASA in* $\mathfrak{A}$*;*
- ***Traces.*** *for all projection* $\pi \in \mathfrak{P}\otimes\mathfrak{R}_{\mathfrak{A}}(\mathfrak{Q})$, $\mathrm{tr}(\pi A) = 0$.

Remark 11.1.2 We remark here that the last 'trace condition' is an addition to Girard's condition of successful projects (see Remark 11.1.1). This is however a quite natural condition as a successful or promising project should be understood as the representation of a set of axiom links in a proof net, i.e. it should be a symmetry without any fixpoints. Even though the alternative definition of success without this additional condition would be satisfactory (it would satisfy coherence and compositionality), it does not convey the intuitions coming from proof nets.

Definition 11.1.6 *A conduct* $\mathbf{A}$ *is* correct w.r.t. the outlook $\mathfrak{A}$ *if there exists a hyperfinite project* $\mathfrak{a} \in \mathbf{A}$ *which is promising w.r.t.* $\mathfrak{A}$.

We can check that the notion of promising project satisfies the essential properties: compositionality and coherence.

Proposition 11.1.1 (Coherence) *Let* $\mathfrak{P}$ *be an outlook. The two conducts* $\mathbf{A}$ *and* $\mathbf{A}^{\perp}$ *cannot both contain a promising project w.r.t.* $\mathfrak{P}$.

Proposition 11.1.2 (Compositionality) *Let* $\mathbf{A}, \mathbf{B}, \mathbf{C}$ *be conducts such that* $\mathbf{A}$ *and* $\mathbf{C}^{\perp}$ *are non-empty. If* $\mathfrak{f} \in \mathbf{A} \multimap \mathbf{B}$ *and* $\mathfrak{g} \in \mathbf{B} \multimap \mathbf{C}$ *are promising hyperfinite projects w.r.t. the outlook* $\mathfrak{P}$*, then* $\mathfrak{f} :: \mathfrak{g}$ *is a promising hyperfinite project w.r.t.* $\mathfrak{P}$ *in the conduct* $\mathbf{A} \multimap \mathbf{C}$.

This notion can then be related to Girard's original notion of success as follows.

Proposition 11.1.3 *Every faithful viewpoint defines an outlook.*

Proposition 11.1.4 *Let π be a faithful viewpoint, and $\mathfrak{B} = \pi^{-1}(L^\infty(\mathbf{R}, \lambda))$ the outlook defined by π. If $\mathfrak{a} = 0 \cdot + \cdot \mathrm{tr} + A$ is successful w.r.t. π, then it is promising w.r.t. $\mathfrak{B}$.*

MASAs in type I factors

Before detailing the results, we need to recall some important results on the classification of maximal abelian subalgebras.

Definition 11.1.7 *Let $\mathfrak{M}$ be a von Neumann algebra. A* maximal abelian sub-algebra *(MASA) $\mathfrak{A}$ of $\mathfrak{M}$ is a von Neumann sub-algebra of $\mathfrak{M}$ such that for all intermediate sub-algebras $\mathfrak{B}$, i.e. $\mathfrak{A} \subset \mathfrak{B} \subset \mathfrak{M}$, if $\mathfrak{B}$ is abelian then $\mathfrak{A} = \mathfrak{B}$.*

If $\mathfrak{A}$ and $\mathfrak{B}$ are MASAs in a von Neumann algebra $\mathfrak{M}$, they can be 'isomorphic' in three different ways:

- they can be isomorphic as von Neumann algebras – this is the weakest notion;
- there can exists an automorphism Φ of $\mathfrak{M}$ such that $\Phi(\mathfrak{A}) = \mathfrak{B}$; we then say that $\mathfrak{A}$ and $\mathfrak{B}$ are conjugated;
- there can exist a unitary operator[1] $u \in \mathfrak{M}$ such that $u\mathfrak{A}u^* = \mathfrak{B}$ – this is the strongest notion; we then say that $\mathfrak{A}$ and $\mathfrak{B}$ are unitarily equivalent.

1: We recall that a unitary operator is an operator u such that $uu^* = u^*u = 1$.

Let us quickly discuss the finite-dimensional case. We fix $\mathbb{H}$ a finite-dimensional Hilbert space of dimension $k \in \mathbf{N}$. Then $\mathcal{L}(\mathbb{H})$ is isomorphic to the algebra of $k \times k$ matrices. Picking a basis $\mathcal{B} = (b_1, \ldots, b_k)$ of $\mathbb{H}$, one can define the sub-algebra $\mathfrak{D}_\mathcal{B}$ of $\mathcal{L}(\mathbb{H})$ containing all diagonal matrices in the basis $\mathcal{B}$. This algebra is obviously commutative, and it is moreover maximal as a commutative sub-algebra of $\mathcal{L}(\mathbb{H})$: if $\mathfrak{A}$ is a commutative sub-algebra of $\mathcal{L}(\mathbb{H})$ containing $\mathfrak{D}_\mathcal{B}$, then $\mathfrak{A} = \mathfrak{D}_\mathcal{B}$. A more involved argument shows that any maximal abelian sub-algebra of $\mathcal{L}(\mathbb{H})$ is the diagonal algebra induced by a basis; this result is also a direct corollary of Proposition 11.1.5. These algebras $\mathfrak{D}_\mathcal{B}$ where $\mathcal{B}$ is a basis of $\mathbb{H}$ are clearly pairwise isomorphic, as it suffices to map bijectively the bases one onto the other. They are in fact unitarily equivalent, as such a bijection induces a unitary operator. This shows that the distinctions we just made are useless in the finite-dimensional case: all MASAs are unitarily equivalent.

We will now state a classification result about maximal abelian sub-algebras of $\mathcal{L}(\mathbb{H})$, which gives a complete answer to the classification problem of MASAs in type I factors. This theorem will be preceded by a proposition showing that all *diffuse* MASAs in $\mathcal{L}(\mathbb{H})$ are unitarily equivalent; this will be of use later on, as those MASAs of a type II factor $\mathfrak{N} \subset \mathcal{L}(\mathbb{H})$ which are also MASAs of $\mathcal{L}(\mathbb{H})$ are necessarily diffuse. These results can be found in Sinclair and Smith's book [15].

[15]: Sinclair et al. (2008), *Finite von Neumann algebras and Masas*

Proposition 11.1.5 *Let $\mathfrak{A}$ be a MASA of $\mathcal{L}(\mathbb{H})$ which does not have (non-zero) minimal projections – we say in this case that $\mathfrak{A}$ is a diffuse MASA. Then there exists a unitary $U : \mathbb{H} \to L^2([0,1])$ such that $U\mathfrak{A}U^* = L^\infty([0,1])$.*

Theorem 11.1.6 *Let $\mathfrak{A}$ be a MASA in $\mathcal{L}(\mathbb{H})$. Then:*

- *either $\mathfrak{A}$ is unitarily equivalent to $L^\infty([0,1])$ (diffuse case);*
- *or $\mathfrak{A}$ is unitarily equivalent to $\mathfrak{D}$, a diagonal algebra (discrete case);*
- *or $\mathfrak{A}$ is unitarily equivalent to $\mathfrak{D} \oplus L^\infty([0,1])$, where $\mathfrak{D}$ is a diagonal algebra (mixed case).*

Things are therefore clear concerning the MASAs in $\mathcal{L}(\mathbb{H})$, as the previous theorem provides a complete classification of those. In the case of von Neumann algebras of type II_1 however, things are more complicated and such a complete classification does not exist in spite of the numerous works on the subject.

Dixmier's Classification. We begin the discussion about MASAs of type II_1 von Neumann algebras by explaining Dixmier's classification [12], which considers the algebra generated by the *normaliser* of the MASA. Let us stress that this classification is not exhaustive. This presentation of Dixmier's classification will also give us the opportunity to state some results that will be of use below.

[12]: Dixmier (1954), *Sous-anneaux abéliens maximaux dans les facteurs de type fini*

Definition 11.1.8 (normaliser) *Let $\mathfrak{M}$ be a von Neumann algebra, and $\mathfrak{A}$ a von Neumann sub-algebra of $\mathfrak{M}$. We will denote by $N_{\mathfrak{M}}(\mathfrak{A})$ the* normaliser *of $\mathfrak{A}$ in $\mathfrak{M}$ which is defined as:*

$$N_{\mathfrak{M}}(\mathfrak{A}) = \{u \in \mathfrak{M} \mid u \text{ unitary}, u\mathfrak{A}u^* = \mathfrak{A}\}$$

We will denote by $\mathcal{N}_{\mathfrak{M}}(\mathfrak{A})$ the von Neumann algebra generated by $N_{\mathfrak{M}}(\mathfrak{A})$.

Definition 11.1.9 (normalising Groupoid) *Let $\mathfrak{M}$ be a von Neumann algebra and $\mathfrak{A}$ be a von Neumann sub-algebra of $\mathfrak{M}$. We will denote by $G_{\mathfrak{M}}(\mathfrak{A})$ the* normalising groupoid *of $\mathfrak{A}$ in $\mathfrak{M}$ which is defined as:*

$$G_{\mathfrak{M}}(\mathfrak{A}) = \{u \in \mathfrak{M} \mid uu^*u = u, uu^* \in \mathfrak{A}, u^*u \in \mathfrak{A}, u\mathfrak{A}u^* \subset \mathfrak{A}\}$$

We will denote by $\mathcal{G}_{\mathfrak{M}}(\mathfrak{A})$ the von Neumann algebra generated by $G_{\mathfrak{M}}(\mathfrak{A})$.

Definition 11.1.10 (Dixmier Classification) *Let $\mathfrak{M}$ be a factor, and $\mathfrak{P}$ a MASA in $\mathfrak{M}$. We distinguish three cases:*

1. *if $\mathcal{N}_{\mathfrak{M}}(\mathfrak{P}) = \mathfrak{M}$, we say that $\mathfrak{P}$ is* regular *(or* Cartan*);*
2. *if $\mathcal{N}_{\mathfrak{M}}(\mathfrak{P}) = \mathfrak{K}$, where $\mathfrak{K}$ is a factor distinct from $\mathfrak{M}$, we say that $\mathfrak{P}$ is* semi-regular*;*
3. *if $\mathcal{N}_{\mathfrak{M}}(\mathfrak{P}) = \mathfrak{P}$, we say that $\mathfrak{P}$ is* singular*.*

The following four theorems can be found in the literature. The first two theorems are due to Dye [151], and Jones and Popa [152]. They can be found along with their proofs in Sinclair and Smith book [15] about MASAs in finite factors. The third is a somewhat recent generalisation [153] of a result which was previously known to hold for singular MASAs.

[151]: Dye (1963), *On groups of measure preserving transformations. II.*

[152]: Jones et al. (1982), *Some properties of MASAs in factors*

[15]: Sinclair et al. (2008), *Finite von Neumann algebras and Masas*

[153]: Chifan (2007), *On the normalizing algebra of a masa in a II_1 factor*

Theorem 11.1.7 *Let $\mathfrak{M}$ be a von Neumann algebra with a faithful normal trace, and $\mathfrak{A}$ a MASA in $\mathfrak{M}$. Then the set $G_{\mathfrak{M}}(\mathfrak{A})$ is contained in the sub-vector space of $\mathfrak{M}$ generated by $N_{\mathfrak{M}}(\mathfrak{A})$.*

Corollary 11.1.8 *Under the hypotheses of the preceding theorem, the von Neumann algebras $\mathcal{N}_{\mathfrak{M}}(\mathfrak{A})$ and $\mathcal{G}_{\mathfrak{M}}(\mathfrak{A})$ are equal.*

Theorem 11.1.9 *Let $\mathfrak{M}$ be a type II_1 factor, and $\mathfrak{A}$ a MASA in $\mathfrak{M}$. Let $p, q \in \mathfrak{A}$ be projections of equal trace. Then, if $\mathcal{N}_{\mathfrak{M}}(\mathfrak{A})$ is a factor, there exists a partial isometry $v_0 \in G_{\mathfrak{M}}(\mathfrak{A})$ such that $p = v_0 v_0^*$ and $q = v_0^* v_0$.*

Theorem 11.1.10 *Let $\mathfrak{M}_1$ and $\mathfrak{M}_2$ be type II_1 factors. For $i = 1, 2$, let $\mathfrak{A}_i$ be a MASA in $\mathfrak{M}_i$. Then:*

$$\mathcal{N}_{\mathfrak{M}_1 \otimes \mathfrak{M}_2}(\mathfrak{A}_1 \otimes \mathfrak{A}_2) = \mathcal{N}_{\mathfrak{M}_1}(\mathfrak{A}_1) \otimes \mathcal{N}_{\mathfrak{M}_2}(\mathfrak{A}_2)$$

Lastly, the following theorem is due to Connes, Feldman and Weiss [154].

[154]: Connes et al. (1981), *An amenable equivalence relation is generated by a single transformation*

Theorem 11.1.11 *Let $\mathfrak{A}$, $\mathfrak{B}$ be two regular MASAs of the hyperfinite factor $\mathfrak{R}$ of type II_1. Then $\mathfrak{A}$ and $\mathfrak{B}$ are unitarily equivalent.*

As the classification proposed by Dixmier is neither exhaustive nor very precise, another invariant was introduced by Pukansky. We now define this invariant and discuss its relationship with Dixmier's classification.

Pukansky's Invariant. Pukansky [155] defined a numerical invariant for MASAs $\mathfrak{A}$ of a type II_1 factor $\mathfrak{R}$. Consider that $\mathfrak{R}$ is endowed with a faithful normal trace τ, and let J be the anti linear isometry $Jx = x^*$ onto $L^2(\mathfrak{R})$. Pukansky's invariant is based on the type I decomposition of $(\mathfrak{A} \cup J\mathfrak{A}J)'$. Indeed this algebra, as the commutant of an abelian algebra, is of type I and therefore can be decomposed as a sum of factors of type I_n (where n might be equal to ∞). The Pukansky invariant is then essentially the set of all values of n that appear in this decomposition.

[155]: Pukanszky (1960), *On maximal abelian subrings of factors of type II_1*

The following lemma justifies the definition of Pukansky's invariant. We define $e_{\mathfrak{A}}$ as the projection of $L^2(\mathfrak{R})$ onto $L^2(\mathfrak{A})$ and we will write $\mathfrak{B}_{\mathfrak{A}}$ the commutative algebra generated by $\mathfrak{A} \cup J\mathfrak{A}J$.

Lemma 11.1.12 *Let $\mathfrak{R}$ be a type II_1 factor represented onto $L^2(\mathfrak{R})$ and $\mathfrak{A}$ a MASA in $\mathfrak{R}$. Then $e_{\mathfrak{A}} \in \mathfrak{B}_{\mathfrak{A}}$ and $e_{\mathfrak{A}}$ is a central projection – i.e. a projection onto the center of the algebra – in $\mathfrak{B}'_{\mathfrak{A}}$.*

Definition 11.1.11 *Let $\mathfrak{A}$ be a MASA in a factor $\mathfrak{R}$ of type II_1. We define the* Pukansky invariant *$Puk(\mathfrak{A}, \mathfrak{R})$ of $\mathfrak{A}$ in $\mathfrak{R}$ – usually denoted by $Puk(\mathfrak{A})$ when the context is clear – as the set of all natural numbers $n \in \mathbf{N} \cup \{\infty\}$ such that $(1 - e_{\mathfrak{A}})\mathfrak{B}'_{\mathfrak{A}}$ has a non-zero type I_n part.*

By removing the projection $e_{\mathfrak{A}}$ from $\mathfrak{B}'_{\mathfrak{A}}$, we are erasing the part $\mathfrak{B}'_{\mathfrak{A}} e_{\mathfrak{A}} = \mathfrak{B}_{\mathfrak{A}} e_{\mathfrak{A}}$ which is abelian for all MASA $\mathfrak{A}$. This allows for a better invariant since its inclusion would add the integer 1 to all Pukansky invariants, rendering impossible the distinction between MASAs of invariant $\{2\}$ and those of invariant $\{1, 2\}$.

The Pukansky invariant satisfies that if $\mathfrak{A}$ and $\mathfrak{B}$ are two unitarily equivalent MASAs in a factor $\mathfrak{M}$ of type II_1, then $\mathrm{Puk}(\mathfrak{A}) = \mathrm{Puk}(\mathfrak{B})$. However, the reciprocal statement is not true. One can even find four MASAs $\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \mathfrak{D}$ in the type II_1 hyperfinite factor with equal invariants (all equal to $\{1\}$) where $\mathfrak{A}$ is regular, $\mathfrak{B}$ is semi-regular, $\mathfrak{C}$ is singular, and $\mathfrak{D}$ lies outside of Dixmier's classification. The Pukansky invariant is nonetheless very useful and some results about it will be used in this paper.

The four following theorems can be found in the book by Sinclair and Smith [15].

[15]: Sinclair et al. (2008), *Finite von Neumann algebras and Masas*

Proposition 11.1.13 *Let $\mathfrak{N}$ be a type II_1 factor and $\mathfrak{A}$ be a MASA in $\mathfrak{N}$. If $\mathfrak{A}$ is regular, then $Puk(\mathfrak{A}) = \{1\}$.*

Proposition 11.1.14 *Let $\mathfrak{N}$ be a type II_1 factor and $\mathfrak{A}$ be a MASA in $\mathfrak{N}$. The following statements are equivalent:*

- *$\mathfrak{A}$ is a MASA in $\mathcal{L}(L^2(\mathfrak{N}))$;*
- *$Puk(\mathfrak{A}) = \{1\}$.*

Proposition 11.1.15 *Let $\mathfrak{N}$ be a type II_1 factor and $\mathfrak{A}$ be a MASA in $\mathfrak{N}$.*

- *If $Puk(\mathfrak{A}) \subset \{2, 3, 4, \ldots, \infty\}$, then $\mathfrak{A}$ is singular.*
- *If $\mathcal{N}(\mathfrak{A}) \neq \mathfrak{A}$, then $1 \in Puk(\mathfrak{A})$.*

Proposition 11.1.16 *Let $\mathfrak{A}$ (resp. $\mathfrak{B}$) be a MASA in a factor $\mathfrak{M}$ (resp. $\mathfrak{N}$) of type II_1. Then:*

$$Puk(\mathfrak{A} \otimes \mathfrak{B}) = Puk(\mathfrak{A}) \cup Puk(\mathfrak{B}) \cup Puk(\mathfrak{A})Puk(\mathfrak{B})$$

where $Puk(\mathfrak{A})Puk(\mathfrak{B}) = \{a \times b \mid a \in Puk(\mathfrak{A}), b \in Puk(\mathfrak{B})\}$.

We have stated above that one can find four MASAs $\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \mathfrak{D}$ of the hyperfinite factor $\mathfrak{R}$ of type II_1 that all have the same Pukansky invariant and such that $\mathfrak{A}$ is regular, $\mathfrak{B}$ is semi-regular, $\mathfrak{C}$ is singular, and $\mathfrak{D}$ lies outside of Dixmier classification (we will say that $\mathfrak{D}$ is *non-Dixmier-classifiable*). The regular MASAs are necessarily of Pukansky invariant $\{1\}$, and one can show that such MASAs exist in the type II_1 hyperfinite factor $\mathfrak{R}$ (for instance by considering a construction of the hyperfinite factor as a crossed product), as explained in Sinclair and Smith book [15]). This gives the existence of a regular MASA $\mathfrak{A}$ in $\mathfrak{R}$ of Pukansky invariant $\{1\}$.

[15]: Sinclair et al. (2008), *Finite von Neumann algebras and Masas*

On the other hand, Stuart White [156] showed that the so-called *Tauer MASAs* all have as Pukansky invariant the singleton $\{1\}$. And it is known that there exists singular Tauer MASAs [157] and semi-regular Tauer MASAs [156] in the hyperfinite factor $\mathfrak{R}$. This gives the existence of a semi-regular MASA $\mathfrak{B}$ and a singular MASA $\mathfrak{C}$ such that $\mathrm{Puk}(\mathfrak{B}) = \mathrm{Puk}(\mathfrak{C}) = \{1\}$.

