Linear Logic and Sub-polynomial Classes of Complexity

Clément Aubert

Université Paris 13
Sorbonne Paris Cité
Laboratoire d'Informatique de Paris Nord

Logique linéaire et classes de complexité sous-polynomiales

Linear Logic and Sub-polynomial Classes of Complexity

Thèse pour l'obtention du diplôme de
Docteur de l'Université de Paris 13,
Sorbonne Paris-Cité,
en informatique
présentée par

Clément Aubert
<aubert@lipn.fr>

Mémoire soutenu le mardi 26 novembre 2013, devant la commission d'examen composée de :

M.	Patrick Baillot	C.N.R.S., E.N.S. Lyon	*Rapporteur*
M.	Arnaud Durand	Université Denis Diderot - Paris 7	*Président*
M.	Ugo Dal Lago	I.N.R.I.A., Università degli Studi di Bologna	*Rapporteur*
Mme.	Claudia Faggian	C.N.R.S., Université Paris Diderot - Paris 7	
M.	Stefano Guerrini	Institut Galilée - Université Paris 13	*Directeur*
M.	Jean-Yves Marion	Lo.R.I.A., Université de Lorraine	
M.	Paul-André Melliès	C.N.R.S., Université Paris Diderot - Paris 7	
M.	Virgile Mogbil	Institut Galilée - Université Paris 13	*Co-encadrant*

⁎ Créteil, le 7 novembre 2014 ⁎

Errata

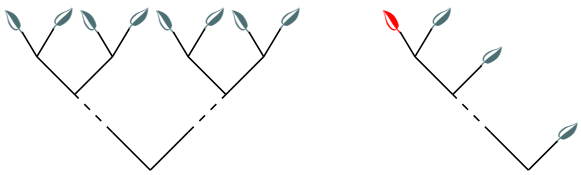
Some minor editions have been made since the defence of this thesis (improvements in the bibliography, fixing a few margins and some typos): they won't be listed here, as those errors were not altering the meaning of this work.[1]

However, while writing an abstract for TERMGRAPH 2014, I realized that the simulation of Proof Circuits by an Alternating Turing Machine, in the Section 2.4 of Chapter 2 was not correct.

More precisely, $\mathbf{bPCC}^{\mathbf{i}}$ is not an object to be considered (as it is trivially equal to $\mathbf{PCC}^{\mathbf{i}}$) and Theorem 2.4.3, p. 51, that states that "For all $i > 0$, $\mathbf{bPCC}^{\mathbf{i}} \subseteq \mathbf{STA}(\log, *, \log^{\mathbf{i}})$", is false. The ATM constructed do normalize the proof circuit given in input, but not within the given bounds. This flaw comes from a mismatch between the *logical depth* of a proof net (Definition 2.1.10) and the *height of a piece* (Definition 2.2.6), which is close to the notion of *depth of a Boolean circuit* (Definition 2.1.2).

First, remark that the class $\mathbf{bPCC}^{\mathbf{i}}$ (Definition 2.2.8) is ill-defined: recall that (Lemma 2.3.1) in the case of $\mathscr{D}^k_{\text{isj}}$ and $\mathscr{C}^k_{\text{onj}}$ pieces, one entry is at the same logical depth than the output, and all the other entries are at depth 3 plus the depth of the output.

In the process of decomposing a single n-ary operation in $n-1$ binary operations, the "depth-efficient way" is not the "*logical* depth-efficient way". Let us consider a tree with n leafs (🍃) and two ways to organize it:



Let us take the *distance* to be the number of edges between a leaf and the root of the tree. Then, on the left tree, every leaf is at the same distance $\log(n)$, and on the right tree, the greatest distance is n.

Everything differs if one consider that every binary branching is a $\mathscr{D}^2_{\text{isj}}$ or $\mathscr{C}^2_{\text{onj}}$ piece and consider the logical depth rather than the distance. In that case, on the left-hand tree, the ith leaf (starting from the left) is at logical depth 3 times the number of 1 in the binary encoding of i.[2] On the right-hand tree, every leaf is at logical depth 3, except for the red leaf (the left-most one), which is at logical depth 0.

So the logical depth is in fact independent of the fan-in of the pieces, and it has the same "logical cost" in terms of depth to compute $\mathscr{D}^i_{\text{isj}}$ or $\mathscr{C}^i_{\text{onj}}$ for any $i \geqslant 2$. A proof circuit of $\mathbf{PCC}^{\mathbf{i}}$ *is* a proof circuit of $\mathbf{bPCC}^{\mathbf{i}}$: every piece of $\mathscr{P}_u$ can be obtained from pieces of $\mathscr{P}_b$ *without increasing the logical depth*. So we have that for all $i \in \mathbb{N}$,

$$\mathbf{PCC}^{\mathbf{i}} = \mathbf{bPCC}^{\mathbf{i}}.$$

Secondly, the proof of Theorem 2.4.3 was an attempt to adapt the function value(n, g, p) [135, p. 65] that prove that an ATM can normalize a Boolean circuit with suitable bounds. The algorithm is correct,

[1] Except maybe for a silly mistake in Proposition 4.3.1, spotted by Marc Bagnol.
[2] Sic. This sequence is known as A000120 and starts with $0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, \ldots$.

but not the bounds: it is stated that "Each call to f uses a constant number of alternations, and $\mathcal{O}(d(\mathsf{P}_n))$ calls are made. As P_n is of depth $\log^i(n)$, $\mathcal{O}(\log^i(n))$ calls are made.", and this argument is wrong.

Consider that $\mathsf{P}_n \in \mathbf{PCC}^{\mathrm{i}}$ (this is harmful taking the first remark into account), when the ATM calls f to evaluate the output of a $\mathscr{D}^k_{\mathrm{isj}}$ or $\mathscr{C}^k_{\mathrm{onj}}$ piece, it makes $k-1$ calls to f at a lesser depth, and one call at the same depth. The only bound on the fan-in of the pieces, k, is the size of the proof circuit, that is to say a polynomial in the size of the input: this is a disaster.

tl;dr: bPCC$^{\mathrm{i}}$ should not be considered as a pertinent object of study and the correspondence between the proof circuits and the Alternating Turing Machines is partially wrong. This does not affect anyhow the rest of this work and this part has never been published nor submitted.

Créteil, 7 November 2014.

This thesis is dedicated to the free sharing of knowledge.

Cette thèse est dédiée au libre partage des savoirs.

Disclaimer: *It is not possible for me to write these acknowledgements in a language other than French.*

Remerciements. Il est d'usage de commencer par remercier son directeur de thèse, et je me plie à cet usage avec une certaine émotion : Stephano Guerrini m'a accordé une grande confiance en acceptant de m'encadrer dans cette thèse en Informatique alors que mon cursus ne me pré-disposait pas tellement à cette aventure. Par ses conseils, ses intuitions et nos discussions, il a su m'ouvrir des pistes, en fermer d'autres, conseiller et pointer de fructueuses lectures.