[156]: White (2006), *Tauer Masas in the Hyperfinite II_1 Factor*

[157]: White et al. (2007), *A continuous path of singular masas in the hyperfinite II_1 factor*

[156]: White (2006), *Tauer Masas in the Hyperfinite II_1 Factor*

Lastly, let us show that the existence of singular MASAs with Pukansky invariant equal to $\{1\}$ implies the existence of non-Dixmier-classifiable MASAs whose Pukansky invariant is equal to the singleton $\{1\}$. Indeed, if $\mathfrak{A}$ is a MASA with $\mathrm{Puk}(\mathfrak{A}) = \{1\}$, we can consider $\mathfrak{A} \otimes \mathfrak{Q}$ where $\mathfrak{Q}$ is a regular MASA (thus $\mathrm{Puk}(\mathfrak{Q}) = \{1\}$) of $\mathfrak{R} \otimes \mathfrak{R}$. We then have that

$\mathrm{Puk}(\mathfrak{A}\otimes\mathfrak{Q}) = \{1\}$ by Proposition 11.1.16, and moreover, by Theorem 11.1.10, we have:

$$\mathcal{N}_{\mathfrak{R}\otimes\mathfrak{R}}(\mathfrak{A}\otimes\mathfrak{Q}) = \mathcal{N}_{\mathfrak{R}}(\mathfrak{A})\otimes\mathcal{N}_{\mathfrak{R}}(\mathfrak{Q}) = \mathfrak{A}\otimes\mathfrak{R}$$

But the center of $\mathfrak{A}\otimes\mathfrak{R}$ is equal to $\mathfrak{A}\otimes\mathbf{C}$ since $\mathfrak{A}$ is commutative and the commutant of a tensor product is equal to the tensor product of the commutants (a result due to Tomita [158]). Thus $\mathfrak{A}\otimes\mathfrak{R}$ is not a factor, which implies that $\mathfrak{A}\otimes\mathfrak{Q}$ is neither regular nor semi-regular. Since $\mathfrak{A}\otimes\mathfrak{Q}$ is obviously not equal to $\mathfrak{A}\otimes\mathfrak{R}$, we know that $\mathfrak{A}\otimes\mathfrak{Q}$ is not singular: it is therefore non-Dixmier-classifiable. Eventually, as $\mathfrak{R}\otimes\mathfrak{R}$ is isomorphic to $\mathfrak{R}$, it is enough to choose such an isomorphism ϕ to define $\mathfrak{D} = \phi(\mathfrak{A}\otimes\mathfrak{Q})$ a MASA in $\mathfrak{R}$ which is non-Dixmier-classifiable and such that $\mathrm{Puk}(\mathfrak{D}) = \{1\}$.

[158]: Tomita (1967), *Quasi-Standard von Neumann Algebras*

Dixmier's Classification and Linear Logic

In this section, we will prove the main technical result of this paper. We first prove that no non-trivial interpretation of linear logic proofs exists w.r.t. a singular outlook. We then show that semi-regular outlooks provide enough structure to interpret the exponential-free fragment of linear logic. Lastly, we show that regular outlooks provide the structure for the interpretation of exponential connectives.

Singular MASAs

First, we show that every promising project w.r.t. an outlook $\mathfrak{P}$ which is a singular MASA in $\mathfrak{R}_{0,1}$ is trivial, i.e. its operator is equal to 0. We will first show two lemmas that will be of use afterwards. We will write $\mathfrak{A}_p$ the von Neumann algebra $p\mathfrak{A}p$ where p is a projection, i.e. the restriction of $\mathfrak{A}$ to the subspace corresponding to p.

Lemma 11.1.17 *Let $\mathfrak{A}$ be a MASA in a factor $\mathfrak{M}$, and let p be a projection in $\mathfrak{A}$. If $A \in \mathfrak{M}$ normalises $\mathfrak{A}$, and $Ap = pA$, then A normalises $\mathfrak{A}_p$.*

Proof. We pick x in $\mathfrak{A}_p \subset \mathfrak{A}$. Then $AxA^* = y \in \mathfrak{A}$ since A normalises $\mathfrak{A}$. Moreover, $yp = AxA^*p = Ax(pA)^* = Ax(Ap)^* = AxpA^* = AxA^* = y$, and a similar argument shows that $py = y$. Thus $y = pyp$ and $y \in \mathfrak{A}_p$. ❀

Remark 11.1.3 This result implies that $pG_{\mathfrak{M}}(\mathfrak{A})p \subset G_{\mathfrak{M}_p}(\mathfrak{A}_p)$.

The following lemma is of particular importance since it will allow us to reduce our study to the case of a factor of type II_1, and thus to use Chifan's result (Theorem 11.1.10). The fact that $\mathfrak{A}$ is abelian is essential here. Indeed, one can even find finite-dimensional counter-examples in the case $\mathfrak{A}$ is a non-commutative singular von Neumann sub-algebra. For instance, the subfactor $\mathfrak{A} = \mathfrak{M}_2(\mathbf{C}) \oplus \mathbf{C}$ of $\mathfrak{M}_3(\mathbf{C})$ is singular : $\mathfrak{N}_{\mathfrak{M}_3(\mathbf{C})}(\mathfrak{A}) = \mathfrak{A}$. Picking the projection $p = 0 \oplus 1 \oplus 1$ in $\mathfrak{M}_3(\mathbf{C})$, we have that $\mathfrak{A}_p$ is not singular in $(\mathfrak{M}_3(\mathbf{C}))_p$ – it is even a regular sub-algebra – since $\mathfrak{N}_{\mathfrak{M}_2(\mathbf{C})}(\mathfrak{A}_p) = \mathfrak{M}_2(\mathbf{C})$.

Lemma 11.1.18 *Let $\mathfrak{A}$ be a MASA in a von Neumann algebra $\mathfrak{M}$, and p a projection in $\mathfrak{A}$. Then $\mathfrak{A}_p$ is a maximal abelian sub-algebra of $\mathfrak{M}_p$. Moreover, if $\mathfrak{A}$ is singular, $\mathfrak{A}_p$ is singular.*

Theorem 11.1.19 (Singular Outlooks and Soundness) *If $\mathfrak{P}$ is a singular MASA in $\mathfrak{R}_{0,1}$, then every promising project w.r.t. the outlook $\mathfrak{P}$ is trivial.*

Remark 11.1.4 Without the additional condition (about projections) in the definition of promising projects, it would be easy to find non-trivial hyperfinite projects which are promising w.r.t. a singular MASA $\mathfrak{P}$ in $\mathfrak{R}_{0,1}$. Indeed, let p, q be two projections in $\mathfrak{P}$. Then the project $\mathfrak{a} = (p+q, 0, 1, \mathbf{C}, p+q)$ would then clearly be promising $\mathfrak{P}$.
One might wonder however if this condition could be weakened, asking for instance that the trace of A be zero. This condition would not be sufficient, since for all projections $p, q \in \mathfrak{P}$ such that $\mathrm{tr}(p) = \mathrm{tr}(q)$, the hyperfinite project $\mathfrak{b}$ defined as $(p+q, 0, 1, \mathbf{C}, p-q)$ would then be promising as $\mathrm{tr}(p-q) = \mathrm{tr}(p) - \mathrm{tr}(q) = 0$.
Another weaker condition would be: for all projections $\pi \in \mathfrak{P} \otimes \mathfrak{Q}$, $\mathrm{tr}(\pi A) = 0$. However, the following project would then be promising w.r.t. $\mathfrak{P}$ when $p, q \in \mathfrak{P}$ are projections:

$$\mathfrak{c} = (p+q, 0, \mathrm{tr}, \mathfrak{M}_2(\mathbf{C}), \begin{pmatrix} 0 & (p+q)_{\otimes 1_{\mathfrak{R}}} \\ (p+q)_{\otimes 1_{\mathfrak{R}}} & 0 \end{pmatrix})$$

All those projects may be considered as successful, so why do we want to exclude them? The reason can be found in the relationship between the GoI interpretation of proofs and the theory of proof nets. Indeed, as it is explained in both Girard and the author's work on the interpretation of multiplicatives [16, 140], the GoI interpretation of a proof corresponds to a representation of the axiom links of the corresponding proof net. As a consequence, a successful project should be understood intuitively as a set of axiom links, i.e. a partial symmetry not containing any fixed point – something that corresponds to the fact that for all non-zero vector ξ the symmetry S satisfies $S\xi \neq \xi$. In this respect, the first projects considered above should therefore not be considered as successful as they obviously do not satisfy this property. The reason why last project should also not be considered as successful is, however, more involved since it is a symmetry not containing fixed points. In this case, however, the vectors ξ and $S\xi$ differ only from the *dialect*[2], i.e. the second projection of the vector. Thinking about proof nets again, this second projection, the dialect, corresponds to *slices* in additive proof nets [159]. This last project represents, in this respect, an axiom link between a formula A in a slice s_1 and the same formula A in a different slice s_2. The reader familiar with additive proof nets should now be convinced that such a project should not be successful, as it represents something which is not a valid axiom link.

[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*
[140]: Girard (1987), *Multiplicatives*

2: The *dialect* is the name given by Girard to the algebra $\mathcal{L}(\mathbf{C}^S)$ where S is the set of control states in the terminology of this document.
[159]: Girard (1995), *Proof nets: the parallel syntax for proof theory*

Non-Singular MASAs

In this section, we consider chosen an outlook $\mathfrak{P}$ which is either a regular or a semi-regular MASA in $\mathfrak{R}_{0,1}$. We will show a full soundness result for the sequent calculus $\mathrm{MALL}_{\mathbf{T},0}$ (Figure 11.1), i.e. we interpret formulas and sequents as conducts and proofs as hyperfinite projects and we show that for all proof π of a sequent $\vdash \Gamma$, the interpretation $\|\pi\|$ is a promising project which belongs to $\|\Gamma\|$.

$$\frac{}{\vdash X_i^{\perp}, X_i}\ \text{Ax} \qquad \frac{\vdash A, \Delta \qquad \vdash A^{\perp}, \Gamma}{\vdash \Delta, \Gamma}\ \text{Cut}$$

$$\frac{\vdash A, \Delta \qquad \vdash B, \Gamma}{\vdash A \otimes B, \Delta, \Gamma}\ \otimes \qquad \frac{\vdash A, B, \Gamma}{\vdash A \parr B, \Gamma}\ \parr$$

$$\frac{\vdash A_i, \Gamma}{\vdash A_0 \oplus A_1, \Gamma}\ \oplus^i \qquad \frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B}\ \&$$

$$\frac{}{\vdash \top, \Gamma}\ \top \qquad \text{No rules for } \mathbf{0}.$$

Figure 11.1.: Sequent calculus $\text{MALL}_{\mathbf{T},\mathbf{0}}$

The Sequent Calculi $\text{MALL}_{\mathbf{T},\mathbf{0}}$**.** We will briefly define the sequent calculus $\text{MALL}_{\mathbf{T},\mathbf{0}}$ for which we show a soundness result. This sequent calculus was defined in order to prove a soundness result for interaction graphs [17]. This is the usual sequent calculus for multiplicative-additive linear logic without multiplicative units (but including additive units). Although multiplicative units can be dealt with, they need a more involved sequent calculus with *polarised formulas* that deals with exponential connectives [14]. A soundness result for this more involved calculus exists, but the result does not justifies the amount of work needed to correctly define the calculus. The interested reader can have a look at the author's work on exponentials [14] to persuade herself that this extended result holds as well.

[17]: Seiller (2016), *Interaction graphs: Additives*

[14]: Seiller (2019), *Interaction Graphs: Exponentials*

In earlier works [14, 16–18], we took into account the locativity of the framework by defining a *localised sequent calculus* $\text{locMALL}_{\mathbf{T},\mathbf{0}}$ for which formulas have a specific location and rules are subject to constraints on the locations of the formulas appearing in the sequents. This localised version of the sequent calculus is used in order to prove a soundness result more easily as it presupposes the locativity constraints of the GoI model. The soundness result for the usual non-localised calculus is then obtained by noticing that every formula, thus sequent, and every proof can be 'localised', i.e. interpreted as a formula, sequent or proof of the localised calculus. We will here define directly localised interpretations of the non-localised sequent calculus in order to limit the space needed to show the results.

[14]: Seiller (2019), *Interaction Graphs: Exponentials*
[16]: Seiller (2012), *Interaction Graphs: Multiplicatives*
[17]: Seiller (2016), *Interaction graphs: Additives*
[18]: Seiller (2017), *Interaction Graphs: Graphings*

Let us fix $\mathcal{V} = \{X_i\}_{i \in \mathbf{N}}$ a set of variables.

Definition 11.1.12 (Formulas of $\text{MALL}_{\mathbf{T},\mathbf{0}}$) *The formulas of* $\text{MALL}_{\mathbf{T},\mathbf{0}}$ *are defined by the following grammar:*

$$F := X_i \mid X_i^{\perp} \mid F \otimes F \mid F \parr F \mid F \,\&\, F \mid F \oplus F \mid \mathbf{0} \mid \mathbf{T}$$

where the X_i *are variables.*

Definition 11.1.13 (Proofs of $\text{MALL}_{\mathbf{T},\mathbf{0}}$) *A proof of* $\text{MALL}_{\mathbf{T},\mathbf{0}}$ *is a derivation obtained from the sequent calculus rules shown in Figure 11.1.*

Interpretation of Formulas.

Definition 11.1.14 (Delocations) *Let* p, q *be projections in* $\mathfrak{P}$*. A* delocation *from* p *onto* q *is a partial isometry* $\theta : p \to q$ *such that* $\theta \in G_{\mathfrak{R}_{0,1}}(\mathfrak{P})$*.*

To interpret the sequent calculus, we will actually work with the MASA $\mathfrak{P} \oplus \mathfrak{P}$ of the algebra $\mathfrak{M}_2(\mathfrak{R}_{0,1})$ in order to distinguish a *primitive space* (the first component of the direct sum $\mathfrak{P} \oplus \mathfrak{P}$) and an *interpretation space* (the second component of the direct sum). Interpretations of proofs and formulas will be elements of the interpretation space, hence the interpretation will in fact take place in $\mathfrak{R}_{0,1}$, while the primitive space will be used in order to define correctly the syntax. The following proposition shows that, since the interpretations will be hyperfinite projects defined in the second component of the sum $\mathfrak{P} \oplus \mathfrak{P}$, the fact that they are promising w.r.t. $\mathfrak{P} \oplus \mathfrak{P}$ in $\mathfrak{M}_2(\mathfrak{R}_{0,1})$ implies that their restriction to $\mathfrak{R}_{0,1}$ (the second component) is promising w.r.t. $\mathfrak{P}$.

Proposition 11.1.20 (Restriction) *Let* $\mathfrak{a} = (p, 0, \mathrm{tr}, \mathfrak{R}, A)$ *be a promising project w.r.t.* $\mathfrak{P} \oplus \mathfrak{P} \subset \mathfrak{M}_2(\mathfrak{R}_{0,1})$ *such that* $p \leqslant 0 \oplus 1$. *Then* $A(0 \oplus 1) = (0 \oplus 1)A = A$, *and* $\mathfrak{a}$ *is a promising project w.r.t.* $\mathfrak{P} \subset (\mathfrak{M}_2(\mathfrak{R}_{0,1}))_{0 \oplus 1} \simeq \mathfrak{R}_{0,1}$.

Let us now define variables. We pick a family of pairwise disjoint projections $(p_i)_{i \in \mathbf{N}}$. The projections $p_i \oplus 0$ will be called the *primitive locations* of the variables, and one should think of this as our actual set of variables.

Definition 11.1.15 (Variable names) *A* variable name *is an integer* $i \in \mathbf{N}$ *denoted by capital letters* X, Y, Z, *etc. A* variable *is a pair* $X_\theta = (X, \theta)$ *where* X *is a variable name, i.e. an integer* i, *and* θ *is a relocation of* $p_i \oplus 0$ *onto a projection* $0 \oplus q_{X_\theta}$. *The projection* $0 \oplus q_{X_\theta}$ *is referred to as the* location *of the variable, and we will sometimes allow ourselves to forget about the first component and simply write* q_{X_θ}.

We now define the interpretation of formulas.

Definition 11.1.16 (Interpretation Basis) *An* interpretation basis *is a map* δ *associating to each variable name* $X = i$ *a dichology* $\delta(X)$ *of carrier the primitive location* p_i *of* X. *This map extends to a function* $\bar{\delta}$ *which associates, to each variable* X_θ, *the dichology* $\bar{\delta}(X_\theta) = \theta(\delta(X))$ *of carrier* q_{X_θ} *– the location of* X_θ.

Definition 11.1.17 (Interpretation of Formulas) *The interpretation* $\|F\|_\delta$ *of a formula* F *along the interpretation basis* δ *is defined inductively as follows:*

- $F = X_\theta$. *We define* $\|F\|_\delta$ *as the dichology* $\bar{\delta}(X_\theta)$ *of carrier* q_{X_θ};
- $F = X_\theta^\perp$. *We define* $\|F\|_\delta = (\|X_\theta\|_\delta)^\perp$, *a dichology of carrier* q_{X_θ};
- $F = A \star B$ ($\star \in \{\otimes, \parr, \&, \oplus\}$). *We define* $\|F\|_\delta = \|A\|_\delta \star \|B\|_\delta$, *a dichology of carrier* $p + q$, *where* p *and* q *are the respective carriers of* $\|A\|_\delta$ *and* $\|B\|_\delta$;
- $F = \mathbf{T}$ *(resp.* $F = \mathbf{0}$*). We define* $\|F\|_\delta = \mathbf{T}_0$ *(resp.* $\mathbf{0}_0$*), the* full conduct *(resp. the* empty conduct *of carrier* 0.

Definition 11.1.18 (Interpretation of Sequents) *A sequent* $\vdash \Gamma$ *will be interpreted as the* $\parr$ *of formulas in* Γ, *denoted by* $\parr\!\!\!\!\bigcirc\, \Gamma$.

Interpretation of proofs. The introduction rule of the $\parr$ as well as the exchange rule will have a trivial interpretation, since premise and conclusion sequents are interpreted by the same dichology: due to locativity, the commutativity and associativity of $\parr$ are real equalities and not morphisms. Similarly, rules for $\oplus$ have an easy interpretation as it

suffices to extend the carrier of the project interpreting the premise to define the interpretation of the conclusion. Moreover, the rule $\top$ has a straightforward interpretation as the project $(0, 0, 1_{\mathbf{C}}, \mathbf{C}, 0)$. Axioms will be easily interpreted by delocations, whose existence is ensured by Theorem 11.1.9. The case of cut has already been treated in Proposition 11.1.2, and we therefore only need to deal with the introduction rules of $\otimes$ and &.

Given two hyperfinite projects $\mathfrak{f}$ and $\mathfrak{g}$ in the interpretations of the premises of a tensor ($\otimes$) introduction rule, we will define a hyperfinite project $\mathfrak{h}$ in the interpretation of the conclusion. The operation that naturally comes to mind is to define this project as the tensor product of the projects $\mathfrak{f}$ and $\mathfrak{g}$. It turns out that this interpretation of the $\otimes$ introduction rule is perfectly satisfactory: the following proposition shows that the project $\mathfrak{h}$ defined as $\mathfrak{f} \otimes \mathfrak{g}$ is a project in the interpretation of the conclusion.

Proposition 11.1.21 (Interpretation of the Tensor Rule) *Let* $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ *be conducts of respective carriers* p_A, p_B, p_C, p_D*. We have the following inclusion:*

$$((\mathbf{A} \multimap \mathbf{B}) \otimes (\mathbf{C} \multimap \mathbf{D})) \subset ((\mathbf{A} \otimes \mathbf{C}) \multimap (\mathbf{B} \otimes \mathbf{D}))$$

We will now interpret the introduction rule for &. We will interpret a proof ending with a & introduction rule by the sum of the projects $\mathfrak{f}_{p+q}$ and $\mathfrak{g}_{p+q}$, where $\mathfrak{f}$ and $\mathfrak{g}$ – of respective carriers p and q – are the interpretations of the sub-proofs whose conclusions are the premises of the & rule. In order to perform this operation, it is necessary to first delocalise the interpretations of the premises as the premises do not have disjoint locations. Once this relocation is done, we can define the project $\mathfrak{h}$ as $\theta_1(\mathfrak{f})$ & $\theta_2(\mathfrak{g})$ – where θ_1 and θ_2 are the delocations just mentioned. We then apply the project implementing distributivity in order to superpose the contexts. We refer the reader to the interpretation of proofs of $MALL_{\mathbf{T},\mathbf{0}}$ in interaction graphs [17] for a more thorough explanation of this.

[17]: Seiller (2016), *Interaction graphs: Additives*

For the next result, we will be needing a proposition shown in earlier work [17] in a different setting, but whose proof easily adapts to the matricial (as well as the hyperfinite) GoI model. This proposition states any element of a dichology[3] $\mathbf{A}$ & $\mathbf{B}$ is *observationally equivalent* to a sum $\mathfrak{a}_1 \otimes \mathfrak{o}_q + \mathfrak{a}_2 \otimes \mathfrak{o}_p$ with $\mathfrak{a}_1 \in \mathbf{A}$, $\mathfrak{a}_2 \in \mathbf{B}$, where observational equivalence is defined, e.g. on elements of a conduct $\mathbf{A}$, as follows:

$$\mathfrak{a} \sim_{\mathbf{A}} \mathfrak{a}' \Leftrightarrow \forall t \in \mathbf{A}^{\perp}, [\![a, t]\!]_{\text{mat}} = [\![a', t]\!]_{\text{mat}}$$

We recall that this notion of equivalence is a congruence, i.e. if $\mathfrak{a} \sim_{\mathbf{A}} \mathfrak{a}'$, then for all $\mathfrak{f} \in \mathbf{A} \multimap \mathbf{B}$ we have $\mathfrak{f} :: \mathfrak{a} \sim_{\mathbf{B}} \mathfrak{f} :: \mathfrak{a}'$. In particular, if $\mathfrak{a} \sim_{\mathbf{A}} \mathfrak{a}'$ then $\mathfrak{a} \otimes \mathfrak{b} \sim_{\mathbf{A} \otimes \mathbf{B}} \mathfrak{a}' \otimes \mathfrak{b}$ for all $\mathfrak{b} \in \mathbf{B}$.

[17]: Seiller (2016), *Interaction graphs: Additives*

3: We must inform the reader that the terminology here differs from the cited paper: what we call here a dichology is called a *behaviour* in the interaction graphs constructions [17].

Proposition 11.1.22 *Let* $\mathbf{A}, \mathbf{B}$ *be dichologies of respective carriers* p, q*. For any element* $\mathfrak{a} \in \mathbf{A}$ & $\mathbf{B}$*, there exists elements* $\mathfrak{a}_1 \in \mathbf{A}$ *and* $\mathfrak{a}_2 \in \mathbf{B}$ *such that* $\mathfrak{a}_1 \otimes \mathfrak{o}_q + \mathfrak{a}_2 \otimes \mathfrak{o}_p \sim_{\mathbf{A}\&\mathbf{B}} \mathfrak{a}$*.*

Corollary 11.1.23 *Let* $\mathbf{A}, \mathbf{B}, \mathbf{C}$ *be dichologies of carriers* p, q, r *respectively, and* $\mathfrak{f}$ *a project of carrier* $p + q + r$*. If* $\mathfrak{a}$ *maps every sum* $\mathfrak{a}_1 \otimes \mathfrak{o}_q + \mathfrak{a}_2 \otimes \mathfrak{o}_p$ *to an element of* $\mathbf{C}$*, then* $\mathfrak{f}$ *belongs to* $(\mathbf{A}$ & $\mathbf{B}) \multimap \mathbf{C}$*.*

Proposition 11.1.24 (Interpretation of the & Rule) *Let* $\mathbf{A}, \mathbf{B}, \mathbf{C}$ *be dichologies of respective pairwise disjoint carriers* p_A, p_B, p_C*, and let* $\phi(\mathbf{A})$ *be a delocation of* $\mathbf{A}$*, with* $\phi \in G_{\mathfrak{R}_{0,1}}(\mathfrak{P})$*, whose carrier is a projection disjoint from the projections* $\mathbf{A}, \mathbf{B}, \mathbf{C}$*. Then for all delocations*[4] $\theta_1, \theta_2, \theta_3$ *in* $G_{\mathfrak{R}_{0,1}}(\mathfrak{P})$*, there exists a project* $\mathfrak{With}$ *in the dichology:*

$$((\mathbf{A} \multimap \mathbf{B}) \,\&\, (\phi(\mathbf{A}) \multimap \mathbf{C})) \multimap (\theta_{\mathbf{1}}(\mathbf{A}) \multimap (\theta_{\mathbf{2}}(\mathbf{B}) \,\&\, \theta_{\mathbf{3}}(\mathbf{C})))$$

Moreover, $\mathfrak{With}$ *is promising w.r.t. the outlook* $\mathfrak{P}$*.*

4: Supposing of course that the carriers are pairwise disjoint.

Remark 11.1.5 The interpretation of the & introduction rule will therefore be defined as the relatively complex construction[5] $\mathfrak{f}\overline{\&}\mathfrak{g} = [\mathfrak{With}](\theta_1(\mathfrak{f}) \,\&\, \theta_2(\mathfrak{g}))$. This construction should not hide however the simplicity of the underlying idea. Indeed, given two projects $\mathfrak{f} = (p + r, 0, \phi, \mathfrak{F}, F)$ and $\mathfrak{g} = (q + r, 0, \gamma, \mathfrak{G}, G)$, we are just constructing the project:

5: We recall that θ_1 and θ_2 are well-chosen delocations.

$$\mathfrak{f}\overline{\&}\mathfrak{g} = (p + q + r, 0, \frac{1}{2}(\phi \oplus \gamma), \mathfrak{F} \oplus \mathfrak{G}, F \oplus G)$$

Soundness. In order to state and show the full soundness result, we first define the interpretation of proofs.

Definition 11.1.19 (Interpretations of Proofs) *We inductively define the interpretation of a proof* Π*:*

- *if* Π *is an axiom rule introducing the sequent* $\vdash X_\theta; X_\phi$ *(*θ, ϕ *are disjoint), we define the interpretation* $\Pi^\bullet$ *as the project* $(q_{X_\theta} + q_{X_\phi}, 0, \mathrm{tr}, \mathfrak{R}, \theta\phi^* + \phi\theta^*)$*;*
- *if* Π *is obtained by application of a rule* $\mathfrak{R}$*, or an exchange rule, to a proof* Π_1*, we define* $\Pi^\bullet = \Pi_1^\bullet$*;*
- *if* Π *is obtained by applying a* $\oplus$ *rule to a proof* Π_1 *whose interpretation's carrier is* p*, then* $\Pi^\bullet = (\Pi_1^\bullet)_{p+q}$ *where* q *is the carrier of the interpretation of the introduced formula;*
- *if* Π *is obtained by applying a cut rule between two proofs* Π_1 *and* Π_2*, then* $\Pi^\bullet = \Pi_1^\bullet :: \Pi_2^\bullet$*;*
- *if* Π *is obtained by applying a* $\otimes$ *introduction rule to the proofs* Π_1 *and* Π_2*, then* $\Pi^\bullet = \Pi_1^\bullet \otimes \Pi_2^\bullet$*;*
- *if* Π *is obtained by the application of a* & *rule on the proofs* Π_1 *and* Π_2 *interpreted by projects* $\Pi_1^\bullet$ *and* $\Pi_2^\bullet$*, we then define* $\Pi^\bullet = [\mathfrak{With}](\theta_1(\Pi_1^\bullet) \,\&\, \theta_2(\Pi_2^\bullet))$ *where* θ_1, θ_2 *are delocations of* $\Pi_1^\bullet$ *and* $\Pi_2^\bullet$ *onto disjoint projections, and where* $\mathfrak{With}$ *is the project whose existence is ensured by Proposition 11.1.24.*

Theorem 11.1.25 (Full Soundness) *Let* π *be a proof of the sequent* $\vdash \Gamma$ *in* $MALL_{\mathbf{T},\mathbf{0}}$*, and* δ *an interpretation basis. Then the interpretation* $\pi^\bullet$ *of* π *is a promising project w.r.t.* $\mathfrak{P}$ *in the interpretation* $\|\vdash \Gamma; A\|_\delta$ *of* $\vdash \Gamma$*.*

Regular MASAs

To interpret exponentials of Elementary Linear Logic (ELL), we consider the construction proposed by Girard [7]. There is one major problem with this construction, however. Indeed, if $\mathfrak{a}$ is a promising hyperfinite project w.r.t. the outlook $\mathfrak{P}$, it is clear that the hyperfinite project $!_\Omega\mathfrak{a}$ is promising w.r.t the outlook $\Omega(\mathfrak{P} \otimes \mathfrak{Q})$ where $\mathfrak{Q}$ is a MASA in $\mathfrak{R}$. However,

[7]: Girard (2011), *Geometry of Interaction V: Logic in the Hyperfinite Factor.*

if it is obvious that $\Omega(\mathfrak{P} \otimes \mathfrak{Q})$ is a MASA in $\mathfrak{R}_{0,1}$, it won't be true, in general, that $\Omega(\mathfrak{P} \otimes \mathfrak{Q}) = \mathfrak{P}$. As those are both MASAs in $\mathfrak{R}_{0,1}$, the two algebras $\Omega(\mathfrak{P} \otimes \mathfrak{Q})$ and $\mathfrak{P}$ are diffuse abelian von Neumann algebras, thus isomorphic *as von Neumann algebras*. This is however too weak a result as this isomorphism is not in general realised by a unitary operator, a necessary condition for an adequate interpretation of the promotion rule.

Proposition 11.1.26 *Let $\mathfrak{a}$ be a promising project w.r.t. the outlook $\mathfrak{P}$. Suppose that $\mathfrak{P}$ is a regular MASA in $\mathfrak{R}_{0,1}$. Then there exists a partial isometry u such that $u\Omega(A)u^*$ is a partial symmetry in the normalising groupoid of $\mathfrak{P}$.*

One can notice that the interpretations of the contraction and functorial promotion rules only use promising projects w.r.t. $\mathfrak{P}$. From this and the preceding proposition, one can easily show an extension of the soundness result stated above for the sequent calculi $\mathrm{ELL}_{\mathrm{pol}}$ and $\mathrm{ELL}_{\mathrm{comp}}$ considered in the author's work on interaction graphs [14] as soon as the outlook is a regular MASA. Let us notice that this proposition do not depend on the morphism Ω chosen to define the exponentials (the soundness result however depends on Ω since not all choices of morphisms would allow for the interpretation of functorial promotion).

[14]: Seiller (2019), *Interaction Graphs: Exponentials*

It is then natural to ask oneself if the converse of this result holds, i.e. if the fact that $\mathfrak{P}$ is not regular implies that one cannot interpret (at least one) exponential connective. We will not fully answer this question in this paper, but we will discuss it anyway.

Let us first consider the Pukansky invariant of the outlook $\mathfrak{P}$ and of the sub-algebra $\Omega(\mathfrak{P} \otimes \mathfrak{Q})$ (using the same notations as in the preceding proof). It is known[6] that there exists singular MASAs in $\mathfrak{R}$ whose Pukansky invariant is included in $\{2, 3, \dots, \infty\}$, and the sub-algebra $\mathfrak{Q}$ satisfies $\mathrm{Puk}(\mathfrak{Q}) = \{1\}$ since it is regular (Proposition 11.1.13). Using Proposition 11.1.16, we get that $\mathrm{Puk}(\Omega(\mathfrak{P} \otimes \mathfrak{Q}))$ contains 1, and it is therefore impossible in this case that $\Omega(\mathfrak{P} \otimes \mathfrak{Q})$ and $\mathfrak{P}$ be unitarily equivalent.

6: White [160] showed that all subset of $\mathbf{N} \cup \{\infty\}$ is the Pukansky invariant of a MASA in $\mathfrak{R}$.

However, the Pukansky invariant of a semi-regular MASA is a subset of $\mathbf{N} \cup \{\infty\}$ that contains 1 (from Proposition 11.1.15). Then, by using Proposition 11.1.16, one shows that in this case $\mathrm{Puk}(\Omega(\mathfrak{P} \otimes \mathfrak{Q})) = \mathrm{Puk}(\mathfrak{P})$. It is therefore not possible to show the reciprocal statement of Proposition 11.1.26 in this manner. We conjecture that there exist perennializations Ω and semi-regular outlooks $\mathfrak{P}$ such that (the equivalent of) Proposition 11.1.26 holds. We also conjecture that there exists perennializations Ω and semi-regular outlooks $\mathfrak{P}$ such that the (equivalent of) Proposition 11.1.26 does not hold. A more interesting question would be to know if for all perennializations (and therefore the one defined by Girard) there exists a semi-regular outlook $\mathfrak{P}$ such that the (equivalent of) Proposition 11.1.26 does not hold.

Conclusion

The results obtained in this section can be combined into the following theorem, which constitute the main technical result of this paper.

Theorem 11.1.27 *Let $\mathfrak{P}$ be a maximal abelian sub-algebra of $\mathfrak{R}_{0,1}$. Then:*

- *if $\mathfrak{P}$ is singular, there are no non-trivial interpretations of any fragment of linear logic by promising hyperfinite projects w.r.t. $\mathfrak{P}$;*
- *if $\mathfrak{P}$ is semi-regular, one can interpret soundly multiplicative-additive linear logic (MALL) by promising hyperfinite projects w.r.t. $\mathfrak{P}$;*
- *if $\mathfrak{P}$ is regular, one can interpret soundly elementary linear logic (ELL) by promising hyperfinite projects w.r.t. $\mathfrak{P}$.*

11.2. Graphings and complexity: the set-up

We now turn to a series of results which can be understood as a continuation of the previous theorem. Indeed, the latter is partial: in the regular case the logical system captured is unclear and depends on the particular algebras considered. In other words, a regular maximal abelian sub-algebra does not in general model exponentials properly, and one can expect to have *intermediate* systems, between MALL and ELL. However, pursuing the characterisation of complexity classes through von Neumann algebras would be difficult, as the theory of maximal abelian von Neumann sub-algebras is very involved, full of open questions, and does not provide a clear connection with programs or proof systems.

This is where graphings come in. As we have seen earlier, those can be understood as representing operators in a group measure space von Neumann algebra. This led me to try and provide a more precise correspondence between maximal abelian sub-algebras and logical systems of varying expressivity. It turns out that one proper way of expressing this correspondence is through the complexity class captured by the type of predicates over the natural numbers in the logical system obtained through linear realisability techniques.

Integers and Machines

We now review some definitions necessary to define the characterisation of complexity classes in the Interaction Graphs models [45]. We start by the representation of binary words, which is related [39, 40] to the type of binary lists in Elementary Linear Logic [144, 161]:

$$\text{BList} := \forall X\, !(X \multimap X) \multimap !(X \multimap X) \multimap !(X \multimap X).$$

Intuitively a binary word, say 001, is represented as a program that takes two functions f_0 and f_1 of type $X \to X$, and produces the function $f_0 \circ f_0 \circ f_1$. This program can also be defined (in Krivine's notation) as the lambda-term $\lambda f_0 \lambda f_1 \lambda x.(f_0)(f_0)(f_1)x$. This lambda-term corresponds to a proof of BList in Elementary Linear Logic which contains exactly four axioms (Figure 11.2). These four axioms give rise to the representation of the proof as a graphing in the Interaction Graph model. The latter representation uses six subspaces, corresponding to the six occurrences of X in the formula BList. We will first introduce some notations for these subspaces and then define formally the graphing representations of binary words.

[45]: Seiller (2018), *Interaction Graphs: Nondeterministic Automata*
[39]: Aubert et al. (2016), *Characterizing co-NL by a group action*
[40]: Aubert et al. (2016), *Logarithmic Space and Permutations*
[144]: Girard (1995), *Light Linear Logic*
[161]: Danos et al. (2003), *Linear Logic & Elementary Time*

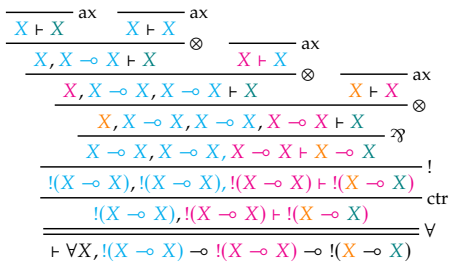


Figure 11.2.: Proof corresponding to $\lambda f_0.\lambda f_1.\lambda x.(f_0)(f_0)(f_1)x$.

Notations 11.2.1. We write $\Sigma^{\Updownarrow}$ the set $\{0,1,\star\}\times\{\mathtt{in},\mathtt{out}\}$. We also denote by $\Sigma^{\Updownarrow}_{\mathtt{a},\mathtt{r}}$ the set $\Sigma^{\Updownarrow}\cup\{\mathtt{a},\mathtt{r}\}$, where $\mathtt{a}$ (resp. $\mathtt{r}$) stands for `accept` (resp. `reject`).

Initial segments of the natural numbers $\{0,1,\dots,n\}$ are denoted $[n]$. Up to renaming, all statesets can be considered to be of this form. We also denote $\dot{+}$ (resp. $\dot{-}$) the sum (resp. substraction) modulo $n+1$.

Notations 11.2.2. We fix once and for all an injection Ψ from the set $\Sigma^{\Updownarrow}_{\mathtt{a},\mathtt{r}}$ to intervals in $\mathbf{R}$ of the form $[k,k+1]$ with $k\in\mathbf{Z}$. For all $v\in\Sigma^{\Updownarrow}_{\mathtt{a},\mathtt{r}}$, we write $\langle v\rangle^Z_Y$ the measurable subset $\Psi(v)\times Y\times Z$ of $\mathbf{X}$, where $Y\subset[0,1]^{\mathbf{N}}$ and $Z\subset\{\star,0,1\}^{\mathbf{N}}$.

When $Y=[0,1]^{\mathbf{N}}$ (resp. $Z=\{\star,0,1\}^{\mathbf{N}}$), we omit the subscript (resp. superscript). The notation extends to subsets $S\subset\Sigma^{\Updownarrow}_{\mathtt{a},\mathtt{r}}$ by $\langle S\rangle=\cup_{v\in S}\langle v\rangle$ (a disjoint union).

Definition 11.2.1 *Given a word $\mathfrak{w}=a_1a_2\dots a_k$, we denote $W^{(2)}_{\mathfrak{w}}$ the graph with set of vertices $V^{W^{(2)}_{\mathfrak{w}}}\times S^{W^{(2)}_{\mathfrak{w}}}=\Sigma^{\Updownarrow}\times[k]$, set of edges $E^{W^{(2)}_{\mathfrak{w}}}=\{r,l\}\times[k]$, and source and target maps $s^{W^{(2)}_{\mathfrak{w}}}$ and $t^{W^{(2)}_{\mathfrak{w}}}$ defined as follows:*

$$\begin{array}{rcl} s^{W^{(2)}_{\mathfrak{w}}} & = & (r,i)\mapsto(a_i,\mathtt{out},i)\\ & & (l,i)\mapsto(a_i,\mathtt{in},i)\\ t^{W^{(2)}_{\mathfrak{w}}} & = & (r,i)\mapsto(a_{i\dot{+}1},\mathtt{in},i\dot{+}1)\\ & & (l,i)\mapsto(a_{i\dot{-}1},\mathtt{out},i\dot{-}1)\end{array}$$

Notations 11.2.3. We write $s^{W^{(2)}_{\mathfrak{w}}}_{\Sigma^{\Updownarrow}}$ (resp. $t^{W^{(2)}_{\mathfrak{w}}}_{\Sigma^{\Updownarrow}}$) the projection of the source (resp. target) map onto $\Sigma^{\Updownarrow}$, and $s^{W^{(2)}_{\mathfrak{w}}}_{[k]}$ (resp. $t^{W^{(2)}_{\mathfrak{w}}}_{[k]}$) the projection of the source (resp. target) map onto $[k]$.