Virgile Mogbil a co-encadré cette thèse avec une attention et un soin qui m'ont touchés. Il a su m'accompagner dans tous les aspects de celles-ci, m'initier à la beauté de la recherche sans me cacher ses défauts, me guider dans le parcours académique et universitaire bien au-delà de ma thèse. Il su faire tout cela dans des conditions délicates, ce qui rend son dévouement encore plus précieux.

Patrick Baillot fût le rapporteur de cette thèse et comme son troisième directeur : il a su durant ces trois années me conseiller, me questionner, chercher à comprendre ce que je ne comprenais pas moi-même. C'est un honneur pour moi qu'il ait accepté de rapporter ce travail avant autant de justesse et de clairvoyance.

Je suis également ravi qu'Ugo Dal Lago ait accepté de rapporter ce travail, et m'excuse pour l'avalanche de coquilles qui ont dû lui abîmer l'œil. Sa place centrale dans notre communauté donne une saveur toute particulière à ses remarques, conseils et critiques.

Je suis honoré qu'Arnaud Durand — qui fût mon enseignant lorsque je me suis reconverti aux Mathématiques — ait accepté d'être le président de ce jury. La présence de Claudia Faggian, Jean-Yves Marion et Paul-André Melliès place cette thèse à l'interscetion de communautés auxquelles je suis ravi d'appartenir. Je les remercie chaleureusement pour leur présence, leurs conseils, leurs encouragements.

Une large partie de ce travail n'aurait pas pu voir le jour sans la patience et la générosité de Thomas Seiller, avec qui j'ai toujours eu plaisir à rechercher un accès dans la compréhension des travaux de Jean-Yves Girard.

§

Mon cursus m'a permis de rencontrer de nombreux enseignants qui ont tous, d'une façon ou d'une autre, rendu ce cheminement possible. J'ai une pensée particulière pour Louis Allix, qui m'a fait découvrir la logique et pour Jean-Baptiste Joinet, qui a accompagné, encouragé et rendu possible ce cursus. De nombreux autres enseignants, tous niveaux et matières confondus, m'ont fait aimer le savoir, la réflexion et le métier d'enseignant, qu'ils m'excusent de ne pouvoir tous les nommer ici, ils n'en restent pas moins présents dans mon esprit.

§

J'ai été accueilli au Laboratoire d'Informatique de Paris Nord par Paulin Jacobé de Naurois (qui encadrait mon stage), Pierre Boudes et Damiano Mazza (qui ont partagé leur bureau avec moi), et je ne pouvais rêver de meilleur accueil. L'équipe Logique, Calcul et Raisonnement offre un cadre de travail amical et chaleureux dans lequel je me suis senti accompagné, écouté et bienvenu.

Devenir enseignant dans une matière que je connaissais mal (pour ne pas dire pas) aurait été une tâche insurmontable pour moi si elle n'avait eu lieu à l'IUT de Villetaneuse, où j'ai trouvé une équipe dynamique et respectueuse de mes capacités. Laure Petrucci, Camille Coti, Jean-Michel Barrachina, Fayssal Benkhaldoun et Emmanuel Viennet m'ont tous énormément appris dans leur accompagnement patient et attentif.

Ce fût toujours un plaisir que de discuter avec mes collègues du L.I.P.N., ainsi qu'avec le personnel administratif de Paris 13. Brigitte Guéveneux, tout particulièrement, sait marquer de par sa bonne humeur et son accueil chaleureux tout personne qui pénètre dans ces locaux, qu'elle en soit remerciée.

§

La science ne vit qu'incarnée, et je remercie Claude Holl d'avoir tout au long des formations ALICE — aux intervenants exceptionnels — su placer la recherche et ses acteurs en perspective. Mes perspectives à moi ont souvent été questionnées, critiquées, par la presse libre (C.Q.F.D., Article XI, Fakir, La décroissance, Le Tigre, et bien d'autres), qui a su faire de moi un meilleur individu.

Mon engagement associatif s'est construit à côté, et parfois à contre-courant, de cette thèse, mais je sais aujourd'hui qu'il a été grandi par ce travail, qui a également profité de ces échanges. J'ai bien évidemment une pensée toute particulière pour microlab, mais je n'oublie par La Goutte d'ordinateur, l'écluse, et tous les acteurs rémois et parisiens de la scène *d.i.y.*

Le lien entre toutes ses dimensions a trouvé un écho dans les discussions avec les doctorants, tous domaines confondus, que j'ai eu la chance de croiser. Je tiens notamment à remercier et à saluer[3] Aloïs Brunel, Sylvain Cabanacq, Andrei Dorman, Antoine Madet, Alberto Naibo, Mattia Petrolo, Marco Solieri, mais n'oublie aucun de ceux avec qui j'ai eu la chance de dialoguer.

Ma famille est passé par toutes sortes d'épreuves durant cette thèse, mais a toujours su m'écouter, me comprendre, m'encourager : me faire confiance et me soutenir.

Enfin, merci à Laura, pour notre amour.
Et merci à Olivier, pour — entre mille autres choses — ses sourires.

Et un merci spécial à l'équipe L.D.P. de l'Institut de Mathématiques de Luminy qui a su m'offrir un cadre serein où finaliser agréablement ce travail…

[3]Dans le « désordre alphabétique », comme l'écrivait Jean-Baptise Joinet.

Tools. This thesis was typeset in the URW Garamond No.8 fonts,[4] with the distribution TeX Live 2014, using pdfTEX and the class Memoir.

The source code was created and edited with various free softwares, among which Git who trustfully backuped and synced and Texmaker.

I am greatly in debt to the TeX - LaTeX Stack Exchange community, and grateful to Siarhei KHIREVICH's "Tips on Writing a Thesis in LATEX", that surely made me lose some time by pushing me forward on the rigorous typesetting.

The bibliography could not have been without DBLP and the drawings owe a lot to some of the examples gathered at TEXample.net.

[4] Whose glyphs are digital renderings of fonts actually designed by Claude GARAMONT in the mid sixteenth century!

Contents

List of Tables and Figures

Page

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Appendix A

Appendix B

Notations

The entries are divided between Complexity, Logic and Mathematics. Although some symbols are used in different ways, the context should always makes clear which denotation we have in mind.

We deal with classes of complexity separately, below this table of notations.