The graph thus defined is the discrete representations of $\mathfrak{w}$, that one can relate [39] to the representation shown in Figure 11.2. Now, a word graphing is a *geometric representation* of a graph representation of a word: it can be defined as a graphing representative with the same graph structure as a word representation.

[39]: Aubert et al. (2016), *Characterizing co-NL by a group action*

Definition 11.2.2 *Let $\mathfrak{w}$ be a word $\mathfrak{w}=a_1a_2\dots a_k$ over the alphabet Σ. The* canonical *graphing representation $\bar{W}^{(2)}_{\mathfrak{w}}$ of $\mathfrak{w}$ is the graphing:*

$$\{(\langle s^{W^{(2)}_{\mathfrak{w}}}_{\Sigma^{\Updownarrow}}(e)\rangle,\phi_e,1,s^{W^{(2)}_{\mathfrak{w}}}_{[k]}(e)\to t^{W^{(2)}_{\mathfrak{w}}}_{[k]}(e))\mid e\in E^{\bar{W}^{(2)}_{\mathfrak{w}}}\},$$

where $\phi_e : \langle s^{W_{\mathfrak{w}}^{(2)}}_{\Sigma\mathbb{1}} \rangle \to \langle t^{W_{\mathfrak{w}}^{(2)}}_{\Sigma\mathbb{1}} \rangle$ *is a translation. A* word graphing W *of stateset* S^W *is a graphing obtained from* $\bar{W}^{(2)}_{\mathfrak{w}}$ *by renaming the stateset w.r.t. an injection* $S^W \mapsto [k]$.

We write $\mathbf{Rep}^{(2)}(\mathfrak{w})$ *the set of word graphings for* $\mathfrak{w}$.

Definition 11.2.3 *Given a word* $\mathfrak{w}$*, a* representation of $\mathfrak{w}$ *is a graphing* $!L$ *where* L *belongs to* $\mathbf{Rep}^{(2)}(\mathfrak{w})$. *The set of representations of words in* Σ *is denoted* $\mathbf{W}^{(2)}_{\Sigma}$*, the set of representations of a specific word* $\mathfrak{w}$ *is denoted* $\mathbf{Rep}^{(\cdot)}(\mathfrak{w})$.

We define the conduct $!\mathbf{Words}^{(2)}_{\Sigma} = (\mathbf{W}^{(2)}_{\Sigma})^{\perp\!\!\perp}$.

A explained in the introduction, we will be interested in machines of type $!\mathbf{Words}^{(2)}_{\Sigma} \multimap \mathbf{NBool}$, where **NBool** should be understood as a non-deterministic version of **Bool**.

Definition 11.2.4 *We define the (unproper) behaviour* **NBool** *as* $\mathbf{T}_{\langle \mathsf{a}, \mathsf{r} \rangle}$*, where for all measurable sets* V *the behaviour* $\mathbf{T}_V$ *is defined as the set of all projects of support* V.

Computations, Tests and Languages

Definition 11.2.5 *An* $\mathfrak{m}$*-graphing* G *is* finite *when it has a representative* H *whose set of edges* E^H *is finite.*

Definition 11.2.6 *For all monoid action* $\mathfrak{m}$*, we define* $\mathbf{Pred}(\mathfrak{m})$ *as the set of* $\mathfrak{m}$*-graphings in* $!\mathbf{Words}^{(2)}_{\Sigma} \multimap \mathbf{NBool}$.

A predicat $\mathfrak{m}$-machine *over the alphabet* Σ *is a finite* $\mathfrak{m}$*-graphing belonging to* $\mathbf{Pred}(\mathfrak{m})$.

The computation of a given machine on a given input is represented by the *execution*, i.e. the computation of paths defined in section 9.2. The result of the execution is an element of **NBool**, i.e. a sort of generalised boolean value[7].

7: For the specific case of 'deterministic machines', the result in fact belongs to the subtype **Bool** of booleans.

Definition 11.2.7 (Computation) *Let* M *be a* $\mathfrak{m}$*-machine,* $\mathfrak{w}$ *a word over the alphabet* Σ *and* $!L \in !\mathbf{Words}^{(2)}_{\Sigma}$. *The* computation *of* M *over* $!L$ *is defined as the graphing* $M :: !L \in \mathbf{NBool}$.

The principle of the approach is to use the orthogonality (which defines types) to capture the notion of acceptance. This is done using *tests*.

Definition 11.2.8 *A* test *is a family of projects of support* $\langle \mathsf{a}, \mathsf{r} \rangle$.

We now define the language characterised by a machine. For this, one could consider *existential* $\mathcal{L}^{\mathcal{T}}_{\exists}(M)$ and *universal* $\mathcal{L}^{\mathcal{T}}_{\forall}(M)$ languages for a machine M w.r.t. a test $\mathcal{T}$:

$$\mathcal{L}^{\mathcal{T}}_{\exists}(M) = \{\mathfrak{w} \in \Sigma^* \mid \forall \mathfrak{t}_i \in \mathcal{T}, \exists \mathfrak{w} \in \mathbf{Rep}^{(\cdot)}(\mathfrak{w}), M :: \mathfrak{w} \perp \mathfrak{t}_i\}$$
$$\mathcal{L}^{\mathcal{T}}_{\forall}(M) = \{\mathfrak{w} \in \Sigma^* \mid \forall \mathfrak{t}_i \in \mathcal{T}, \forall \mathfrak{w} \in \mathbf{Rep}^{(\cdot)}(\mathfrak{w}), M :: \mathfrak{w} \perp \mathfrak{t}_i\}$$

The best situation is in fact when both definitions coincide, as it ensures that only one representation of $\mathfrak{w}$ need to be considered to check whether

$\mathfrak{w}$ belongs to the language or not. This situation is captured by the notion of *uniform test*.

Definition 11.2.9 (Uniformity) *Let $\mathfrak{m}$ be a monoid action. The test $\mathcal{T}$ is said* uniform *w.r.t. $\mathfrak{m}$-machines if for all such machine M, and any two elements $\mathfrak{w}, \mathfrak{w}'$ in $\mathbf{Rep}^{(\cdot)}(\mathfrak{w})$:*

$$M :: \mathfrak{w} \in \mathcal{T}^{\perp} \text{ if and only if } M :: \mathfrak{w}' \in \mathcal{T}^{\perp}$$

We write in this case $\mathcal{L}^{\mathcal{T}}(M) = \mathcal{L}^{\mathcal{T}}_{\exists}(M) = \mathcal{L}^{\mathcal{T}}_{\forall}(M)$.

We now have introduced all the needed ingredients to state and prove characterisations of complexity classes. We will first recall previously known results [45].

[45]: Seiller (2018), *Interaction Graphs: Nondeterministic Automata*

Notations 11.2.4. For $U \subset \mathbf{X}$, we define Id_U as the graphing with a single edge and stateset $[0]$: $\{(\langle \mathsf{r} \rangle, x \mapsto x, 1 \cdot \mathbf{1}, 0 \to 0)\}$.

Proposition 11.2.5 *The test $\mathcal{T}_{-}$, defined as*

$$\{\mathrm{t}^{-}_{\zeta} = (\zeta, \mathrm{Id}_{\langle r \rangle}) \mid \zeta \neq 0\},$$

is uniform w.r.t. $\mathfrak{n}_{\infty}$-machines.

Note that since it is uniform w.r.t. $\mathfrak{n}_{\infty}$-machines, it is uniform w.r.t. $\mathfrak{m}$-machines for any sub-monoid action $\mathfrak{m}$ – hence w.r.t. all monoid actions considered in this paper. We will now define classes defined from *non-deterministic* $\mathfrak{m}$-machines, i.e. finite graphing representatives of type $\mathbf{Pred}(\mathfrak{m})$ in the model $\mathbb{M}[\{0,1\}, \mathfrak{m}]$ (i.e. weights are either 0 or 1).

Definition 11.2.10 *We define the complexity class*

$$\mathbf{Pred}^{\mathrm{co}}(\mathfrak{m}) = \{\mathcal{L}^{\mathcal{T}_{-}}(M) \mid M \ \mathfrak{m}\text{-machine in } \mathbb{M}[\{0,1\}, \mathfrak{m}]\}.$$

The starting point of this work was the realisation that one can define another test $\mathcal{T}_{+}$ capturing the notion of acceptance in NLogspace. Based on this idea, and using technical lemmas from the previous paper, we can characterise easily the hierarchy of complexity classes defined by k-head non-deterministic automata with the standard non-deterministic acceptance condition (i.e. there is at least one accepting run). We state the results and provide explanations, but we do not provide a formal proof. Indeed, while an adaptation of the techniques used in previous work [45] could be used, the result follows from the more general method presented in the next sections.

[45]: Seiller (2018), *Interaction Graphs: Nondeterministic Automata*

Notations 11.2.6. For $U \subset \mathbf{X}$, we define $\mathrm{Id}^{1/2}_U$ as the graphing with a single edge and stateset $[0]$: $\{(\langle \mathsf{r} \rangle, x \mapsto x, \frac{1}{2} \cdot \mathbf{1}, 0 \to 0)\}$.

Proposition 11.2.7 *The test $\mathcal{T}_{+}$ defined as the family*

$$\{(0, \mathrm{Id}^{1/2}_{A_n}) \mid A_n = \langle a \rangle_{[0,\frac{1}{n}]^n \times [0,1]^{\mathbf{N}}}, n \in \mathbf{N}\}$$

is uniform w.r.t. $\mathfrak{n}_{\infty}$-machines.

Definition 11.2.11 *We define the complexity class* $\mathbf{Pred}^{\mathtt{ndet}}(\mathfrak{m})$ *as the set*

$$\{\mathcal{L}^{\mathcal{T}_+}(M) \mid M \ \mathfrak{m}\text{-machine in } \mathbb{M}^{\mathtt{ndet}}[\{0,1\}, m]\}.$$

The considered test does indeed capture the usual condition for acceptance of non-deterministic machines. In fact, the sole element $(0, \mathrm{Id}^{1/2}_{\langle \mathtt{a} \rangle})$ is enough to obtain completeness, by a result from our earlier work [45, Proposition 46]. We do not state it here, as it is generalised below. From these results, a k-heads two-way automaton $\mathtt{M}$ accepts a word $\mathfrak{w}$ if and only if there exist at least one alternating path between the graphing translation $\{\mathtt{M}\}$ of M and the word representation $!\bar{W}^{(2)}_{\mathfrak{w}}$ whose source and target is $\langle \mathtt{a} \rangle_Y$ for some subspace Y. Thus, $\mathtt{M}$ accepts $\mathfrak{w}$ if and only if there are alternating cycles between $\{\mathtt{M}\} :: !\bar{W}^{(2)}_{\mathfrak{w}}$ and $\mathrm{Id}^{1/2}_{\langle \mathtt{a} \rangle}$, i.e. if and only if $[\![\{\mathtt{M}\} :: !\bar{W}^{(2)}_{\mathfrak{w}}, \mathrm{Id}^{1/2}_{\langle \mathtt{a} \rangle}]\!] \neq 0, \infty$, or equivalently if and only if $\{\mathtt{M}\} :: !\bar{W}^{(2)}_{\mathfrak{w}} \perp \mathrm{Id}^{1/2}_{\langle \mathtt{a} \rangle}$.

However, the whole family of tests is required to obtain soundness. Indeed, in the general case, it might be possible that a $\mathfrak{m}_i$-machine G passes the test $\{(0, \mathrm{Id}^{1/2}_{\langle \mathtt{a} \rangle})\}$ by taking several (possibly different) paths through the execution $G :: !\bar{W}^{(2)}_{\mathfrak{w}}$, creating a cycle of arbitrary length between $G :: !\bar{W}^{(2)}_{\mathfrak{w}}$ and $\{(0, \mathrm{Id}^{1/2}_{\langle \mathtt{a} \rangle})\}$. In that case, it is not clear that the existence of such a cycle can be decided with some automaton M. However, if $G :: !\bar{W}^{(2)}_{\mathfrak{w}}$ passes all tests in $\mathcal{T}_+$, it imposes the existence of a cycle of length 2 between $G :: !\bar{W}^{(2)}_{\mathfrak{w}}$ and $\mathrm{Id}^{1/2}_{\langle \mathtt{a} \rangle}$, something that can be decided by an automaton. The existence of the length 2 cycle is enforced by the restriction of the test to subspaces of the form $[0, \frac{1}{n}]$; we refer the the proof [46] for more details.

[46]: Seiller (2023), *Implicit complexity through linear realisability: polynomial time and probabilistic classes*

11.3. Statement of the results

Multihead automata with pushdown stacks

The proof of the characterisation theorem [45] relies on a representation of multihead automata as graphings. We here generalise the result to probabilistic automata with a pushdown stack. For practical purposes, we consider a variant of the classical notion of probabilistic two-way multihead finite automata with a pushdown stack obtained by:

[45]: Seiller (2018), *Interaction Graphs: Nondeterministic Automata*

- fixing the right and left end-markers as both being equal to the fixed symbol $\star$;
- fixing once and for all unique initial, accept and reject states;
- choosing that each transition step moves exactly one of the multiple heads of the automaton;
- imposing that all heads are repositioned on the left end-marker and the stack is emptied before accepting/rejecting.
- symbols from the stack are read by performing a `pop` instruction; if the end-of-stack symbol $\star$ is popped, it is pushed on the stack in the next transition.

It should be clear that these choices in design have no effect on the sets of languages recognised.

Definition 11.3.1 *A k-heads non-deterministic two-way multihead finite automata with a pushdown stack (2NFA+s(k)) $\mathtt{M}$ is defined as a tuple $(\Sigma, Q, \to)$, where the transition $\to$ is a relation that associates to each element of $\Sigma_\star^k \times Q$ a subset of $(Inst \times Q)$ where Inst is the set of instructions: $(\{1, \ldots, k\} \times \{in, out\}) \times \{\mathrm{Id}, pop, push_1, push_0, push_\star\}$.*

We say $\mathtt{M}$ is deterministic *if $\to$ is a function.*

Definition 11.3.2 *A k-heads probabilistic two-way multihead finite automata with a pushdown stack (2PFA+s(k)) $\mathtt{M}$ is defined as a tuple $(\Sigma, Q, \to)$, where the transition function $\to$ is a map that associates to each element of $\Sigma_\star^k \times Q$ a sub-probability distribution over the set $(Inst \times Q)$.*

Notations 11.3.1. The set of deterministic (resp. non-deterministic, resp. probabilistic) two-way multihead automata with k heads and *without pushdown stack* (i.e. not using stack instructions) is written **2dfa**(k) (resp. **2nfa**(k), resp. **2pdfa**(k)) and the corresponding complexity class is noted 2DFA(k) (resp. 2PFA(k)). The set of all deterministic two-way multihead automata $\cup_{k \geqslant 1}$**2dfa**(k) is denoted by **2dfa**. We define in a similar manner the sets **2nfa** and **2pfa**. The corresponding complexity classes 2DFA(∞), 2NFA(∞), and 2PFA(∞) are known to be equal to Logspace, NLogspace, and PLogspace [162].

[162]: Holzer et al. (2011), *Complexity of multi-head finite automata: Origins and directions*

Notations 11.3.2. The set of k heads deterministic (resp. non-deterministic, resp. probabilistic) two-way multihead automata with k heads and a pushdown stack is written **2dfa** + **s**(k) (resp. **2nfa** + **s**(k) , resp. **2pdfa** + **s**(k)) and the corresponding complexity class is noted 2DFA+s(k) (resp. 2NFA+s(k), resp. 2PFA+s(k)). The set of all deterministic two-way multihead automata with a pushdown stack $\cup_{k \geqslant 1}$**2dfa** + **s**(k) is denoted by **2dfa** + **s**. We define in a similar way the sets **2nfa** + **s** and **2pfa** + **s**. The corresponding complexity classes 2DFA+s(∞), 2NFA+s(∞), and 2PFA+s(∞) are known to be equal to Ptime [163], Ptime, and PPtime respectively.

[163]: Macarie (1997), *Multihead Two-Way Probabilistic Finite Automata*

Remark 11.3.1 We note that non-deterministic two-way multihead automata with a pushdown stack characterise Ptime and not NPtime, as shown by Cook [164] using memoization.

[164]: Cook (1971), *Characterizations of Pushdown Machines in Terms of Time-Bounded Computers*

Results

Before stating the theorems, we need to define the complexity classes considered in the realisability models. We already introduced two notions of tests; we will require a last one adapted to probabilistic models of computation.

Proposition 11.3.3 *Let $\eta > 0$. The test $\mathcal{T}^p[\epsilon]$ defined by*

$$(\log(1 - \frac{1}{2}.u), \mathrm{Id}^{1/2}\langle a \rangle^{V(\star^n)}_{[0,\frac{1}{n}]^n \times [0,1]^{\mathbf{N}}}]) \mid u \in [0, \epsilon], n \in \mathbf{N}\}$$

is uniform w.r.t. $\mathfrak{n}_\infty$-machines.

Definition 11.3.3 *We define the complexity class* $\mathbf{Pred}^{\mathrm{prob}}(\mathfrak{m})$ *as the set*

$$\{\mathcal{L}^{\mathcal{T}^p[\frac{1}{2}]}(M) \mid M \ \mathfrak{m}\text{-machine in } \mathbb{M}^{\mathrm{prob}}[[0,1], m]\}.$$

In the remaining sections, we will establish the following theorem.

Theorem 11.3.4 *For all $i \in \mathbf{N}^* \cup \{\infty\}$,*

$$\begin{array}{ll}
\mathbf{Pred}^{\mathtt{det}}(\mathfrak{m}_i) = 2D_{FA}(i), & \mathbf{Pred}^{\mathtt{det}}(\mathfrak{n}_i) = 2D_{FA+S}(i) \\
\mathbf{Pred}^{\mathtt{ndet}}(\mathfrak{m}_i) = 2N_{FA}(i), & \mathbf{Pred}^{\mathtt{ndet}}(\mathfrak{n}_i) = 2N_{FA+S}(i) \\
\mathbf{Pred}^{\mathtt{co}}(\mathfrak{m}_i) = co2N_{FA}(i), & \mathbf{Pred}^{\mathtt{co}}(\mathfrak{n}_i) = co2N_{FA+S}(i) \\
\mathbf{Pred}^{\mathtt{prob}}(\mathfrak{m}_i) = 2P_{FA}(i), & \mathbf{Pred}^{\mathtt{prob}}(\mathfrak{n}_i) = 2P_{FA+S}(i)
\end{array}$$

Corollary 11.3.5 *As special cases of the previous theorem,*

$$\begin{array}{ll}
\mathbf{Pred}^{\mathtt{det}}(\mathfrak{m}_\infty) = \textsc{Logspace}, & \mathbf{Pred}^{\mathtt{det}}(\mathfrak{n}_\infty) = \textsc{Ptime} \\
\mathbf{Pred}^{\mathtt{ndet}}(\mathfrak{m}_\infty) = \textsc{NLogspace}, & \mathbf{Pred}^{\mathtt{ndet}}(\mathfrak{n}_\infty) = \textsc{Ptime} \\
\mathbf{Pred}^{\mathtt{co}}(\mathfrak{m}_\infty) = \textsc{coNLogspace}, & \mathbf{Pred}^{\mathtt{co}}(\mathfrak{n}_\infty) = \textsc{Ptime} \\
\mathbf{Pred}^{\mathtt{prob}}(\mathfrak{m}_\infty) = \textsc{PLogspace}, & \mathbf{Pred}^{\mathtt{prob}}(\mathfrak{n}_\infty) = \textsc{PPtime}
\end{array}$$

The proofs are quite similar, even though we will need to state variants of the key lemmas depending on the case considered: deterministic, non-deterministic (with different notions of acceptance), and probabilistic. However, the principle is the same and both directions rely on the fact that computation is represented by paths (cf. section 9.2) and orthogonality is based on a measurement of cycles (cf. section 9.3).

To prove completeness, we then show how any automata can be simulated by a graphing. This requires the definition of the graphing translating the automaton, and proving that the orthogonality translates the existence of accepting runs in a quantitative manner (i.e. in the case of probabilistic machines, the sum of weights of the accepting paths will be equal to the probability of accepting).

The second part of the proof, soundness, is the most involved. It requires to show that given any $\mathfrak{n}_\infty$-machine M, the computation of M on the graphing representation of a word $\mathfrak{w}$ boils down to the computation of alternating paths between finite graphs (namely a graph mimicking M and the graph representation of $\mathfrak{W}$).

Part III.

Two roads that lie ahead

12. Invariants for lower bounds

This axis is a refined version of a longstanding research project, enriched with time by new results. The main question it aims to tackle is the following: **Can invariants for dynamical systems be used to prove separation results in computational complexity?**

A number of results explained above provide strong grounds for believing in this claim.

- The hierarchies of complexity classes shown in chapter 11 are strict (this is shown in a series of five different papers), and one proof technique is based on the incompressibility method based on Kolmogorov complexity. Moreover ,there are known relationships between Kolmogorov complexity and topological entropy.
- The result presented in Section 2.3 relies on the topological entropy of the dynamical system representing the machine. Topological entropy is an invariant for group/monoid actions which is finer than another notion: *orbit equivalence*.
- The equivalences studied on abstract models of computations (as part of my habilitation) highlights the relevance of purely mathematical equivalences between group/monoid actions, and in particular the notion of *orbit equivalence*.
- The characterisations of complexity classes in Section 2.2 are based on linear realisability techniques, and thus involve in an essential manner the orthogonality based on *zeta functions* (as explained in Section 2.1). Zeta functions are defined from the set of finite orbits in a dynamical system, which makes them an invariant w.r.t. orbit equivalence.

Putting everything together. More precisely, I showed in Theorem 11.1.27 that the hyperfinite geometry of interaction interprets fragments of linear logic of different expressivity depending on the type of maximal abelian von Neumann subalgebra $\mathfrak{A}$ chosen. It should be clear that picking a semi-regular masa in $\mathfrak{R}_{0,1}$ is equivalent to choosing a Cartan subalgebra of another von Neumann algebra $\mathfrak{N}$, namely $\mathfrak{N} = \mathcal{N}_{\mathfrak{R}_{0,1}}(A)$. I.e. the theorem states that the expressivity of the logic interpreted in linear realisability models depends on the pair $\mathfrak{A} \subset \mathfrak{N}$.

Based on the understanding of graphings in a chosen AMC $\alpha : \mathbb{M}(I) \curvearrowright \mathbf{X}$ as a generalisation of the group measure space construction of Murray and von Neumann (chapter 9), this choice of a pair $\mathfrak{A} \subset \mathfrak{N}$ corresponds – in the case of measure-preserving group actions – to the inclusion $L^\infty(\mathbf{X}) \subset \mathcal{L}(L^2(\mathbf{X}, \mathbb{M}(I)))$.

This leads to understanding Theorem 11.3.4 as an extension and refinement of Theorem 11.1.27, exploiting the more tractable setting of graphings (as opposed to operators). This leads to the idea that invariants for the pair $\mathfrak{A} \subset \mathfrak{N}$, or equivalently[1], could lead to separation results. Note that one may also consider whether the corresponding Borel equivalence relations (the preorder defined in Equation 4.1, which becomes a Borel

1: Note that a theorem by Singer [165] shows that α is orbit equivalent to β if and only if the induced pairs $\mathfrak{A} \subset \mathfrak{N}$ and $\mathfrak{A}' \subset \mathfrak{N}'$are isomorphic.

equivalence relation when α is a measure-preserving group action) are isomorphic in some sense.

The first step in that direction was presented in chapter 8. Indeed, we show that invariants (namely topological entropy) of the graphings can be exploited to provide proofs of complexity lower bounds. While this result does not exploit the invariants for the monoid action, I hope that the technique can be lifted to the realisability models, and provide lower bounds techniques for the uniform classes characterised in Theorem 11.3.4.

Perspectives. A conjectured result would be that two abstract models of computation that capture the same complexity class necessarily are orbit equivalent, or have isomorphic induced pre-orders (Equation 4.1). Such a result would allow the use of known invariants for orbit equivalence or Borel equivalence relations (such as cost or ℓ^2-Betti numbers [85] or cost[21]) to prove separation results. A possible (involved) proof mechanism for such a result would be to start from two non-orbit-equivalent actions α and β, and deduce from this the existence of a β-graphing that computes a language which cannot be computed by an α-graphing. The techniques for proving this would be extremely involved, and require a very good understanding of the zeta functions involved, but not impossible to achieve.

[85]: Gaboriau (2002), *Invariants ℓ^2 de relations d'équivalence et de groupes*

[21]: Gaboriau (2000), *Coût des relations d'équivalence et des groupes*

However the general objective is both very ambitious and extremely hard. In order to approach the question and make partial progress, I identify three specific lines of research.

1. Providing more evidence that other standard separation and lower bounds results from the literature can be expressed in terms of invariants for dynamical systems. In particular, some lower bounds based on Kolmogorov complexity [166, 167] could surely be shown to be specific applications of the invariant methods, exploiting the connection between entropy and Kolmogorov complexity [168]. More generally, I will try to reformulate standard lower bound techniques on boolean circuits. At this point, I have identified two different methods (or rather groups of methods) for proving lower bounds which we I believe can be related to the invariant method: the *Fusion Method*, a term coined by A. Widgerson [169] to describe both Sipser's topological approach [170, 171] and Razborov's *approximation method* [172], and the polynomial method [173, 174].
2. Generalising our abstract invariant method to tackle more involved parts of the Geometric complexity Theory (GCT) program of Mulmuley. Indeed, in Section 2.3 I explained how we obtained a strengthening of a result by Mulmuley [125]. This result is considered as the first instance of the GCT approach. Building on this connection, we expect to reformulate and bring new dynamic insights within the GCT program.
3. Exploring another important and connecting direction in computational complexity, namely lower bounds proofs based on improving algorithms, such as the result of Kabanets and Impagliazzo relating feasible algorithms for polynomial identity testing (PIT) and lower bound results [175], or Williams's recent proof that NEXPTIME (Nondeterministic Exponential Time) does not have quasi-polynomial size ACC circuits [176]. As argued by Williams during an invited

[166]: Chrobak et al. (1986), *k+1 heads are better than k for PDA's*

[167]: Li et al. (2008), *An introduction to Kolmogorov complexity and its applications*

[168]: Galatolo et al. (2010), *Effective symbolic dynamics, random points, statistical behavior, complexity and entropy*

[169]: Wigderson (1993), *The fusion method for lower bounds in circuit complexity*

[170]: Sipser (1983), *Borel Sets and Circuit Complexity*

[171]: Sipser (1984), *A Topological View of Some Problems in Complexity Theory*

[172]: Razborov (1989), *On the Method of Approximations*

[173]: Beigel (1993), *The Polynomial Method in Circuit Complexity*

[174]: Williams (2014), *The Polynomial Method in Circuit Complexity Applied to Algorithm Design (Invited Talk)*

[125]: Mulmuley (1999), *Lower Bounds in a Parallel Model without Bit Operations*

[175]: Kabanets et al. (2003), *Derandomizing Polynomial Identity Tests Means Proving Circuit Lower Bounds*

[176]: Williams (2014), *Nonuniform ACC Circuit Lower Bounds*

talk at the Computer Science Logic (CSL) 2015 conference, some logical understanding of the mechanisms behind his proof would provide important insights on the method. On top of these results, recent work on (a weak version of) polynomial identity testing by A. Wigderson *et al.* [177] exhibits connections between this approach and Mulmuley's GCT program and I hope that the dynamic point of view exposed here will shed light on this connection, building in particular on the previous item.

[177]: Allen-Zhu et al. (2018), *Operator Scaling via Geodesically Convex Optimization, Invariant Theory and Polynomial Identity Testing*

13. Architecture-oriented complexity

This last axis is very new. I believe it to be of particular interest, which is why I include it in this document.

Among the examples of models of computation that can be represented by monoid actions, there are most (if not all) the abstract models encountered in the literature (Turing machines, lambda-calculus, BSS models, PRAMs, etc.). However, I realised that more realistic models could also be represented faithfully. In fact one can show that *instruction set architectures* give rise to abstract models of computation. Those can be enriched with a detailed cache structure from which can be defined a notion of cost model (i.e. complexity measure) taking into account cache memory access times. This lead me to the realisation that complexity as usually considered (based on Turing machines) does not account for specific and important aspects of real machines, among which at least:

- the significant gain in accessing cache memory involves a notion of locality (using a given value in constrained parts of the code) that should be accounted for in theoretical complexity results;
- the speculation mechanisms in current processors also impacts the complexity of the considered algorithm.

I would therefore like to develop a machine-level complexity theory that could aim at obtaining theoretical results on architecture-specific complexity. While some results along those lines can be found in the literature, they consist in a few examples of specific algorithms that run faster than standard ones on specific architectures.

- A number of results on matrix multiplication algorithms have been obtained, but these results are mainly experimental (e.g. [178] which may be one of the most theoretical). In particular, the specific choices of parameters (e.g. the size of the blocks used in dividing the matrices) giving optimal results in specific architectures are not explained in terms of parameters of the latter (e.g. cache size, number of registers, etc.).
- In some cases [179], speculation can be shown to make theoretically optimal algorithms slower than more naive ones. Intuitively, this is because the speculation in the optimal algorithm fails sufficiently to cancel the theoretical gain in time complexity w.r.t. a theoretically less efficient algorithm with a more 'regular' structure (i.e. for which the speculation guesses right most of the time).

[178]: Goto et al. (2008), *Anatomy of High-Performance Matrix Multiplication*

[179]: Auger et al. (2016), *Good predictions are worth a few comparisons*

I expect to be able to obtain theoretical results based on chosen parameters of the architecture. These results could be of interest to provide explanations for some behaviours found experimentally, provide proven optimal algorithms for specific architectures, and maybe even lead to guide architecture design choices.

Obviously, my position as non-expert of computer architecture led me to question at first the feasibility and interest of developing such an architecture-oriented complexity. I therefore started interacting with researchers in computer architecture in order to understand if the approach makes sense and could be of interest for more applied communities

(which is an essential motivation for investigating these questions further; another theoretical framework that could not impact actual implementations would not motivate me). The feedback has been quite positive for the moment (parts of the contents of this section take their origin in our discussions). A first proof of concept result is required to show the soundness of the method. In the meantime, I have started to investigate architecture-oriented complexity of simple problems (sorting, matrix multiplication) in order to understand if it can lead to interesting results, and started to investigate potential connection with techniques from WCET (worst-case execution time) analyses [180].

[180]: Wilhelm et al. (2008), *The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools*

This objective of trying to understand machine-level complexity goes hand-in-hand with the static analyses work described in Section 2.5. Our aim has always been to analyse the code as far as possible down the compilation pipeline. We currently have a candidate implementation of the mwp-analysis (in fact a parametrised analysis defined as a functor) on CompCert intermediate representation. My hope is to push these analyses even further down to machine code, in order to have more realistic results. From the point of view of compilers, another interesting research research direction could arise from the confrontation of back-end optimisations of compilers (i.e. from intermediate representation to assembly code) with our theoretical architecture-based approach.

Part IV.

Appendix

14. von Neumann algebras

First Definitions and Results

The theory of von Neumann algebras, under the name of 'rings of operators', was first developed by Murray and von Neumann in a series of seminal papers [181–188].

The double commutant theorem.

A normed $*$-algebra is a normed algebra endowed with an antilinear isometric involution $(\cdot)^*$ which reverses the product:

$$(t^*)^* = t \quad \|t^*\| = \|t\| \quad (t+u)^* = t^* + u^* \quad (\lambda t)^* = \bar{\lambda} t^* \quad (tu)^* = u^* t^*$$

A normed $*$-algebra is a C*-algebra when it is complete (i.e. it is a Banach algebra) and satisfies the *C*-identity* $\|t^*t\| = \|t\|^2$.

We denote by $\mathcal{L}(\mathbb{H})$ the $*$-algebra of continuous (or equivalently, bounded) linear maps from the Hilbert space $\mathbb{H}$ to itself. This algebra can be endowed with the following three topologies:

- The norm topology, for which a net (T_λ) converges toward 0 when the net $\|T_\lambda\|$ converges to 0 in $\mathbf{C}$;
- The strong operator topology (SOT) which is the topology of pointwise convergence when $\mathbb{H}$ is considered endowed with its norm topology: a net (T_λ) converges toward 0 when for all $\zeta \in \mathbb{H}$, the net $(\|T_\lambda \zeta\|)$ converges towards 0 in $\mathbf{C}$;
- The weak operator topology (WOT) which is the topology of pointwise convergence when $\mathbb{H}$ is considered endowed with its weak topology: a net (T_λ) converges toward 0 when for all $\zeta, \eta \in \mathbb{H}$, the net $(\langle T_\lambda \zeta, \eta \rangle)$ converges towards 0 in $\mathbf{C}$;

Definition 14.0.1 (von Neumann algebra) *A von Neumann algebra is a $*$-sub-algebra of $\mathcal{L}(\mathbb{H})$ which is closed for the strong operator topology (SOT).*

We now explain Murray and von Neumann's fundamental 'double commutant theorem'. Pick $M \subset \mathcal{L}(\mathbb{H})$. We define the commutant of M (in $\mathcal{L}(\mathbb{H})$) as the set $M'^{\mathcal{L}(\mathbb{H})} = \{x \in \mathcal{L}(\mathbb{H}) \mid \forall m \in M, mx = xm\}$. We will in general omit to precise the ambient algebra and denote abusively M' the commutant of M if the context is sufficiently clear. We will denote by M'' the bi-commutant $(M')'$ of M.

The following theorem is the keystone of the von Neumann algebras theory. It is particularly elegant, since it shows an equivalence between a purely algebraic notion – being equal to its bi-commutant – and a purely topological notion – being closed for the strong operator topology. This theorem is due to von Neumann [181].

Theorem 14.0.1 (Double Commutant Theorem) *Let M be a $*$-sub-algebra of $\mathcal{L}(\mathbb{H})$ such that $1_{\mathcal{L}(\mathbb{H})} \in M$. Then M is a von Neumann algebra if and only if $M = M''$.*

Remark 14.0.1 Since the strong operator topology (SOT) is weaker than the norm topology, a von Neumann algebra $\mathfrak{M}$ is also closed for the norm topology, and is also a C*-algebra. Moreover, since $\mathfrak{M}$, as a von Neumann algebra, is the commutant of a set of operators, it necessarily contains the identity operator in $\mathcal{L}(\mathbb{H})$, and consequently is a unital C*-algebra. One can therefore define the continuous spectral calculus for operators in $\mathfrak{M}$.

Direct Integrals.

Let $\mathfrak{M}$ be a von Neumann algebra. We define the *center* of $\mathfrak{M}$ as the von Neumann algebra $\mathfrak{Z}(\mathfrak{M}) = \mathfrak{M} \cap \mathfrak{M}'$.

Definition 14.0.2 (Factor) *A* factor *is a von Neumann algebra $\mathfrak{M}$ whose center is trivial, i.e. such that $\mathfrak{Z}(\mathfrak{M}) = \mathbf{C}.1_{\mathcal{L}(\mathbb{H})}$.*

The study of von Neumann algebras can be reduced to the study of factors. This is one of the most important results of the theory, which is due to von Neumann [188]: he showed that every von Neumann algebra can be written as a *direct integral* of factors. A direct integral is a direct sum over a continuous index set, in the same way an integral is a sum over a continuous index set. A complete exposition of this result can be found in the first book of the Takesaki series [189], Section IV.8, page 264.

Here are the main ideas. If $\mathfrak{A}$ is not a factor, its center $\mathfrak{Z}(\mathfrak{A})$ is a non-trivial commutative von Neumann algebra (i.e. different from $\mathbf{C}$). Suppose now that $\mathfrak{Z}(\mathfrak{A})$ is a *diagonal* algebra, i.e. that there exists a countable set I (which could be finite) and a family $(p_i)_{i\in I}$ of pairwise disjoint minimal projections such that $\sum_{i\in I} p_i = 1$. Then the algebras $p_i\mathfrak{A}p_i$ are factors, and one has $\mathfrak{A} = \bigoplus_{i\in I} p_i\mathfrak{A}p_i$. However, in the general case, the center $\mathfrak{Z}(\mathfrak{A})$ does not need to be a diagonal algebra, and it can contain a *diffuse* sub-algebra, i.e. a sub-algebra that does not have minimal projections. Then it is necessary to consider a continuous version of the direct sum: the direct integral.

Definition 14.0.3 *Let $(X, \mathcal{B}, \mu)$ be a measured space. A family $(\mathbb{H}_x)_{x\in X}$ of Hilbert spaces is* measurable *over $(X, \mathcal{B}, \mu)$ when there exists a countable partition $(X_i)_{i\in I}$ of X such that for all $i \in I$:*

$$\exists \mathbb{K},\ \forall x \in X_i, \mathbb{H}_x = \mathbb{K}$$

where $\mathbb{K}$ is either equal to $\mathbf{C}^n$ ($n \in \mathbf{N}$) or equal to $\ell^2(\mathbf{N})$.

A section $(\xi_x)_{x\in X}$ ($\xi_x \in \mathbb{H}_x$) is measurable when its restriction to each element X_n of the partition is measurable.

Definition 14.0.4 *Let $(\mathbb{H}_x)_{x\in X}$ be a measurable family of Hilbert spaces over a measured space $(X, \mathcal{B}, \lambda)$. The* direct integral $\int_X^{\oplus} \mathbb{H}_x d\lambda(x)$ *is the Hilbert space whose elements are equivalence classes of measurable sections modulo almost everywhere equality, and the scalar product is defined by:*

$$\langle (\xi_x)_{x\in X}, (\zeta_x)_{x\in X} \rangle = \int_X^{\oplus} \langle \xi_x, \zeta_x \rangle d\lambda(x)$$

In the same way commutative C*-algebras are exactly the algebras of continuous functions from locally compact Hausdorff spaces to **C** (this is Gelfand's theorem, [190]), one can show that every commutative von Neumann algebra can be identified with the algebra $L^\infty(X, \mathcal{B}, \lambda)$ of essentially bounded measurable functions on a measured space $(X, \mathcal{B}, \lambda)$.

Theorem 14.0.2 *Let $\mathfrak{A}$ be a commutative von Neumann algebra. There exists a measurable family of Hilbert spaces $(\mathbb{H}_x)_{x\in X}$ over a measured space $(X, \mathcal{B}, \lambda)$ such that $\mathfrak{A}$ is unitarily equivalent to the algebra $L^\infty(X)$ acting on the Hilbert space $\int_X^\oplus \mathbb{H}_x d\lambda(x)$.*

We will not define here neither the notion of measurable family of von Neumann algebras, nor the one of direct integrals of von Neumann algebras. We only state the fundamental theorem mentioned above. The result is due to von Neumann [188] and appears in Takesaki's second book [191] (Theorem IV.8.21 page 275).

Theorem 14.0.3 *Every von Neumann algebra can be written as a direct integral of factors.*

Classification of factors.

The study of factors led to a classification based on the study of the set of projections and their isomorphisms (partial isometries). We recall that a projection is an operator p such that $p = p^* = p^2$ (this is sometimes referred to as an 'orthogonal projection'). If $\mathfrak{M}$ is a von Neumann algebra, we will denote by $\Pi(\mathfrak{M})$ the set of projections in $\mathfrak{M}$. Since $\mathfrak{M}$ is a subalgebra of $\mathcal{L}(\mathbb{H})$ for a given Hilbert space $\mathbb{H}$, the projections in $\Pi(\mathfrak{M})$ are in particular projections in $\mathcal{L}(\mathbb{H})$. As such, they are in correspondence with subspaces of $\mathbb{H}$: the projection p corresponds to the closed subspace $p\mathbb{H}$. Two projections p, q are *disjoint* when $pq = 0$, translating the fact that the two corresponding closed subspaces $p\mathbb{H}$ and $q\mathbb{H}$ are disjoint. Moreover, the set $\Pi(\mathfrak{M})$ is endowed with a partial ordering inherited from the inclusion of subspaces: $p \leq q$ if and only if $pq = p$ if and only if $p\mathbb{H} \subset q\mathbb{H}$.