Complexity

ATM	Alternating Turing Machine . . . 6
$\mathfrak{B}_b$	A set of (bounded) Boolean functions (a basis) . . . 22
$\mathfrak{B}_u$	A set of (unbounded) Boolean functions (a basis) . . . 22
$\mathbf{C}$	Family of Boolean circuit . . . 23
C, C_n	Boolean circuits . . . 22
C_M	The set of pseudo-configurations of a PM(p) M . . . 108
DFA(k)	Set of deterministic Multi-Head Finite Automaton with k heads . . . 99
DPM(p)	The set of deterministic pointer machines with p pointers . . . 108
$G_{M(x)}$	Computation graph of an ATM M on input x . . . 8
$G_{M,n}$	Computation graph of an ATM M on input of size n . . . 46
$\widehat{(\cdot)}$	Translation from families of computation graphs to proof circuit families . . . 47
$\mathcal{L}(M)$	Language accepted by a machine M . . . 7
$\mathrm{L}_{\mathrm{DC}}(\mathbf{C})$	Direct Connection Language of a family of Boolean circuits $\mathbf{C}$ . . . 23
$\mathrm{L}_{\mathrm{DC}}(\mathsf{P})$	Direct Connection Language of a family of Boolean proof nets P . . . 32
[not a]	Any symbol of the alphabet except a . . . 99
NFA(k)	Set of Non-Deterministic Multi-Head Finite Automaton with k heads . . . 98
NPM(p)	The set of non-deterministic pointer machines with p pointers . . . 108
$\mathcal{O}$	"Big O" asymptotic notation . . . 5

$\mathrm{PM}(p)$	The set of (deterministic and non-deterministic) pointer machines with p pointers.	108
$P_{+,1}$	Set of boolean observations of 1-norm $\leqslant 1$	83
P_{+}	Set of boolean observations	83
$P_{\geqslant 0}$	Set of positive observations	83
$Q^{\uparrow}$	Extended set of states	110
$\sharp H$	Distance between $\triangleright$ and the bit currently read by the head H	99
σ	Transition relation of a NDPM	98
$\lvert x\rvert$	The size of x	5
$\star$	First and last bit of a circular binary word	69
$\triangleright$	Left end-marker on the input tape of a FA	98
$\triangleleft$	Right end-marker on the input tape of a FA	98
$\vdash_{M(x)}$	The transition relation between configurations in an ATM M on input x	7
$< 1^{y}, \overline{g}, \overline{p}, \overline{b} >$	The tuples used to describe a proof net	32
$\{X\}$	Language decided by the set of observations X	82

Logic

$A[B/D]$	The substitution of every occurrence of B by D in A	24
$A[D]$	The substitution of every occurrence of α by D in A	24
α	Literal (or atom)	13
$\overrightarrow{A}$, (resp. $\overleftarrow{A}$)	The ordered sequence of formulae $A_1,\ldots,A_n$, (resp. $A_n,\ldots,A_1$)	24
$\rightrightarrows$	The parallel evaluation of a proof net	28
$\mathbf{B}$	The Boolean type of $\mathrm{MLL}_{\mathrm{u}}$	28
b_0, b_1	Proof nets representing 0 and 1 in $\mathrm{MLL}_{\mathrm{u}}$	28
$(\cdot)^{\perp}$	Negation, or duality	13
P	Boolean proof net	30
$\mathbf{P}$	A Boolean proof net familly	30
$d(A)$	Depth of a $\mathrm{MLL}_{\mathrm{u}}$ formula A	25
$\rightarrow_{\mathrm{ev.}}$	The "evolution" through normalisation of a Boolean proof net	30

$\vdash \Gamma$	A sequent, Γ being a multiset of formulae	13
$\Rightarrow$	Classical implication	12
$\multimap$	Linear implication	13
$\mathrm{MLL}_{\mathrm{u}}$	Multiplicative Linear Logic	24
$\bullet$	An axiom link, in a proof net	25
!	The “of course” modality	13
$\parr$	The multiplicative disjunction (“par”)	13
$\mathscr{P}$	Piece of a proof circuit	33
$\mathscr{P}_b$	Set of pieces of bounded fan-in	35
$\mathscr{P}_u$	Set of pieces of unbounded fan-in	35
$\mathcal{P}, \mathcal{Q}$	Proof nets of $\mathrm{MLL}_{\mathrm{u}}$	25
π	A proof, in the sequent presentation	14
$\otimes$	The multiplicative conjunction (“tensor”)	13
?	The “why not” modality	13

Mathematics

α	Action of a group	137
$\lvert\cdot\rvert$	Absolute value	132
$\mathcal{B}(\mathbb{H})$	Banach algebra of continuous mapping over $\mathbb{H}$	131
det(M)	Determinant of a matrix M	134
d	Distance function	126
δ_{ij}	Kronecker delta	126
$\sim_{\mathfrak{M}}$	Murray and von Neumann equivalence over the von Neumann algebra $\mathfrak{M}$	133
$\mathscr{F}$	Field	127
$\lfloor\cdot\rfloor$	Floor function	102
$\mathscr{G}$	Group	127
$\mathbb{H}$	Hilbert space	130
$\ell^2(S)$	Square-summable functions over S	132

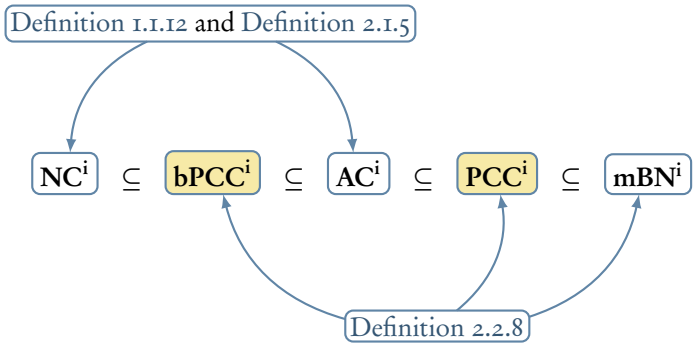
$\mathrm{M}[i,j]$	Element in the ith row and jth column of M	134
$\mathrm{M}_{\mathrm{n,m}}(\mathscr{F})$	Matrix of dimensions $n \times m$ with coefficients in the field $\mathscr{F}$	134
$\mathfrak{M}$	von Neumann algebra	131
$\mathfrak{M} \rtimes \mathscr{G}$	Crossed product of a von Neumann algebra $\mathfrak{M}$ with a group $\mathscr{G}$	137
$\|\|\cdot\|\|_1^k$	The 1-norm	82
$\pi_{k,i}$	Projection of dimension $(k+1)\times(k+1)$	74
$\Pi(\mathfrak{M})$	Set of projections of a von Neumann algebra $\mathfrak{M}$	133
$\mathcal{P}_{\mathrm{fin}}(S)$	Set of finite subsets of S	6
$\mathfrak{R}$	The II_1 hyperfinite factor	134
$(\cdot)^*$	Involution	130
$\mathscr{S}(S)$	Permutation group of S	128
$\mathscr{S}$	Group of all finite permutations over $\mathbb{N}$	128
$\mathrm{sgn}(\sigma)$	Signature of a permutation σ	128
$\otimes$	Tensor product	133
$\mathcal{T}$	Topology (over a set)	131
$\mathrm{tr}(\mathrm{M})$	Trace of a matrix M	134
$\mathrm{Tr}(\mathrm{M}_n)$	Normalised trace of a $n \times n$ matrix M_n	135
$\underline{\mathscr{G}}$	Underlying set of the algebraic structure $\mathscr{G}$	127
V	Vector space	128

See referenced pages for formal definitions.