Now, the idea of Murray and von Neumann [182] was to consider an equivalence relation on the set of projections. This equivalence relation depends on the algebra $\mathfrak{M}$ and translates the fact that $\mathfrak{M}$ contains an isomorphism between the corresponding subspaces. Namely, they define the equivalence as follows: two projections p, q are *Murray von Neumann equivalent in* $\mathfrak{M}$, noted $p \sim_{\mathfrak{M}} q$, when there exists an element $u \in \mathfrak{M}$ such that $uu^* = p$ and $u^*u = q$. Notice that this implies that u is a partial isometry.

The partial ordering $\leq$ then induces a partial ordering $\precsim_{\mathfrak{M}}$ on the equivalence classes of projections in $\mathfrak{M}$, i.e. on the set $\Pi(\mathfrak{M})/\sim_{\mathfrak{M}}$.

Remark 14.0.2 As we explained above, $p \leq q$ means that $p\mathbb{H}$ is a closed subspace of $q\mathbb{H}$. The fact that $p \sim_{\mathfrak{M}} q$ translates the fact that $p\mathbb{H}$ and $q\mathbb{H}$ are *inner (w.r.t $\mathfrak{M}$) isomorphic*, i.e. there exists an isomorphism between them which is an element of $\mathfrak{M}$, or in other terms, the fact that they are isomorphic is witnessed by an element of $\mathfrak{M}$. Consequently, the fact that $p \precsim_{\mathfrak{M}} q$ translates the idea that $p\mathbb{H}$ is *inner isomorphic* to a closed subspace

of $q\mathbb{H}$, and therefore that $p\mathbb{H}$ is somehow *smaller* than $q\mathbb{H}$ in the sense that an element of $\mathfrak{M}$ witnesses the fact that it is smaller.

Definition 14.0.5 *A projection p in a von Neumann algebra $\mathfrak{M}$ is* infinite *(in $\mathfrak{M}$) when there exists $q \prec p$ (i.e. a proper sub-projection) such that $q \sim_{\mathfrak{M}} p$. A projection is* finite *(in $\mathfrak{M}$) when it is not infinite (in $\mathfrak{M}$).*

The following result combines Proposition V.1.3 page 291 and Theorem V.1.8 page 293 in Takesaki's book [189].

Proposition 14.0.4 *Let $\mathfrak{M}$ be a von Neumann algebra. Then $\mathfrak{M}$ is a factor if and only if the relation $\preceq_{\mathfrak{M}}$ is a total ordering.*

To state the following definition and theorem, we will use a slight variant of the usual notion of order type: we distinguish the element denoted by ∞ from any other element, considering that ∞ represents a class of infinite projections. For instance, $\{0, 1\}$ and $\{0, \infty\}$ should be considered as distinct since the first does not contain infinite elements contrarily to the second.

Definition 14.0.6 (Type of a Factor) *Let $\mathfrak{M}$ be a factor. We will say that:*

- *$\mathfrak{M}$ is of type I_n when $\preceq_{\mathfrak{M}}$ has the same order type as $\{0, 1, \ldots, n\}$;*
- *$\mathfrak{M}$ is of type I_∞ when $\preceq_{\mathfrak{M}}$ has the same order type as $\mathbf{N} \cup \{\infty\}$;*
- *$\mathfrak{M}$ is of type II_1 when $\preceq_{\mathfrak{M}}$ has the same order type as $[0, 1]$;*
- *$\mathfrak{M}$ is of type II_∞ when $\preceq_{\mathfrak{M}}$ has the same order type as $\mathbf{R}_{\geqslant 0} \cup \{\infty\}$;*
- *$\mathfrak{M}$ is of type III when $\preceq_{\mathfrak{M}}$ has the same order type as $\{0, \infty\}$, i.e. all non-zero projections are infinite.*

Proposition 14.0.5 *There exists factors of all types. Moreover, $\preceq_{\mathfrak{M}}$ cannot be of another order type as the ones listed above.*

Proof. Existence of type I factors is clear; the algebra $\mathcal{L}(\mathbb{H})$ with $\mathbb{H}$ a Hilbert space of dimension k ($k \in \mathbf{N}^* \cup \{\infty\}$) is a type I_k factor. For the existence of type II and type III factors, we refer to the first volume of Takesaki's series [189], section V.7, page 362. For the second part of the proposition, we refer once again to the first volume of Takesaki's series [189], Theorem V.1.19 and Corollary V.1.20 pages 296-297. ❀

We can show that a factor of type I_n is isomorphic to $\mathfrak{M}_n(\mathbf{C})$, the algebra of square matrices of size $n \times n$ with complex coefficients. A factor of type I_∞ is isomorphic to $\mathcal{L}(\mathbb{H})$, where $\mathbb{H}$ is an infinite-dimensional Hilbert space.

We will now define the notion of *trace*. One of the important properties of type II_1 factors is the existence of a faithful normal finite trace, i.e. an adequate generalisation of the trace of matrices. Traces, in general, are not defined for all elements, but only for *positive elements*.

Definition 14.0.7 *Let a be an operator in $\mathfrak{M}$ a von Neumann algebra (more generally, a C^*-algebra). We say that a is* positive *if $Spec_{\mathfrak{M}}(a) \subset \mathbb{R}_+$. We denote by $\mathfrak{M}^+$ the set of positive operators in M.*

Proposition 14.0.6 *We have $\mathfrak{M}^+ = \{u^*u \mid u \in \mathfrak{M}\}$.*

Definition 14.0.8 *A* trace τ *on a von Neumann algebra* $\mathfrak{M}$ *is a function from* $\mathfrak{M}^+$ *into* $[0,\infty]$ *satisfying:*

1. $\tau(x+y)=\tau(x)+\tau(y)$ *for all* $x,y\in\mathfrak{M}^+$;
2. $\tau(\lambda x)=\lambda\tau(x)$ *for all* $x\in\mathfrak{M}^+$ *and all* $\lambda\geqslant 0$;
3. $\tau(x^*x)=\tau(xx^*)$ *for all* $x\in\mathfrak{M}$.

We will moreover say that τ *is:*

- faithful *if* $\tau(x)>0$ *for all* $x\neq 0$ *in* $\mathfrak{M}^+$;
- finite *when* $\tau(1)<\infty$;
- semi-finite *when for all element* x *in* $\mathfrak{M}^+$ *there exists* $y\in\mathfrak{M}^+$ *such that* $x-y\in\mathfrak{M}^+$ *and* $\tau(y)<\infty$;
- normal *when* $\tau(\sup\{x_i\})=\sup\{\tau(x_i)\}$ *for all increasing bounded net* $\{x_i\}$ *in* $\mathfrak{M}^+$.

The following theorem can be found in Takesaki's book [189] as Theorem V.2.6 page 312.

Theorem 14.0.7 *If* $\mathfrak{M}$ *is a finite factor (i.e. the identity is a finite projection), then there exists a faithful normal finite trace* τ*. Moreover, every other faithful normal finite trace* ρ *is proportional to* τ*.*

If $\mathfrak{M}$ *is of type* II_1*, we will refer to the unique faithful normal finite trace* tr *such that* $\mathrm{tr}(1)=1$ *as the* normalised trace*.*

Remark 14.0.3 Since the set of positive operators in $\mathfrak{M}$ generates the von Neumann algebra $\mathfrak{M}$, a finite trace τ extends uniquely to a positive linear form on $\mathfrak{M}$ that we will abusively write τ as well. In particular, every operator a in a type II_1 factor has a finite trace.

In order to define the notion of hyperfiniteness we need to define yet another topology on $\mathcal{L}(\mathbb{H})$, the so-called σ-weak topology. This definition is based upon the notion of weak* topology: if X is a space and X^* is its dual, then the weak* topology on X^* is defined as the topology of pointwise convergence on X. To define the σ-weak topology on $\mathcal{L}(\mathbb{H})$ as a weak* topology, we moreover need to see $\mathcal{L}(\mathbb{H})$ as the dual space of some other space. This is a well-known result which can be found in standard textbooks: the algebra $\mathcal{L}(\mathbb{H})$ is the dual of the space of *trace-class operators* that we will denote $\mathcal{L}(\mathbb{H})_*$ and which is itself the dual space of the algebra of compact operators.

Definition 14.0.9 *Let* $\mathbb{H}$ *be a Hilbert space. The* σ*-weak topology on* $\mathcal{L}(\mathbb{H})$ *is defined as the weak** *topology induced by the predual* $\mathcal{L}(\mathbb{H})_*$ *of* $\mathcal{L}(\mathbb{H})$*.*

Remark 14.0.4 If $\mathbb{H}$ is an infinite-dimensional separable Hilbert space, $\mathcal{L}(\mathbb{H})$ embeds into $\mathcal{L}(\mathbb{H}\otimes\mathbb{H})$ through the morphism $x\mapsto x\otimes 1$. One can show that the restriction of the weak operator topology (WOT) on $\mathcal{L}(\mathbb{H}\otimes\mathbb{H})$ coincides with the σ-weak topology on $\mathcal{L}(\mathbb{H})$.

Definition 14.0.10 *A von Neumann algebra* $\mathfrak{M}$ *is* hyperfinite *if there exists a directed family* $\mathfrak{M}_i$ *of finite-dimensional* $*$*-sub-algebras of* $\mathfrak{M}$ *such that* $\cup_i\mathfrak{M}_i$ *is dense in* $\mathfrak{M}$ *for the* σ*-weak topology.*

The following theorems can be found in Takesaki's third volume [192], as Theorem XIV.2.4 page 97 and Theorem XVI.1.22 page 236 respectively.

Theorem 14.0.8 *Two hyperfinite type II_1 factors are isomorphic. We will write $\mathfrak{R}$ the unique hyperfinite type II_1 factor.*

Theorem 14.0.9 *Two hyperfinite type II_∞ factors are isomorphic. In particular, they are isomorphic to the tensor product $\mathfrak{R}_{0,1} = \mathcal{L}(\mathbb{H}) \otimes \mathfrak{R}$.*

Sakai's Theorem and W*-algebras.

We have defined above the von Neumann algebras as sub-algebras of $\mathcal{L}(\mathbb{H})$ where $\mathbb{H}$ is a separable Hilbert space. We therefore defined a von Neumann algebra as a 'concrete' algebra, i.e. as a set of operators acting on a given space. As it is the case with C*-algebras, which can be defined either concretely as a norm-closed sub-algebra of $\mathcal{L}(\mathbb{H})$ or abstractly as an involutive Banach algebra satisfying the C*-identity, there exists an abstract definition of von Neumann algebras. This important result is due to Sakai.

Definition 14.0.11 *Let $\mathfrak{M}$ be a von Neumann algebra. The* pre-dual $\mathfrak{M}_*$ *of $\mathfrak{M}$ is the set of linear forms* which are continuous for the σ-weak topology (Definition 14.0.9).*

The following theorem can be found in Takesaki's first volume [189] (Theorem II.2.6, page 70)

Proposition 14.0.10 *Let $\mathfrak{M}$ be a von Neumann algebra. There exists an isometric isomorphism between $\mathfrak{M}$ and $(\mathfrak{M}_*)^*$ – the dual (as a Banach space) of the pre-dual of $\mathfrak{M}$.*

The reciprocal statement was proved by Sakai [193] and gives an exact characterisation of von Neumann algebras among C*-algebras. A proof can be found in Takesaki [189], Theorem 3.5, page 133, and Corollary 3.9, page 135.

Theorem 14.0.11 *A C*-algebra $\mathfrak{A}$ is a von Neumann algebra if and only if there exists a Banach algebra B such that $\mathfrak{A}$ is the dual of B: $\mathfrak{A} = B^*$. The algebra B is moreover unique (up to isomorphism).*

One can then define von Neumann algebras *abstractly*, i.e. as an abstract algebra *vs* as an algebra of operators acting on a specific space. Such abstract algebras can then be *represented* as algebras of operators.

Definition 14.0.12 *A* representation *of a von Neumann algebra $\mathfrak{M}$ is a pair $(\mathbb{H}, \pi)$ where $\pi : \mathfrak{M} \to \mathcal{L}(\mathbb{H})$ is a C*-algebra homomorphism. If the homomorphism π is injective, we say the associated representation is* faithful.

* We recall that a *linear form on a vector space V* is a linear map from V into $\mathbf{C}$, i.e. an element of the dual of V. When V is a topological vector space, the elements of the topological dual of V are therefore the continuous linear forms.

The Standard Representation.

One of the major results in the theory of von Neumann algebras is that every such algebra has a 'standard representation', i.e. a representation that satisfies a number of important properties. Namely, once realised that von Neumann algebras can be defined in an abstract way, the next step is to identify them with particularly satisfying concrete algebras. A proof of the following result can be found in Takesaki [191], Section IX.1, page 142. The theorem is due to Haagerup [194].

Theorem 14.0.12 *Let $\mathfrak{M}$ be a von Neumann algebra. Then there exists a Hilbert space $\mathbb{H}$, a von Neumann algebra $\mathfrak{S} \subset \mathcal{L}(\mathbb{H})$, an isometric antilinear involution $J : \mathbb{H} \to \mathbb{H}$ and a cone $\mathfrak{P}$ closed under $(\cdot)^*$ such that:*

- *$\mathfrak{M}$ and $\mathfrak{S}$ are isomorphic;*
- *$J\mathfrak{M}J = \mathfrak{M}'$;*
- *$JaJ = a^*$ for all $a \in \mathfrak{Z}(\mathfrak{M})$;*
- *$Ja = a$ for all $a \in \mathfrak{P}$;*
- *$aJaJ\mathfrak{P} = \mathfrak{P}$ for all $a \in \mathfrak{M}$.*

The tuple $(\mathfrak{S}, \mathbb{H}, J, \mathfrak{P})$ is called the standard form *of the algebra $\mathfrak{M}$.*

Let us work out the case of a von Neumann algebra $\mathfrak{M}$ endowed with a faithful normal semi-finite trace. In this case, we can describe a quite easy construction of the standard form of $\mathfrak{M}$. We first define the ideal $\mathfrak{n}_\tau = \{x \in \mathfrak{M} \mid \tau(x^*x) < \infty\}$ (notice that in the case of a finite algebra $\mathfrak{n}_\tau = \mathfrak{M}$). We then consider the map $(\cdot, \cdot)$ from $\mathfrak{M}$ to real numbers defined by:

$$(x, y) = \tau(y^*x)$$

From the linearity of the trace and the anti-linearity of the involution, we can show that it is a sesquilinear form. Moreover, since x^*x is a positive operator, we know that $\tau(x^*x) \geqslant 0$. Therefore, this defines a scalar product on $\mathfrak{n}_\tau$, and we can now define the Hilbert space $L^2(\mathfrak{M}, \tau)$ as the completion of $\mathfrak{n}_\tau$ ($\mathfrak{M}$ when the algebra is finite) for the norm defined by $\|x\|_2 = \tau(x^*x)^{\frac{1}{2}}$.

One can then show that for every element $a \in \mathfrak{M}$ and every $x \in \mathfrak{n}_\tau$:,

$$\begin{aligned} \|ax\|_2 &\leqslant \|a\| \, \|x\|_2 \\ \|xa\|_2 &\leqslant \|a\| \, \|x\|_2 \end{aligned}$$

We then denote by π_τ (resp. π'_τ) the representation of $\mathfrak{M}$ onto $L^2(\mathfrak{M}, \tau)$ by left (resp. right) multiplication.

We then notice that the operation $(\cdot)^*$ defines an isometry on $\mathfrak{n}_\tau$ for the norm $\|\cdot\|_2$. It thus extends to an antilinear involution $J : L^2(\mathfrak{M}, \tau) \to L^2(\mathfrak{M}, \tau)$. One then shows that:

- π_τ (resp. π'_τ) is a faithful representation (resp. antirepresentation[†]);
- $\pi_\tau(\mathfrak{M})' = \pi'_\tau(\mathfrak{M})$ and $\pi'_\tau(\mathfrak{M})' = \pi_\tau(\mathfrak{M})$;
- $J\pi_\tau(a)J = \pi'_\tau(a^*)$ for all $a \in \mathfrak{M}$.

[†] An antirepresentation is a representation that inverses multiplication: $\pi'_\tau(xy) = \pi'_\tau(y)\pi'_\tau(x)$.

Bibliography

Here are the references in citation order.

[1] Dana Scott. *Outline of a mathematical theory of computation*. Tech. rep. PRG02. OUCL, Nov. 1970, p. 30.

[2] N. D. Jones and L. Kristiansen. 'A Flow Calculus of Mwp-bounds for Complexity Analysis'. In: *ACM Trans. Comput. Logic* 10.4 (2009). DOI: 10.1145/1555746.1555752 (cited on pages 2, 10, 16, 100, 118, 120, 125).

[3] Jean-Yves Girard. 'Normal functors, power series and λ-calculus'. In: *Annals of Pure and Applied Logic* 37.2 (1988), pp. 129–177. DOI: 10.1016/0168-0072(88)90025-5 (cited on page 2).

[4] Jean-Yves Girard. 'Linear logic'. In: *Theoretical Computer Science* 50.1 (1987), pp. 1–101. DOI: 10.1016/0304-3975(87)90045-4 (cited on pages 2, 195).

[5] Thomas Ehrhard. 'On Köthe Sequence Spaces and Linear Logic'. In: *Mathematical Structures in Computer Science* 12.5 (2002), pp. 579–623 (cited on page 2).

[6] Thomas Ehrhard and Laurent Regnier. 'Differential interaction nets'. In: *Theoretical Computer Science* 364.2 (2006), pp. 166–195 (cited on page 2).

[7] Jean-Yves Girard. 'Geometry of Interaction V: Logic in the Hyperfinite Factor.' In: *Theoretical Computer Science* 412 (2011), pp. 1860–1883. DOI: 10.1016/j.tcs.2010.12.016 (cited on pages 3, 4, 186, 191, 193, 195, 196, 207).

[8] Jean-Yves Girard. 'Geometry of Interaction IV: the Feedback Equation'. In: *Logic Colloquium '03*. Ed. by Stoltenberg-Hansen and Väänänen. 2006, pp. 76–117 (cited on pages 3, 4, 159, 160).

[9] Thomas Seiller. 'Logique dans le facteur hyperfini : géometrie de l'interaction et complexité'. PhD thesis. Université Aix-Marseille, 2012 (cited on pages 3, 4, 166).

[10] Bent Fuglede and Richard V. Kadison. 'Determinant theory in finite factors'. In: *Annals of Mathematics* 56.2 (1952) (cited on pages 4, 167, 191).

[11] Thomas Seiller. 'A Correspondence between Maximal Abelian Sub-Algebras and Linear Logic Fragments'. In: *Mathematical Structures in Computer Science* 28.1 (2018), pp. 77–139. DOI: 10.1017/S0960129516000062 (cited on pages 4, 11, 190, 193).

[12] Jacques Dixmier. 'Sous-anneaux abéliens maximaux dans les facteurs de type fini'. In: *Annals of mathematics* 59 (1954), pp. 279–286 (cited on pages 4, 199).

[13] Jean-Yves Girard. 'Geometry Of Interaction III: Accommodating The Additives'. In: *Advances in Linear Logic*. Lecture Notes Series 222. Cambridge University Press, 1995, pp. 329–389 (cited on pages 4, 167, 186).

[14] Thomas Seiller. 'Interaction Graphs: Exponentials'. In: *Logical Methods in Computer Science* 15.3 (2019). DOI: 10.23638/LMCS-15(3:25)2019 (cited on pages 4, 9, 189, 204, 208).

[15] Allan Sinclair and Roger Smith. *Finite von Neumann algebras and Masas*. London Mathematical Society Lecture Note Series 351. Cambridge University Press, 2008 (cited on pages 5, 198, 199, 201).