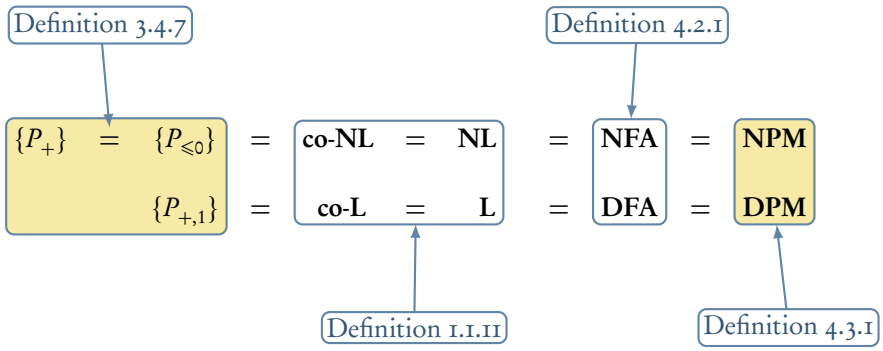
We are going to use several classes of complexity, some of them classical, some of them being introduced in this work (in a yellow rectangle in the figure below). The following pictures describe the situation at the end of our work. Some equalities and inclusions are classical, but we will in the course of this work give at least ideas of proofs for every of them.

For all $i > 1$,[5] we have at the end of Chapter 2[6] the following situation:

[5] Even if those bounds are going to be sharpened, we prefer to give our results in all generality at this point.
[6] See errata.



Chapter 3 and Chapter 4 will define and prove properties about the following classes:



We are also going to introduce **STA**$(*,*,*)$ (Definition 1.1.7) as a convenient notation to express any of these classes.

Preface

This work is somehow two-sided, for it studies two models of computation —and the implicit characterization of complexity classes that comes with— that share the same tool and the same aim, but are not directly related. The tool —mainly proof theory, and more precisely Linear Logic— and aim —to contribute to the complexity theory— are presented in Chapter 1. The Section 1.3 presents some of the previous attempts to link those two fields.

The first model —presented in Chapter 2— is related to the simulation of efficient computation in parallel with Boolean proof nets. It introduces proof circuits as an innovative way of understanding the work led by Terui. It sharpens the characterization of some complexity classes, and provides new evidence that this model is pertinent by linking it to the more classical Turing Machines. This chapter is —almost— self-contained and a gentle introduction to some important concepts.

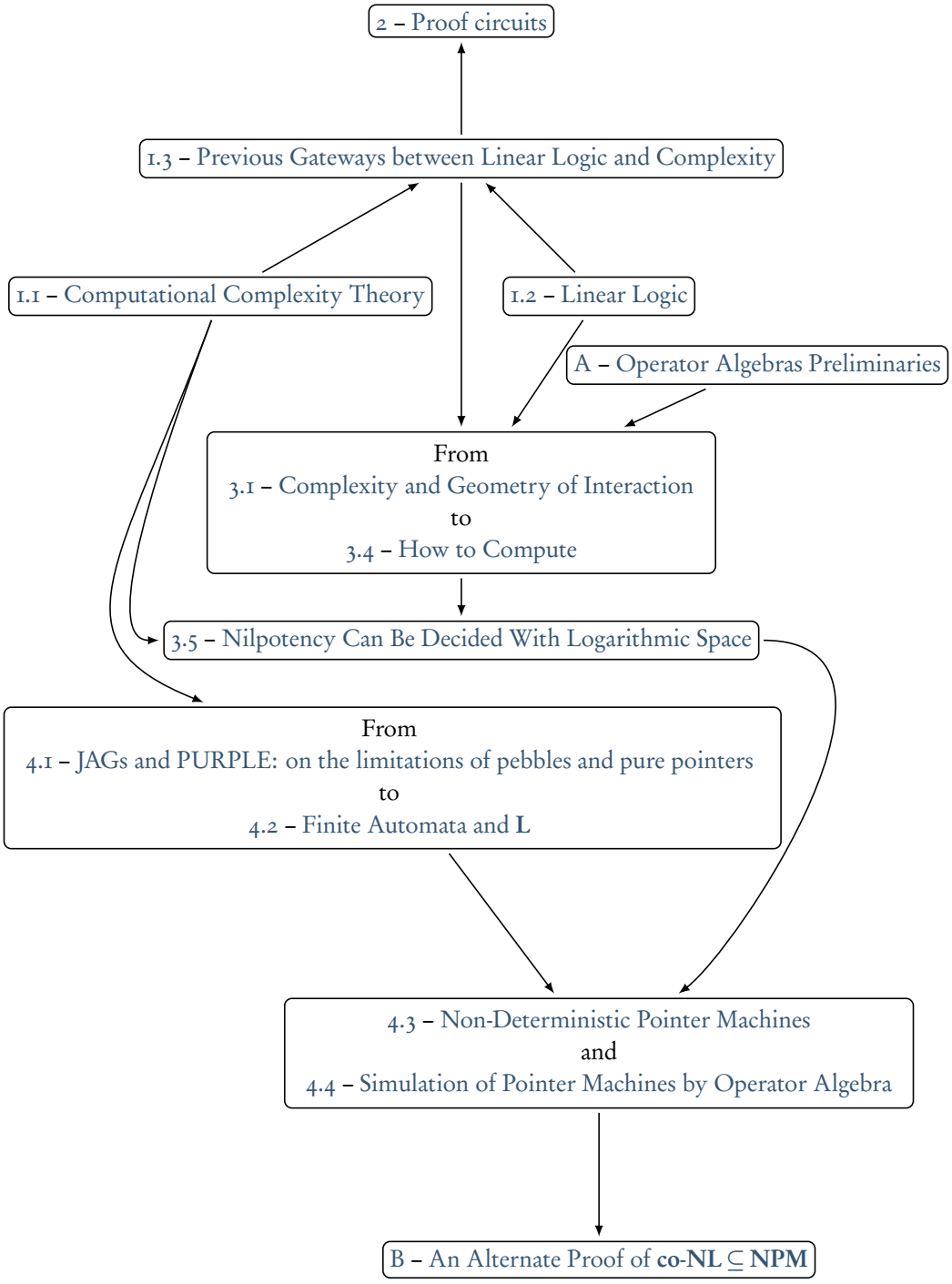
The second model is presented in Chapter 3, it develops a new world where logic is dynamically built from von Neumann algebras. We tried to ease the reading of this quite technical material by pushing slowly the constructions and taking time to explain some of the choices we made. We finish this first part by showing how complex our object is from a complexity point of view.