[16] Thomas Seiller. 'Interaction Graphs: Multiplicatives'. In: *Annals of Pure and Applied Logic* 163 (Dec. 2012), pp. 1808–1837. DOI: 10.1016/j.apal.2012.04.005 (cited on pages 5, 9, 161, 166, 167, 170, 172, 186, 203, 204).

[17] Thomas Seiller. 'Interaction graphs: Additives'. In: *Annals of Pure and Applied Logic* 167 (2016), pp. 95–154. DOI: 10.1016/j.apal.2015.10.001 (cited on pages 5, 9, 165, 166, 185, 204, 206).

[18] Thomas Seiller. 'Interaction Graphs: Graphings'. In: *Annals of Pure and Applied Logic* 168.2 (2017), pp. 278–320. DOI: 10.1016/j.apal.2016.10.007 (cited on pages 5, 9, 161, 167, 168, 172, 181, 188, 204).

[19] Scot Adams. 'Trees and amenable equivalence relations'. In: *Ergodic Theory and Dynamical Systems* 10 (01 1990), pp. 1–14 (cited on pages 5, 47).

[20] Gilbert Levitt. 'On the cost of generating an equivalence relation'. In: *Ergodic Theory and Dynamical Systems* 15 (06 1995), pp. 1173–1181. DOI: 10.1017/S0143385700009846 (cited on page 5).

[21] Damien Gaboriau. 'Coût des relations d'équivalence et des groupes'. In: *Inventiones Mathematicae* 139 (2000), pp. 41–98. DOI: 10.1007/s002229900019 (cited on pages 5, 47, 218).

[22] Yasutaka Ihara. 'On discrete subgroups of the two by two projective linear group over $\mathfrak{p}$ -adic fields'. In: *J. Math. Soc. Japan* 18.3 (July 1966), pp. 219–235. DOI: 10.2969/jmsj/01830219 (cited on page 6).

[23] David Ruelle. 'Zeta-functions for expanding maps and Anosov flows'. In: *Inventiones mathematicae* 34.3 (1976), pp. 231–242. DOI: 10.1007/BF01403069 (cited on pages 6, 171).

[24] Eric Finster et al. 'A Cartesian Bicategory of Polynomial Functors in Homotopy Type Theory'. In: *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics, MFPS 2021, Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021*. Ed. by Ana Sokolova. Vol. 351. EPTCS. 2021, pp. 67–83. DOI: 10.4204/EPTCS.351.5 (cited on page 8).

[25] Morgan Rogers, Thomas Seiller, and William Troiani. 'Simplifying normal functors: an old and a new model of λ-calculus'. working paper or preprint. 2024 (cited on page 9).

[26] Clovis Eberhart, Tom Hirschowitz, and Thomas Seiller. 'An Intensionally Fully-abstract Sheaf Model for pi'. In: *6th Conference on Algebra and Coalgebra in Computer Science, CALCO 2015, June 24-26, 2015, Nijmegen, The Netherlands*. Ed. by Lawrence S. Moss and Pawel Sobocinski. Vol. 35. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 86–100. DOI: 10.4230/LIPIcs.CALCO.2015.86 (cited on page 9).

[27] Clovis Eberhart, Tom Hirschowitz, and Thomas Seiller. 'An intensionally fully-abstract sheaf model for π (expanded version)'. In: *Log. Methods Comput. Sci.* 13.4 (2017). DOI: 10.23638/LMCS-13(4:9)2017 (cited on page 9).

[28] John Martin Elliott Hyland and C.-H. Luke Ong. 'On full abstraction for PCF: I, II, and III'. In: *Information and Computation* 163.2 (2000), pp. 285–408 (cited on pages 9, 195).

[29] Hanno Nickau. 'Hereditarily Sequential Functionals'. In: *Proceedings of the Third International Symposium on Logical Foundations of Computer Science*. LFCS '94. Springer-Verlag, 1994, pp. 253–264 (cited on page 9).

[30] Thomas Seiller. 'Interaction Graphs: Full Linear Logic'. In: *IEEE/ACM Logic in Computer Science (LICS)*. 2016 (cited on pages 9, 48, 189).

[31] Lê Thành Dung Nguyên and Thomas Seiller. 'Coherent Interaction Graphs'. In: *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity-TLLA@FLoC 2018, Oxford, UK, 7-8 July 2018*. Ed. by Thomas Ehrhard et al. Vol. 292. EPTCS. 2018, pp. 104–117 (cited on pages 9, 186).

[32] Adrien Ragot, Thomas Seiller, and Lorenzo Tortora de Falco. 'Linear Realisability Over Nets and Second Order Quantification (short paper)'. In: *Proceedings of the 24th Italian Conference on Theoretical Computer Science, Palermo, Italy, September 13-15, 2023*. Ed. by Giuseppa Castiglione and Marinella Sciortino. Vol. 3587. CEUR Workshop Proceedings. CEUR-WS.org, 2023, pp. 59–64 (cited on pages 10, 192).

[33] Adrien Ragot, Thomas Seiller, and Lorenzo Tortora de Falco. 'Linear realisability on untyped nets'. working paper or preprint. 2024 (cited on pages 10, 192).

[34] Boris Eng and Thomas Seiller. 'Multiplicative linear logic from a resolution-based tile system'. In: *CoRR* abs/2207.08465 (2022) (cited on pages 10, 191).

[35] Valentin Maestracci and Thomas Seiller. 'Linear Realisability and Cobordisms'. In: *CoRR* abs/2310.19339 (2023) (cited on page 10).

[36] Stephen Bellantoni and Stephen Cook. 'A new recursion-theoretic characterization of the polytime functions'. In: *Computational Complexity* 2 (1992) (cited on pages 10, 67).

[37] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 'Bounded linear logic: a modular approach to polynomial-time computability'. In: *Theor. Comput. Sci.* 97.1 (Apr. 1992), pp. 1–66. DOI: 10.1016/0304-3975(92)90386-T (cited on page 10).

[38] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 'The size-change principle for program termination'. In: *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*. Ed. by Chris Hankin and Dave Schmidt. ACM, 2001, pp. 81–92 (cited on page 10).

[39] Clément Aubert and Thomas Seiller. 'Characterizing co-NL by a group action'. In: *Mathematical Structures in Computer Science* 26 (4 2016), pp. 606–638. DOI: 10.1017/S0960129514000267 (cited on pages 10, 209, 210).

[40] Clément Aubert and Thomas Seiller. 'Logarithmic Space and Permutations'. In: *Information and Computation* 248 (2016), pp. 2–21. DOI: 10.1016/j.ic.2014.01.018 (cited on pages 10, 209).

[41] Clément Aubert et al. 'Logic Programming and Logarithmic Space'. In: *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*. Ed. by Jacques Garrigue. Vol. 8858. Lecture Notes in Computer Science. Springer, 2014, pp. 39–57. DOI: 10.1007/978-3-319-12736-1_3 (cited on pages 10, 11).

[42] Clément Aubert, Marc Bagnol, and Thomas Seiller. 'Unary Resolution: Characterizing Ptime'. In: *FOSSACS 2016*. 2016 (cited on pages 10, 11).

[43] Jean-Yves Girard. 'Normativity in Logic'. In: *Epistemology versus Ontology*. Ed. by Peter Dybjer et al. Vol. 27. Logic, Epistemology, and the Unity of Science. Springer, 2012, pp. 243–263. DOI: 10.1007/978-94-007-4435-6_12 (cited on page 10).

[44] Clément Aubert and Marc Bagnol. 'Unification and Logarithmic Space'. In: *arXiv preprint arXiv:1402.4327* (2014) (cited on page 11).

[45] Thomas Seiller. 'Interaction Graphs: Nondeterministic Automata'. In: *ACM Transaction in Computational Logic* 19.3 (2018) (cited on pages 11, 209, 212, 213).

[46] Thomas Seiller. 'Implicit complexity through linear realisability: polynomial time and probabilistic classes'. Submitted. 2023 (cited on pages 11, 213).

[47] Thomas Seiller, Luc Pellissier, and Ulysse Léchine. 'Unifying lower bounds for algebraic machines, semantically'. Under revision for publication in Information and Computation, https://hal.archives-ouvertes.fr/hal-01921942. 2022 (cited on pages 12, 135).

[48] Ulysse Léchine and Thomas Seiller. 'Kolmogorov time hierarchy and novelty games'. submitted. 2024 (cited on page 12).

[49] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Berlin, Heidelberg: Springer-Verlag, 1993 (cited on pages 12, 13).

[50] Luc Longpré. 'Resource Bounded Kolmogorov Complexity, A Link between Computational Complexity & Information Theory'. PhD thesis. Cornell University, USA, 1986 (cited on page 13).

[51] Thomas Seiller and Jakob Grue Simonsen. 'An Embellished Account of Agafonov's Proof of Agafonov's Theorem'. hal-02891463. 2020 (cited on page 14).

[52] Thomas Seiller and Jakob Grue Simonsen. 'Agafonov's Theorem for finite and infinite alphabets and probability distributions different from equidistribution'. hal-02993635. 2022 (cited on page 15).

[53] Ulysse Léchine, Thomas Seiller, and Jakob G. Simonsen. 'Agafonov's theorem for probabilistic selectors'. submitted. 2024 (cited on pages 15, 95).

[54] Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. 'Loop Quasi-Invariant Chunk Detection'. In: *Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings*. Ed. by Deepak D'Souza and K. Narayan Kumar. Springer International Publishing, 2017, pp. 91–108. DOI: 10.1007/978-3-319-68167-2_7 (cited on pages 16, 107, 131).

[55] Clément Aubert et al. 'mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity'. In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. 2022. DOI: 10.4230/LIPIcs.FSCD.2022.26 (cited on page 16).

[56] Clément Aubert et al. 'pymwp: A Static Analyzer Determining Polynomial Growth Bounds'. In: *21st International Symposium on Automated Technology for Verification and Analysis (ATVA 2023)*. 2023. DOI: 10.1007/978-3-031-45332-8_14 (cited on pages 16, 132).

[57] Clément Aubert et al. 'Distributing and Parallelizing Non-canonical Loops'. In: *24th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2023)*. 2023. DOI: 10.1007/978-3-031-24950-1_1 (cited on page 17).

[58] Alberto Naibo, Mattia Petrolo, and Thomas Seiller. 'On the computational meaning of axioms'. In: *Epistemology, Knowledge and the Impact of Interaction*. Springer, 2016, pp. 141–184. DOI: 10.1007/978-3-319-26506-3_5 (cited on page 17).

[59] Alberto Naibo, Mattia Petrolo, and Thomas Seiller. 'Verificationism and classical realizability'. In: *Perspectives on Interrogative Models of Inquiry*. Logic, Argumentation & Reasoning. Springer, 2015 (cited on page 18).

[60] Alberto Naibo, Mattia Petrolo, and Thomas Seiller. 'Logical Constants From a Computational Point of View'. in preparation (cited on page 18).

[61] Jean-Baptiste Joinet and Thomas Seiller. 'From abstraction and indiscernibility to classification and types: revisiting Hermann Weyl's theory of ideal elements'. In: *Kagaku tetsugaku* 53.2 (2021), pp. 65–93. DOI: 10.4216/jpssj.53.2_65 (cited on pages 18, 177).

[62] Garrett Birkhoff. *Lattice theory*. Vol. 25. American Mathematical Soc., 1940 (cited on page 18).

[63] Rudolf Wille. 'Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts'. In: *Ordered Sets*. Ed. by Ivan Rival. Dordrecht: Springer Netherlands, 1982, pp. 445–470 (cited on pages 18, 177).

[64] Hermann Weyl. 'Über die Definitionen der mathematischen Grundbegriffe'. In: *Mathematisch-naturwissenschaftliche Blätter* 7 (1910) (cited on page 18).

[65] Hermann Weyl. *Philosophy of Mathematics and Natural Sciences*. Princeton University Press, 1949 (cited on page 18).

[66] Alberto Naibo and Thomas Seiller. 'Algorithms'. entry in Encyclopedia Universalis, to appear (cited on page 19).

[67] Alberto Naibo and Thomas Seiller. 'An ontology of computer science'. in preparation (cited on page 19).

[68] Robin Gandy. 'Church's Thesis and Principles for Mechanisms'. In: *The Kleene Symposium*. Ed. by Jon Barwise, H. Jerome Keisler, and Kenneth Kunen. Vol. 101. Studies in Logic and the Foundations of Mathematics. Elsevier, 1980, pp. 123–148. DOI: https://doi.org/10.1016/S0049-237X(08)71257-6 (cited on page 22).

[69] Y. Gurevich. 'Sequential Abstract State Machines capture Sequential Algorithms'. In: *ACM Transactions on Computational Logic* 1 (2000), pp. 77–111 (cited on pages 22, 26, 92).

[70] Lenore Blum, Mike Shub, and Steve Smale. 'On a Theory of Computation over the Real Numbers; NP Completeness, Recursive Functions and Universal Machines (Extended Abstract)'. In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 387–397. DOI: 10.1109/SFCS.1988.21955 (cited on page 25).

[71] Eric Allender and Fengming Wang. 'On the power of algebraic branching programs of width two'. In: *Comput. Complex.* 25.1 (2016), pp. 217–253. DOI: 10.1007/S00037-015-0114-7 (cited on page 26).

[72] Samuel L. Braunstein and Peter van Loock. 'Quantum information with continuous variables'. In: *Rev. Mod. Phys.* 77 (2 June 2005), pp. 513–577. DOI: 10.1103/RevModPhys.77.513 (cited on page 26).

[73] Christian Weedbrook et al. 'Gaussian quantum information'. In: *Rev. Mod. Phys.* 84 (2 May 2012), pp. 621–669. DOI: 10.1103/RevModPhys.84.621 (cited on page 26).

[74] Yuri Gurevich. 'Specification and Validation Methods'. In: ed. by Egon Börger. New York, NY, USA: Oxford University Press, Inc., 1995. Chap. Evolving Algebras 1993: Lipari Guide, pp. 9–36 (cited on pages 27, 47).

[75] Jean-Louis Krivine. 'A call-by-name lambda-calculus machine'. In: *Higher Order Symbol. Comput.* 20.3 (2007), pp. 199–207. DOI: 10.1007/s10990-007-9018-9 (cited on page 32).

[76] Peter J Landin. 'The mechanical evaluation of expressions'. In: *The computer journal* 6.4 (1964), pp. 308–320 (cited on page 32).

[77] P. J. Landin. 'The next 700 programming languages'. In: *Communications of the ACM* 9.3 (1966), pp. 157–166. DOI: 10.1145/365230.365257 (cited on page 32).

[78] Jean-Louis Krivine. 'Typed lambda-calculus in classical Zermelo-Fraenkel set theory'. In: *Archive for Mathematical Logic* 40.3 (2001), pp. 189–205 (cited on pages 32, 195).

[79] Jean-Louis Krivine. 'Realizability in classical logic'. In: *Panoramas et synthèses* 27 (2009), pp. 197–229 (cited on pages 32, 195).

[80] M. V. Wilkes and W. Renwick. 'The EDSAC (Electronic delay storage automatic calculator)'. In: *Mathematics of Computation* 4 (1950), pp. 61–65 (cited on page 35).

[81] The EDSAC Replica project. 'Tutorial Guide to the EDSAC Simulator' (cited on page 35).

[82] unknown authors. 'The EDSAC: general description'. typewritten reports, archived at https://waldorf.cs.manchester.ac.uk/ftp/pub/CCS-Archive/misc/EDSAC/, captured on February 25th 2024 at https://web.archive.org/web/20240226013456/https://waldorf.cs.manchester.ac.uk/ftp/pub/CCS-Archive/misc/EDSAC/. 1948 (cited on page 35).

[83] Claude E. Shannon. 'Mathematical Theory of the Differential Analyzer'. In: *Journal of Mathematics and Physics* 20.1-4 (1941), pp. 337–354 (cited on page 38).

[84] E.B. Davies. *One-parameter Semigroups*. L.M.S. monographs. Academic Press, 1980 (cited on page 38).

[85] Damien Gaboriau. 'Invariants ℓ^2 de relations d'équivalence et de groupes'. In: *Publ. Math. Inst. Hautes Études Sci* 95.93-150 (2002), pp. 15–28 (cited on pages 47, 218).

[86] Damien Gaboriau. 'Orbit equivalence and measured group theory'. In: *Proceedings of the International Congress of Mathematicians 2010 (ICM 2010) (In 4 Volumes) Vol. I: Plenary Lectures and Ceremonies Vols. II–IV: Invited Lectures*. World Scientific. 2010, pp. 1501–1527 (cited on page 47).

[87] A. Kechris. *Classical Descriptive Set Theory*. Graduate Texts in Mathematics. Springer New York, 2012 (cited on page 57).

[88] S. C. Kleene. 'Recursive Functionals and Quantifiers of Finite Types I'. In: *Transactions of the American Mathematical Society* 91.1 (1959), pp. 1–52 (cited on pages 64, 71).

[89] P. M. Lewis, R. E. Stearns, and J. Hartmanis. 'Memory bounds for recognition of context-free and context-sensitive languages'. In: *6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*. 1965, pp. 191–202. DOI: 10.1109/FOCS.1965.14 (cited on page 75).

[90] Walter Baur and Volker Strassen. 'The complexity of partial derivatives'. In: *Theoretical Computer Science* 22.3 (1983), pp. 317–330. DOI: https://doi.org/10.1016/0304-3975(83)90110-X (cited on page 76).

[91] Xi Chen, Neeraj Kayal, and Avi Wigderson. 'Partial Derivatives in Arithmetic Complexity and Beyond'. In: *Foundations and Trends® in Theoretical Computer Science* 6.1–2 (2011), pp. 1–138. DOI: 10.1561/0400000043 (cited on page 76).

[92] J. Hartmanis and R. E. Stearns. 'On the Computational Complexity of Algorithms'. In: *Journal of Symbolic Logic* 32.1 (1967), pp. 120–121. DOI: 10.2307/2271275 (cited on page 84).

[93] I. H. Sudborough and A. Zalcberg. 'On Families of Languages Defined by Time-Bounded Random Access Machines'. In: *SIAM Journal on Computing* 5.2 (1976), pp. 217–230. DOI: 10.1137/0205018 (cited on page 84).

[94] Neil D. Jones. 'Constant time factors do matter'. In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '93. San Diego, California, USA: Association for Computing Machinery, 1993, pp. 602–611. DOI: 10.1145/167088.167244 (cited on page 84).

[95] Stéphane Lamassé. 'Relationships between French "practical arithmetics" and teaching?' In: *Scientific Sources and Teaching Contexts Throughout History: Problems and Perspectives*. Springer, 2013, pp. 125–153 (cited on page 85).

[96] A. A. Markov. *The theory of algorithms*. Vol. 42. Trudy Mat. Inst. Steklov. Acad. Sci. USSR, 1954, pp. 3–375 (cited on page 85).

[97] A. N. Kolmogorov and V. A. Uspenskii. 'On the definition of an algorithm'. In: *Uspekhi Mat. Nauk* 13 (4(82) 1958), pp. 3–28 (cited on page 85).

[98] Alonzo Church. 'An Unsolvable Problem of Elementary Number Theory'. In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363 (cited on page 85).

[99] M. Airoldi. *Machine Habitus: Toward a Sociology of Algorithms*. Polity Press, 2022 (cited on page 85).

[100] Cathy O'neil. *Weapons of math destruction: How big data increases inequality and threatens democracy*. Crown, 2017 (cited on page 86).

[101] Erwan Le Merrer, Benoıt Morgan, and Gilles Trédan. 'Setting the Record Straighter on Shadow Banning'. In: *40th IEEE Conference on Computer Communications, INFOCOM 2021, Vancouver, BC, Canada, May 10-13, 2021*. IEEE, 2021, pp. 1–10. DOI: 10.1109/INFOCOM42981.2021.9488792 (cited on page 87).