Chapter 4 gently presents some "pointer machines" before defining a model that is proven adapted to be simulated by the model of Chapter 3. It recollects some historical variation to help the reader to understand the framework we are working with.

We made the choice to put aside in Appendix A the introduction of the mathematical tools needed to understand the most advanced elements of Chapter 3. This disposition could disturb the reader by forcing him/her to go back and forth between Chapter 3 and this appendix. Yet, those tools should remains only tools, for they are not what is at stake here, and we tried to go as rapidly as possible to the point without getting too technical.

The Appendix B gives an alternate proof of a statement of Chapter 4. It is somehow more algorithmic, for two problems are proved solvable by machines that are actually described. It is worth reading for it get some insight on pointer computation, and by extension on log-space reduction.

The picture that follows presents some of the dependencies between the different parts of this work.


2 – Proof circuits
1.3 – Previous Gateways between Linear Logic and Complexity
1.1 – Computational Complexity Theory
1.2 – Linear Logic
A – Operator Algebras Preliminaries
From
3.1 – Complexity and Geometry of Interaction
to
3.4 – How to Compute
3.5 – Nilpotency Can Be Decided With Logarithmic Space
From
4.1 – JAGs and PURPLE: on the limitations of pebbles and pure pointers
to
4.2 – Finite Automata and L
4.3 – Non-Deterministic Pointer Machines
and
4.4 – Simulation of Pointer Machines by Operator Algebra
B – An Alternate Proof of co-NL ⊆ NPM

Introduction

Logic is so tightly bound to computer science that the sonship is somehow reversed: computers, through programming languages, now teach us how reasoning can be algorithmically represented. We learnt how to modelise the cost of solving a problem, answering a question mathematically presented. And yet Logic, thanks to the proof-as-program correspondence, still has a lot to say about this modelisation.

The first cost models that were developed on abstract machines discriminate between the "reasonable" models of computation and the one that are not. Of course, they could not foresee the inherent complexity of solving a precise problem: one had to actually run the program to observe the resources it needs to obtain a result. Some important problems, in a way equivalent to any other "as complex" problem, were defined and were taken as representative of the power of some complexity classes.

The Implicit Computational Complexity approach reversed this dependency, by defining frameworks where any algorithm that could be expressed was "pre-bounded". Among the approaches developed stands Linear Logic, and among the classes "captured" stands **P**, the class of "tractable" problems. This class drew a large part of the attention because of the challenge its separation from the other classes represents.

There is no reason to stick to that class, for Linear Logic has at its disposal numerous tools to simulate other ways of computing. Modalities —the salt of Linear Logic— were proven fruitful to handle the polynomial time complexity, but there are numerous other built-in mechanisms to cope with the complexity.

This work presents two innovative ways to represent non-deterministic and parallel computation with Linear Logic. They were not invented by the author, but by Terui and by Girard, respectively in 2004 and in 2012. Those approaches have some elements in common, at least a part of the tools and the aims, and they are two natural extensions of a tradition of bridging the gaps between Linear Logic and complexity. Still, they are largely unexplored, excepted for two works led by Mogbil and Rahli, and except of Seiller's PhD Thesis.

This work may be hard to get through, for it mixes three fields of research: complexity theory, Linear Logic, and a mathematical framework known as Geometry of Interaction. We tried to ease the presentation by developing some examples and by taking some time to discuss the general shape of the construction as well as some details. May this work ease the understanding of those two exciting approaches.

CHAPTER I

Historical and Technical Preliminaries

This chapter will set up most of the elements needed to understand the rest of this work. A strict minimal knowledge on set theory and graph theory is required to grasp some definitions, but apart from that everything should be quite self-explanatory. However, we do not always take the time to introduce gently the notions and go straight to the point, but many references are provided.

This work takes place in the complexity area, a subject that mix theoretical computer science and mathematics, and tries to classify computational problems by their inherent complexity. This was historically done by showing off algorithms that run on abstract machines, where every operation is endowed with a cost, and by studying the overall cost of the computation. To be pertinent, this consumption of resources is expressed *relatively to the size of the input* by a function that does not have to be precise: we ignore the constant factors and the lower order terms to focus on a larger magnitude.

Later on came the *Implicit Computational Complexity*, that describe complexity classes without explicit reference to a machine model and to bounds. It massively uses mathematical logic to define programming language tools —like type-systems – that enforce resource bounds on the programs. Among the techniques developed, one could mention

- Recursion Theory, which studies the class characterized by restricting the primitive recursion schema.[1]
- Model Theory, which characterizes complexity classes by the kind of logic needed to express the languages in them. It is also known as Descriptive complexity, and we refer to Immerman [77] for a clear and self-contained presentation.
- Proof Theory, which uses the Curry-Howard correspondence to match up execution of a program and *normalisation* of a proof.

This latter way of characterizing complexity classes by implicit means is a major accomplishment of the cross-breeding of Logic and computer science. We will in the third section gives some examples of the previous works led onto that direction, but we will first present its two components: computational complexity theory, and a special kind of logic, Linear Logic.

[1] The seminal work was made by Leivant, but one could with benefits look to a clear and proper re-statement of his results by Dal Lago, Martini, and Zorzi [34].

1.1 Computational Complexity Theory

A short and yet efficient introduction to the theory of complexity can be found in the chapters "Basic notions in computational complexity" written by Jiang, Li, and Ravikumar [82] and published in a classical handbook[6]. In the same book, the two chapters written by Allender, Loui, and Regan [1, 2], "Complexity Classes" and "Reducibility and Completeness" synthesise all the results that will be used in this section. The handbook of Papadimitriou [111] is often refereed to as "the" handbook for a nice and complete introduction to this theory: its unique style allows to read it as a novel, and to grasp the tools one after the other, but for the very same reason it is hard to look for a precise information in it. The reader searching for a precise definition or a theorem should probably rather refer to the textbooks of Greenlaw, Hoover, and Ruzzo [65], and Arora and Barak [4].

Complexity and models of computation

This section will quickly try to present what a "reasonable" model of computation is, what is at stake when we take parallel models of computation, and gives hints toward the specificities of subpolynomial classes of complexity. It borrows some elements from van Emde Boas [132], whose preliminary chapter is clear and concise.

To study complexity in an "explicit" framework, one needs a machine model and a cost model. A machine model is a class of similar structures, described set-theoretically as tuples most of the time, endowed with a program, or finite control, and a definition of how *computation* is actually performed with a machine of that model. The cost model fixes the cost of the basic operations, allowing to measure the resources (most of the time understood as time and space) needed to process an input.

Two fundamentals of this theory is that something is "algorithmically computable" iff it is computable by a Turing Machine (Church–Turing thesis), and that taking the Turing Machine or another "reasonable" model of computation does not really matter, this is the invariance thesis:

Invariance Thesis. *"Reasonable" machines can simulate each other within a polynomially bounded overhead in time and a constant-factor overhead in space.*

Of course, what is a simulation needs to be define.

Definition 1.1.1 (Simulation)**.** Given two machines models M and M', we say that M' simulates M with time (resp. space) overhead $f(n)$ if there exists two functions c and s such that, for every machine M_i in M:

- for all input x of M, $c(x)$ is the translation of x in the setting of M';
- $M_i(x)$ halts iff $M'_{s(i)}(c(x))$ halts, and $M_i(x)$ accepts iff $M'_{s(i)}(c(x))$ accepts[2] ;
- if $t(|x|)$ is the time (space) bound needed by M_i for processing x, then the time (resp. space) required by $M'_{s(i)}$ for processing $c(x)$ is bounded by $f(t(|x|))$;

If the time overhead function f is linear, the simulation is said to be *real-time*.

[2] Where $M_i(x)$ denotes the processing of x by M_i.

Something is missing in this "definition": what are the resources needed to compute s and c? It would be meaningless to provide more computational power to compute $M'_{s(i)}$ than what $M'_{s(i)}$ can actually provide: we could as well have computed the output directly! In other words, if the machine used to process c and s is strictly more powerful than $M'_{s(i)}$, we cannot extract any knowledge about M'. In the following, we need to be careful about the cost of those simulations.

We will tend to keep the word simulation for the case where c and s are computed by a machine of M' itself: in other word, provided a description of M_i, the machine M'_i will by itself "acts as M_i", without external intervention. If a third machine model M'', given the description of M_i and x, outputs the description of $M'_{s(i)}$ and $c(x)$, we will speak of a *translation*.

Notice that f takes as input $|x|$ the size of the input, for we measure the resources needed by the computation *relatively to the size of the input*. This function should be *proper*,[3] i.e. it should respect some basic properties, but we won't take that parameter into account, for we will use only the proper functions below:

Polynomial i.e. there exists $c, d \in \mathbb{N}^*$ such that $f(|x|)$ is bounded by $c \times (|x|^d)$.

Constant i.e. there exists $c \in \mathbb{N}^*$ such that $f(|x|)$ is bounded by $c \times |x|$.

(Binary) Logarithm i.e. there exists $c \in \mathbb{N}^*$ such that $f(|x|)$ is bounded by $c \times (\log(|x|))$.

Poly-logarithmic i.e. there exists $c, i \in \mathbb{N}^*$ such that $f(|x|)$ is bounded by $c \times (\log^i(|x|))$.

We will use the "big O" asymptotic notation, i.e. we write that $g = \mathcal{O}(f)$ iff $\exists k \in \mathbb{N}^*$ such that $\forall x, g(x) \leqslant k \times f(x)$. So if f is a polynomial (resp. the constant function, the logarithm, a poly-logarithm), we will sometimes write $n^{\mathcal{O}(1)}$ (reps. $\mathcal{O}(1)$, $\mathcal{O}(\log)$, $\mathcal{O}(\log^i)$) or even more simply poly (resp. 1, log, $\log^i$), without any reference to the size of the input.

This notation is a bit counter-intuitive, because despite the equal sign, it is not symmetric: $\mathcal{O}(x) = \mathcal{O}(x^2)$ is true but $\mathcal{O}(x^2) = \mathcal{O}(x)$ is not. As Sipser [124, p. 277] writes it, "[t]he big-O interacts with logarithms in a particular way": the changing of the base of the logarithm affects the complexity only by a constant factor, so we may simply ignore it and we will consider that the logarithmic function is always in base 2. We will use in the course of this work some classical identities, but maybe it is worth remarking that as $n = 2^{\log(n)}$, for all $d \in \mathbb{N}$, $n^d = 2^{d\log(n)}$ and so $2^{(\log(n))^{\mathcal{O}(1)}} = n^{\mathcal{O}(1)}$. We can also remark that as $\log(n^d) = d\log(n)$, $\log(n^{\mathcal{O}(1)}) = \mathcal{O}(\log(n))$. This will be more or less the only two equalities we are going to use.

We will focus here on the subpolynomial world, that is to say the machines that use less resources than polynomial time. This world is less studied than the polynomial word, as the "reasonable" computation is quite limited in it, but a fruitful approach often used is to look toward parallel computation. This world also have a kind of invariance thesis to define what is a reasonable model:

Parallel Computation Thesis. *Whatever can be solved in polynomially bounded space on a reasonable sequential machine model can be solved in polynomially bounded time on a reasonable parallel machine, and vice versa.*

This thesis was first formulated by Goldschlager [64], and by Chandra, Kozen, and Stockmeyer [24] quite simultaneously. The first one stated that "Time-bounded parallel machines are polynomially related to space-bounded computers" by trying to "capture mathematically this intuitive

[3] For a precise definition, the reader should for instance refer to Papadimitriou [111, p. 140, definition 7.1.].

notion of parallel computation, much in the same spirit that 'Church's thesis' captures the notion of effective computation." Chandra, Kozen, and Stockmeyer, on their way, defined *Alternating Turing Machine* and proved that "an alternating Turing machine is, to within a polynomial, among the most powerful types of parallel machines".

Alternating Turing Machines are also the most general version of the most well-known model, Turing Machines, so it is quite natural to take them as the reference, as the basic model. We should make one last remark concerning this parallel world: time is sometimes refereed to as the *depth*, i.e. the maximal number of intermediate steps needed ; and space is taken to be the size, i.e. the number of processors needed to perform the computation. This will be the case with the Boolean circuits and the proof circuits, defined in Chapter 2.

Alternating Turing Machine

This model of computation is really handy, for it will allow us to define at the same time parallel and sequential computing, and it has a characterization of every complexity class we will use in the following. The following definition borrows some elements of the clear presentation of Vollmer [135, p. 53].

Definition 1.1.2 (ATM, Chandra, Kozen, and Stockmeyer [24])**.** An *Alternating Turing Machine* (ATM) M with $k \in \mathbb{N}^*$ tapes is a 5-tuple

$$M = (Q, \Sigma, \delta, \mathbf{q}_0, g)$$

with

- Q a finite set of *states*,
- Σ an alphabet,[4]
- $\delta : (Q \times \Sigma^{k+1} \times \mathbb{N}^{k+1}) \to \mathcal{P}_{\text{fin}}(Q \times \Sigma^k \times \{-1, 0, +1\}^{k+1})$ the *transition relation*,
- $\mathbf{q}_0 \in Q$ the *initial state*,
- and $g : Q \to \{\forall, \exists, 0, 1\}$ the *state type function*.

This definition does not explain how one can actually perform a computation with ATMs, and we will quickly expose those —very classical— mechanism right after we introduce some nomenclature.

The input is written on the *input tape*, and the head that scans it is read-only. The other k heads are read-write, and each one of them work on a dedicated tape. A *configuration* α of M is an element of $Q \times (\Sigma^*)^k \times \mathbb{N}^{k+1}$ that entirely describe the current situation. It reflects the current state, the content of the k working tapes and the position of the $k+1$ heads, as a "snapshot" from where it is possible to resume the computation, provided that we know the input and the transition relation.