[102] Erwan Le Merrer, Gilles Trédan, and Ali Yesilkanat. 'Modeling rabbit-holes on YouTube'. In: *Soc. Netw. Anal. Min.* 13.1 (2023), p. 100. DOI: 10.1007/S13278-023-01105-9 (cited on page 87).

[103] Florian Jaton. *The Constitution of Algorithms: Ground-Truthing, Programming, Formulating*. MIT Press, 2021 (cited on page 87).

[104] Yiannis Moschovakis. 'What is an Algorithm?' In: *Mathematics Unlimited — 2001 and beyond*. 2001 (cited on pages 88, 97).

[105] Yiannis N. Moschovakis. 'On Founding the Theory of Algorithms'. In: *Truth in Mathematics*. Ed. by H. G. Dales and Gianluigi Oliveri. Oxford University Press, Usa, 1998, pp. 71–104 (cited on page 88).

[106] Yuri Gurevich. 'What Is an Algorithm?' In: *SOFSEM 2012: Theory and Practice of Computer Science: 38th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlyn, Czech Republic, January 21-27, 2012. Proceedings*. Ed. by Mária Bieliková et al. Springer Berlin Heidelberg, 2012, pp. 31–42. DOI: 10.1007/978-3-642-27660-6_3 (cited on page 88).

[107] Andread Blass and Yuri Gurevich. 'Algorithms: A quest for absolute definitions'. In: *Bulletin of the European Association for Theoretical Computer Science* (2003) (cited on page 88).

[108] T.L. Heath. *The Thirteen Books of Euclid's Elements*. vol. 1. Cambridge University Press, 1956 (cited on page 89).

[109] T.L. Heath. *The Thirteen Books of Euclid's Elements*. vol. 2. Cambridge University Press, 1956 (cited on page 89).

[110] T.L. Heath. *The Thirteen Books of Euclid's Elements*. vol. 3. Cambridge University Press, 1956 (cited on page 89).

[111] Josh Alman and Virginia Vassilevska Williams. 'A Refined Laser Method and Faster Matrix Multiplication'. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*. Ed. by Dániel Marx. SIAM, 2021, pp. 522–539. DOI: 10.1137/1.9781611976465.32 (cited on page 93).

[112] Virginia Vassilevska Williams et al. 'New Bounds for Matrix Multiplication: from Alpha to Omega'. In: *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 3792–3835. DOI: 10.1137/1.9781611977912.134 (cited on page 93).

[113] Josh Alman. 'Limits on the Universal Method for Matrix Multiplication'. In: *Theory Comput.* 17 (2021), pp. 1–30 (cited on page 93).

[114] Josh Alman and Virginia Vassilevska Williams. 'Limits on All Known (and Some Unknown) Approaches to Matrix Multiplication'. In: *SIAM J. Comput.* 52.6 (2023), S18–285. DOI: 10.1137/19M124695X (cited on page 93).

[115] Michael Klemm and Bronis R. de Supinski, eds. *OpenMP Application Programming Interface Specification Version 5.2*. OpenMP Architecture Review Board, Nov. 2021 (cited on page 100).

[116] F. R. K. Chung. 'On the coverings of graphs'. In: *Discrete Mathematics* 30.2 (1980), pp. 89–93. DOI: 10.1016/0012-365X(80)90109-0 (cited on page 113).

[117] Lawrence Rauchwerger and David A. Padua. 'Parallelizing While Loops for Multiprocessor Systems'. In: *Proceedings of the 9th International Symposium on Parallel Processing*. IPPS '95. USA: IEEE Computer Society, 1995, pp. 347–356 (cited on page 114).

[118] Joris van der Hoeven and Robin Larrieu. 'Fast Gröbner basis computation and polynomial reduction for generic bivariate ideals'. In: *Applicable Algebra in Engineering, Communication and Computing* 30.6 (Dec. 2019), pp. 509–539. DOI: 10.1007/s00200-019-00389-9 (cited on page 129).

[119] Michael Ben-Or. 'Lower Bounds for Algebraic Computation Trees'. In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. STOC '83. New York, NY, USA: ACM, 1983, pp. 80–86. DOI: 10.1145/800061.808735 (cited on pages 136, 148, 151).

[120] J. M. Steele and A. Yao. 'Lower bounds for algebraic decision trees'. In: *Journal of Algorithms* 3 (1982), pp. 1–8 (cited on page 136).

[121] R. L. Adler, A. G. Konheim, and M. H. McAndrew. 'Topological Entropy'. In: *Transactions of the American Mathematical Society* 114.2 (1965), pp. 309–319 (cited on page 140).

[122] J. E. Hofer. 'Topological entropy for noncompact spaces.' In: *The Michigan Mathematical Journal* 21.3 (1975), pp. 235–242. DOI: 10.1307/mmj/1029001311 (cited on page 140).

[123] R. L. Adler, A. G. Konheim, and M. H. McAndrew. 'Topological Entropy'. In: *Transactions of the American Mathematical Society* 114.2 (1965), pp. 309–319 (cited on page 143).

[124] L. Wayne Goodwyn. 'The Product Theorem for Topological Entropy'. In: *Transactions of the American Mathematical Society* 158.2 (1971), pp. 445–452 (cited on page 143).

[125] Ketan Mulmuley. 'Lower Bounds in a Parallel Model without Bit Operations'. In: *SIAM Journal of Compution* 28.4 (1999), pp. 1460–1509. DOI: 10.1137/S0097539794282930 (cited on pages 147, 150, 152, 218).

[126] J. Milnor. 'On the Betti numbers of real varieties'. In: *Proceedings of the AMS* 15 (1964), pp. 275–280 (cited on page 148).

[127] René Thom. 'Sur l'homologie des variétés algébriques réelles'. In: *Differential and Combinatorial Topology*. Princeton University Press, 1965, pp. 255–265 (cited on page 148).

[128] K. G Murty. 'Computational complexity of parametric linear programming'. In: *Mathematical programming* 19.1 (1980), pp. 213–219 (cited on page 149).

[129] P. J. Carstensen. 'The Complexity of Some Problems in Parametric Linear and Combinatorial Programming'. PhD thesis. Ann Arbor, MI, USA, 1983 (cited on page 149).

[130] Felipe Cucker. '$\mathrm{P}_{\mathbf{R}} \neq \mathrm{NC}_{\mathbf{R}}$'. In: *J. Complexity* 8.3 (1992), pp. 230–238. DOI: 10.1016/0885-064X(92)90024-6 (cited on page 151).

[131] Lenore Blum, Mike Shub, and Steve Smale. 'On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines'. In: *American Mathematical Society. Bulletin. New Series* 21.1 (1989), pp. 1–46. DOI: 10.1090/S0273-0979-1989-15750-9 (cited on page 151).

[132] L. M. Goldschlager, R. A. Shaw, and J. Staples. 'The maximum flow problem is log space complete for P'. In: *Theoretical Computer Science* 21 (1 1982), pp. 105–111. DOI: https://doi.org/10.1016/0304-3975(82)90092-5 (cited on page 154).

[133] André Joyal, Ross Street, and Dominic Verity. 'Traced monoidal categories'. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 119.3 (1996), pp. 447–468. DOI: 10.1017/S0305004100074338 (cited on pages 159, 160, 164).

[134] Octavio Malherbe, Philip J. Scott, and Peter Selinger. 'Partially traced categories'. In: *Journal of Pure and Applied Algebra* 216.12 (2012), pp. 2563–2585. DOI: https://doi.org/10.1016/j.jpaa.2012.03.026 (cited on page 159).

[135] Thomas Seiller. 'Zeta functions and the (linear) logic of Markov processes'. Under revision for publication in Logical Methods in Computer Science, https://hal.archives-ouvertes.fr/hal-02458330. 2022 (cited on pages 163, 173, 189, 190).

[136] Masahito Hasegawa. 'Recursion from Cyclic Sharing: Traced Monoidal Categories and Models of Cyclic Lambda Calculi'. In: Springer Verlag, 1997, pp. 196–213 (cited on page 164).

[137] Esfandiar Haghverdi and Philip Scott. 'A categorical model for the geometry of interaction'. In: *Theoretical Computer Science* 350.2 (2006), pp. 252–274 (cited on page 164).

[138] Jean-Yves Girard. 'Geometry of Interaction I: Interpretation of System F'. In: *In Proc. Logic Colloquium 88*. 1989 (cited on pages 167, 186).

[139] Jean-Yves Girard. 'Geometry of Interaction II: Deadlock-free Algorithms'. In: *Proceedings of COLOG*. Lecture Notes in Computer Science 417. Springer, 1988, pp. 76–93 (cited on pages 167, 186).

[140] Jean-Yves Girard. 'Multiplicatives'. In: *Logic and Computer Science : New Trends and Applications*. Ed. by Lolli. Rendiconti del seminario matematico dell'università e politecnico di Torino, special issue 1987. Torino: Università di Torino, 1987, pp. 11–34 (cited on pages 167, 185, 203).

[141] Audrey Terras. *Zeta functions of graphs: a stroll through the garden*. Vol. 128. Cambridge University Press, 2010 (cited on page 169).

[142] Michael Artin and Barry Mazur. 'On periodic points'. In: *Annals of Mathematics* (1965), pp. 82–99 (cited on page 171).

[143] Tai-Danae Bradley, Juan-Luis Gastaldi, and John Terrilla. 'The structure of meaning in language'. In: *Notices of the American Mathematical Society* (Feb. 2024) (cited on page 176).

[144] Jean-Yves Girard. 'Light Linear Logic'. In: *Selected Papers from the International Workshop on Logical and Computational Complexity*. LCC '94. London, UK, UK: Springer-Verlag, 1995, pp. 145–176 (cited on pages 181, 189, 191, 209).

[145] Christian Retoré. 'Handsome proof-nets: perfect matchings and cographs'. In: *Theoretical Computer Science* 294.3 (2003). Linear Logic, pp. 473–488. DOI: https://doi.org/10.1016/S0304-3975(01)00175-X (cited on page 187).

[146] Vincent Danos and Jean-Baptiste Joinet. 'Linear logic and elementary time'. In: *Information and Computation* 183 (2003), 123Ñ137 (cited on pages 189, 191).

[147] Laszlo Kalmar. 'Egyszerii pelda eldonthetetlen aritmetikai problemara'. In: *Matematikai es Fizikai Lapok* 50 (1943), pp. 1–23 (cited on page 189).

[148] Jean-Yves Girard. 'Locus solum: From the rules of logic to the logic of rules'. In: *Mathematical Structures in Computer Science* 11.3 (2001) (cited on page 191).

[149] Kazushige Terui. 'Computational ludics'. In: *Theor. Comput. Sci.* 412.20 (2011), pp. 2048–2071. DOI: 10.1016/j.tcs.2010.12.026 (cited on page 191).

[150] Jean-Yves Girard. 'Transcendental syntax I: deterministic case'. In: *Math. Struct. Comput. Sci.* 27.5 (2017), pp. 827–849. DOI: 10.1017/S0960129515000407 (cited on page 191).

[151] Henry A. Dye. 'On groups of measure preserving transformations. II.' In: *American Journal of Mathematics* 85 (1963), pp. 119–159 (cited on page 199).

[152] Vaughan F.R. Jones and Sorin Popa. 'Some properties of MASAs in factors'. In: *Invariant subspaces and other topics*. 1982 (cited on page 199).

[153] Ionut Chifan. 'On the normalizing algebra of a masa in a II_1 factor'. In: *Preprint* (2007) (cited on page 199).

[154] Alain Connes, J. Feldman, and B. Weiss. 'An amenable equivalence relation is generated by a single transformation'. In: *Ergodic Theory Dynamical Syst.* 1.4 (1981), pp. 431–450 (cited on page 200).

[155] L. Pukanszky. 'On maximal albelian subrings of factors of type II_1'. In: *Canad. J. Math.* 12 (1960), pp. 289–296 (cited on page 200).

[156] Stuart White. 'Tauer Masas in the Hyperfinite II_1 Factor'. In: *Quarterly Journal of Mathematics* 57 (2006), pp. 377–393 (cited on page 201).

[157] Stuart White and Allan Sinclair. 'A continuous path of singular masas in the hyperfinite II_1 factor'. In: *J. London Math. Soc.* 75 (2 2007), pp. 243–254 (cited on page 201).

[158] Minoru Tomita. 'Quasi-Standard von Neumann Algebras'. Mimeographed Notes (Kyushu University). 1967 (cited on page 202).

[159] Jean-Yves Girard. 'Proof nets: the parallel syntax for proof theory'. In: *Logic and Algebra*. Ed. by Agliano Ursini. Lecture Notes in Pure and Applied Mathematics 180. M. Dekker, 1995 (cited on page 203).

[160] Stuart White. 'Values of the Pukanszky invariant in McDuff factors'. In: *Journal of Functional Analysis* 254 (2008), pp. 612–631 (cited on page 208).

[161] Vincent Danos and Jean-Baptiste Joinet. 'Linear Logic & Elementary Time'. In: *Information and Computation* 183.1 (2003), pp. 123–137. DOI: 10.1016/S0890-5401(03)00010-5 (cited on page 209).

[162] Markus Holzer, Martin Kutrib, and Andreas Malcher. 'Complexity of multi-head finite automata: Origins and directions'. In: *Theoretical Computer Science* 412.1 (2011). Complexity of Simple Programs, pp. 83–96. DOI: https://doi.org/10.1016/j.tcs.2010.08.024 (cited on page 214).

[163] Ioan I. Macarie. 'Multihead Two-Way Probabilistic Finite Automata'. In: *Theory Comput. Syst.* 30.1 (1997), pp. 91–109. DOI: 10.1007/s002240000043 (cited on page 214).

[164] Stephen A. Cook. 'Characterizations of Pushdown Machines in Terms of Time-Bounded Computers'. In: *J. ACM* 18.1 (Jan. 1971), pp. 4–18. DOI: 10.1145/321623.321625 (cited on page 214).

[165] I. M. Singer. 'Automorphisms of Finite Factors'. In: *American Journal of Mathematics* 77.1 (1955), pp. 117–133 (cited on page 217).

[166] M. Chrobak and M. Li. 'k+1 heads are better than k for PDA's'. In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 1986, pp. 361–367. DOI: 10.1109/SFCS.1986.27 (cited on page 218).

[167] Ming Li, Paul Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*. Vol. 3. Springer, 2008 (cited on page 218).

[168] Stefano Galatolo, Mathieu Hoyrup, and Cristóbal Rojas. 'Effective symbolic dynamics, random points, statistical behavior, complexity and entropy'. In: *Information and Computation* 208.1 (2010), pp. 23–41. DOI: https://doi.org/10.1016/j.ic.2009.05.001 (cited on page 218).

[169] Avi Wigderson. 'The fusion method for lower bounds in circuit complexity'. In: *Combinatorics, Paul Erdos is Eighty* 1.453-468 (1993), p. 68 (cited on page 218).

[170] Michael Sipser. 'Borel Sets and Circuit Complexity'. In: *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*. Ed. by David S. Johnson et al. ACM, 1983, pp. 61–69. DOI: 10.1145/800061.808733 (cited on page 218).

[171] Michael Sipser. 'A Topological View of Some Problems in Complexity Theory'. In: *Mathematical Foundations of Computer Science 1984, Praha, Czechoslovakia, September 3-7, 1984, Proceedings*. Ed. by Michal Chytil and Václav Koubek. Vol. 176. Lecture Notes in Computer Science. Springer, 1984, pp. 567–572. DOI: 10.1007/BFb0030341 (cited on page 218).

[172] Alexander A. Razborov. 'On the Method of Approximations'. In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*. Ed. by David S. Johnson. ACM, 1989, pp. 167–176. DOI: 10.1145/73007.73023 (cited on page 218).

[173] Richard Beigel. 'The Polynomial Method in Circuit Complexity'. In: *Proceedings of the Eigth Annual Structure in Complexity Theory Conference, San Diego, CA, USA, May 18-21, 1993*. IEEE Computer Society, 1993, pp. 82–95. DOI: 10.1109/SCT.1993.336538 (cited on page 218).

[174] Richard Ryan Williams. 'The Polynomial Method in Circuit Complexity Applied to Algorithm Design (Invited Talk)'. In: *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*. Ed. by Venkatesh Raman and S. P. Suresh. Vol. 29. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 47–60. DOI: 10.4230/LIPIcs.FSTTCS.2014.47 (cited on page 218).

[175] Valentine Kabanets and Russell Impagliazzo. 'Derandomizing Polynomial Identity Tests Means Proving Circuit Lower Bounds'. In: *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '03. San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 355–364. DOI: 10.1145/780542.780595 (cited on page 218).

[176] Ryan Williams. 'Nonuniform ACC Circuit Lower Bounds'. In: *J. ACM* 61.1 (Jan. 2014). DOI: 10.1145/2559903 (cited on page 218).

[177] Zeyuan Allen-Zhu et al. 'Operator Scaling via Geodesically Convex Optimization, Invariant Theory and Polynomial Identity Testing'. In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 172–181. DOI: 10.1145/3188745.3188942 (cited on page 219).

[178] Kazushige Goto and Robert A. van de Geijn. 'Anatomy of High-Performance Matrix Multiplication'. In: *ACM Trans. Math. Softw.* 34.3 (2008). DOI: 10.1145/1356052.1356053 (cited on page 220).

[179] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. 'Good predictions are worth a few comparisons'. In: *STACS 2016*. Vol. 47. Orléans, France, Feb. 2016, 12:1–12:14. DOI: 10.4230/LIPIcs.STACS.2016.12 (cited on page 220).

[180] Reinhard Wilhelm et al. 'The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools'. In: *ACM Trans. Embed. Comput. Syst.* 7.3 (2008). DOI: 10.1145/1347375.1347389 (cited on page 221).

[181] John von Neumann. 'Zür Algebra der Funktionaloperatoren und Theorie der normalen Operatoren'. In: *Mathematische Annalen* 102 (1930), pp. 370–427 (cited on page 223).

[182] Francis Murray and John von Neumann. 'On Rings of Operators'. In: *Annals of Mathematics* 37.2 (1936), pp. 116–229 (cited on pages 223, 225).

[183] Francis Murray and John von Neumann. 'On rings of operators. II'. In: *Transactions of the American Mathematical Society* 41 (2 1937), pp. 208–248 (cited on page 223).

[184] John von Neumann. 'On infinite direct products'. In: *Compositiones Mathematicae* 6 (1938), pp. 1–77 (cited on page 223).

[185] John von Neumann. 'On rings of operators III'. In: *Annals of Mathematics* 41 (1940), pp. 94–161 (cited on page 223).

[186] Francis Murray and John von Neumann. 'On rings of operators. IV'. In: *Annals of Mathematics* 44 (1943), pp. 716–808 (cited on page 223).

[187] John von Neumann. 'On Some Algebraical Properties of Operator Rings'. In: *Annals of Mathematics* 44 (1943), pp. 709–715 (cited on page 223).

[188] John von Neumann. 'On Rings of Operators. Reduction Theory'. In: *Annals of Mathematics* 50 (1949), pp. 401–485 (cited on pages 223–225).

[189] Masamichi Takesaki. *Theory of Operator Algebras 1*. Vol. 124. Encyclopedia of Mathematical Sciences. Springer, 2001 (cited on pages 224, 226–228).

[190] Izrail Gelfand. 'Normierte Ringe'. In: *Matematicheskii Sbornik* 51 (1941), pp. 3–24 (cited on page 225).

[191] Masamichi Takesaki. *Theory of Operator Algebras 2*. Vol. 125. Encyclopedia of Mathematical Sciences. Springer, 2003 (cited on pages 225, 229).

[192] Masamichi Takesaki. *Theory of Operator Algebras 3*. Vol. 127. Encyclopedia of Mathematical Sciences. Springer, 2003 (cited on page 227).

[193] Shoichiro Sakai. *C*-algebras and W*-algebras*. English. Springer-Verlag, New York ; Berlin : 1971, xii, 256 p. (Cited on page 228).

[194] Uffe Haagerup. 'The Standard Form of von Neumann Algebras'. In: *Math. Scand.* 37.271-283 (1975) (cited on page 229).

List of Figures

List of Tables

Alphabetical Index