The initial configuration α_0 of M is when the entry is written on the input tape, the k working tapes are blank,[5] the $k+1$ heads are on the first cell of their tape, and the state is $\mathbf{q}_0$.

A state $\mathbf{q} \in Q$ —and by extension a configuration whose state is $\mathbf{q}$— is said to be *universal* (resp. *existential*, *rejecting*, *accepting*) if $g(\mathbf{q}) = \forall$ (resp. $\exists$, 0, 1). A state $\mathbf{q} \in Q$ —and a configuration whose state is $\mathbf{q}$— is *final* if $\forall a_0, \ldots, a_k \in \Sigma$, $\delta(\mathbf{q}, a_0, \ldots, a_k) = \emptyset$. This condition is equivalent to $g(\mathbf{q}) \in \{0, 1\}$.

[4] One should precise that every tape is divided into *cells* and that every *letter* of the alphabet need exactly one cell to be written.
[5] Usually a special "blank" symbol $\flat$ is written on every cell, but it works as well to take any symbol of the alphabet.

Suppose that M is working on input $x \equiv x_1 \dots x_n$ (we denote this situation with $M(x)$) and that at a certain point, M is in configuration

$$\alpha = (\mathbf{q}, w_1, \dots, w_k, n_0, \dots, n_k)$$

where $w_i \equiv w_{i,1} \dots w_{i,l_i}$ for $1 \leqslant i \leqslant k$. Suppose that $\delta(\mathbf{q}, x_{n_0}, w_{1,n_1}, \dots, w_{k,n_k})$ contains (possibly among others) the tuple $(\mathbf{q'}, a_1, \dots, a_k, X_0, \dots, X_k)$, where $a_1, \dots, a_k \in \Sigma$ and $X_0, \dots, X_k \in \{-1, 0, +1\}$. Then

$$\beta = (\mathbf{q'}, w'_1, \dots, w'_k, n'_0, \dots, n'_k)$$

is a *successor configuration* of α, where for $1 \leqslant i \leqslant k$, w'_i is obtained from w_i as follow: replace the n_ith letter by a_i, and to cover the case where the head moves to the left or right out of the word by prefixing or suffixing the blank symbol $\flat$; and for $0 \leqslant i \leqslant k$, let $n'_i = n_i + X_i$, if $n_i + X_i \geqslant 0$, otherwise we leave $n'_i = n_i$.

That β is such a successor configuration of α (and respectively that α is a *predecessor* of β) is denoted by $\alpha \vdash_{M(x)} \beta$, or if M and x are clear from the context by $\alpha \vdash \beta$. As usual, $\vdash^*$ is the reflexive transitive closure of $\vdash$. Observe that a final configuration has no successor and that the initial configuration has no predecessor.

A *computation path* of length n in $M(x)$ is a string $\alpha_1, \dots, \alpha_n$ such that $\alpha_1 \vdash_{M(x)} \dots \vdash_{M(x)} \alpha_n$. We say that $M(x)$ has an *infinite computation path* if for every $n \in \mathbb{N}$ there exists a computation path of length n. A computation path $\alpha_1, \dots, \alpha_n$ is *cyclic* if $\alpha_1 = \alpha_n$.

We should be more rigorous and define the ATM to be a 6-tuple with k —the number of tapes— to be a parameter. But thanks to some classical theorems, we know that the number of tapes does not really matter, even if we have complexity concern: an Alternating Turing Machine with k tapes can reasonably be simulated with an Alternating Turing Machine with 3 tapes, one for the entry, one working tape, and one tape to write down the output.

In fact, we won't even bother taking into account the output tape in most of the case, for our Alternating Turing Machines will only *decide*, i.e. accept or reject an input, as the following definition explains.

Definition 1.1.3 (Accepted word)**.** We define a labelling l from the configurations of $M(x)$ to $\{0, 1\}$: a configuration α of $M(x)$ is labelled with 1 iff

- $g(\alpha) = 1$, or
- $g(\alpha) = \forall$ and for every configuration β such that $\alpha \vdash \beta$, $l(\beta) = 1$, or
- $g(\alpha) = \exists$ and there exists a configuration β such that $\alpha \vdash \beta$ and $l(\beta) = 1$.

We can recursively label every configuration of $M(x)$ with l. As soon as the initial configuration of M has been labelled, we can stop the procedure and leave some configuration unlabelled.

A word $x \in \Sigma^*$ is said to be *accepted by* M if the initial configuration of $M(x)$ is labelled with 1. We set $\mathcal{L}(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}$.

Once the computation is done,[6] that is to say when the initial configuration may be labelled, we can know what are the resources consumed by the processing of the input. We define three costs models on our ATMs.

[6] This is precisely what differentiate implicit and explicit computational complexity. Here, we have to perform the computation to know the resources needed to solve a problem.

Definition 1.1.4 (Bounds on ATM)**.** Given M an ATM and s, t and a three functions, we say that M is space- (resp. time-, alternation-) bounded by s (resp. t, a) if for every input x of size $|x|$,

at every moment, at most $\mathcal{O}(s(|x|))$ cells of the working tapes are left non-blank,[7]

every computation path is of length at most $\mathcal{O}(t(|x|))$,

in every computation path, the number of times an existential configuration has for successor an universal configuration plus the number of times an universal configuration has for successor an existential configuration is at most $\mathcal{O}(a(|x|))$.

Note that if M is space-bounded by s then given an input x the number of configurations of M is at most $2^{\mathcal{O}(s(|x|))}$: $M(x)$ can have $\mathrm{Card}(Q)$ different states, the working tapes can contains $k \times \mathrm{Card}(\Sigma)^{s(|x|)}$ different words of length at most $s(|x|)$, the head on the input tape can be in $|x|$ different places and the k heads can be in $k \times s(|x|)$ different positions. To sum up, $M(x)$ can only have $\mathrm{Card}(Q) \times (k \times \mathrm{Card}(\Sigma)^{s(|x|)}) \times (|x|) \times (k \times s(|x|))$ different configurations. As all those value except $|x|$ are given with M, so we know there exists a d such that $M(x)$ has less that $2^{d \times s(|x|)}$ different configurations.

Moreover, if M is bounded in time by t, then M is bounded in alternation by t, for M cannot "alternate more than it computes".

If all the configurations are existential (resp. universal), we say that M is a *non-deterministic* (resp. *universally non-deterministic*) Turing Machine. If the transition relation δ is functional, M is said to be a *deterministic* Turing Machine.

The rest of this subsection will define some other important notions like problems, reductions or complexity classes. We should dwell however on the links between Alternating Turing Machines, graphs and parallel complexity. It will be proven very useful in the rest of this thesis.

Definition 1.1.5 (Computation graph, inspired by Vollmer [135, p. 54])**.** Given an ATM M and an input $x \equiv x_1 \dots x_n$, a *computation graph* $G_{M(x)}$ is a finite acyclic directed graph whose nodes are configurations of M, built as follow:

- Its roots is the initial configuration of M,
- if $\alpha \in G_{M(x)}$ and $\alpha \vdash \beta$, then $\beta \in G_{M(x)}$ and there is an edge from α to β,

A node in $G_{M(x)}$ is a *sink* if the associated configuration is final.

The computation graph associated to M on input x, $G_{M(x)}$, is of depth[8] $\mathcal{O}(t(|x|))$ if M is bounded in time by t. If M is bounded in space by s, the number of configurations of M is bounded by $2^{\mathcal{O}(s(|x|))}$ and so is the number of nodes of $G_{M(x)}$. Note also that if M is deterministic, $G_{M(x)}$ is a linear graph.

To get a better insight on this construction, one may refer to Figure 1.1 where the example of a computation graph is sketched.

To determine from a computation graph if the course of computation described leads to acceptance of rejection, we simply have to label every node following the procedure described in Definition 1.1.3.

[7] The entry tape does not count in the measure of the space needed to perform a computation. We dwell on that point for it would make no sense to speak of a computation using a logarithmic amount of space in the size of the input elsewhere: one would not event have the space to write down the input! This is one of the "log-space computation oddities", more are to come.

[8] The depth of $G_{M(x)}$ is the maximal number of edges between the node associated to the initial configuration and any other node.

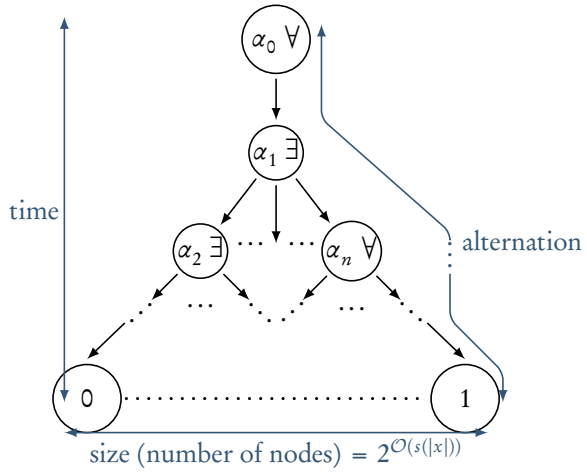


Figure 1.1: The computation graph of an ATM

In Section 2.4, we will define ATMs in input normal forms (Definition 2.4.1) and use this crucial notion to define family of computation graphs (Definition 2.4.2).

This definition as the others use some classical theorems without stating them. For instance, the fact that a computational graph is *finite* and *acyclic* comes from two reasonable modifications that one can perform on any ATM. We can always add a clock to our ATMs, and forgive any transition such that $\alpha \vdash \beta \vdash^* \alpha$, i.e. to prevent any loop in our program. Notice also that we did not bother to define negative states for our ATMs, for it was proven in the very first paper on this topic [24, Theorem 2.5, p. 120] that those states did not bring any computational advantage.

We will also use the following classical modifications: we can always take the alphabet Σ to be $\{0,1\}$ and the number of working tapes k to be 1. This has some minor consequences on the complexity of our object, but stay within the realm of "reasonable" modifications, i.e. does not modify the complexity classes characterized.

Problems and complexity classes

Definition 1.1.6 ((Decision) Problem)**.** A problem P is a subset of $\mathbb{N}$. If there exists M an ATM, such that for all $x \in \mathbb{N}$, $M(x)$ accepts iff $x \in P$, we say that M *decides* P. We will sometimes say in this case that P is the *language* of M, i.e. we define $\mathcal{L}(M) = \{n \in \mathbb{N} \mid M(x) \text{ accepts}\}$.

Definition 1.1.7 (**STA**$(*,*,*)$[9])**.** Given s, t and a three functions, we define the *complexity class* **STA**$(\mathbf{s},\mathbf{t},\mathbf{a})$ to be the set of problems P such that for every of them there exists an ATM bounded in space by s, in time by t and in alternation by a that decides it.

If one of the function is not specified, there is no bound on that parameter and we write $*$, and if —in the case of alternation— we allow only ∃ (resp. ∀), we write ∃ (resp. ∀) in the corresponding field. If M is deterministic, we write **d** in the alternation field.

Decisions problems are not sufficient to define everything we need, for instance because the simulation of the Invariance Thesis needs to process functions. Roughly, we say that a function f is

[9]This acronym is of course to be read as *Space, Time, Alternation*.

computable in **C** if a machine with corresponding bounds can for all $x \in \mathbb{N}$ write on a dedicated tape ("the output tape") $f(x)$. Note that the space needed to write down this output may not be taken into account when measuring space bound, for we can consider it is a write-only tape. This is one of the reasons composition is so tricky in the log-space bounded word: if we compose two log-space bounded machines M and M', the output-tape of M becomes one of the working tape of $M \circ M'$, and so $M \circ M'$ does not respect any more the log-space bound.

This also force us to trick the classical definition of "reductions", that are the equivalent of simulation between machine models when applied to problems:

Definition 1.1.8 (Reduction)**.** Given two problems P and P', we say that P reduces to P' with a **C**-reduction and we write $P \leqslant_{\mathbf{C}} P'$ iff there exists a function f computable in **C** such that for all x, $x \in P$ iff $f(x) \in P'$.

There is a whole galaxy of papers studying what a reduction exactly is, for one should be really careful about the resources needed to process it. We refer to any classical textbook for an introduction to this notion, but we define anyhow log-space reduction for it will be useful.

Definition 1.1.9 (log-space reduction)**.** A function $f : \{0,1\}^* \to \{0,1\}^*$ is[10] said to be *implicitly* log*-space computable* if there exists a deterministic Turing Machine with a log-space bound that accepts $< x, i >$ iff the ith bit of $f(x)$ is 1, and that rejects elsewhere.

A problem P is log*-space reducible* to a problem P' if there exists a function f that is implicitly log-space computable and $x \in P$ iff $f(x) \in P'$.

Reduction should always be defined accordingly to the resources consumed, but in the following we will always suppose that the "handy" reduction is taken, i.e. the one that uses the less (or supposedly less) computational power. An other crucial notion for complexity theory is the notion of *complete problem of a complexity class*: this notion somehow captures a "representative problem", it helps to get an intuitive understanding of what a complexity class is capable of.

Definition 1.1.10 (**C**-complete problem)**.** A problem P is **C***-complete (under a* $\mathbf{C}'$*-reduction)* if

- $P \in \mathbf{C}$
- For all $P' \in \mathbf{C}$, $P' \leqslant_{\mathbf{C}'} P$ with $\mathbf{C}' \subseteq \mathbf{C}$.

If P entails only the second condition, P is said to be **C***-hard*.

We can now define some classical complexity classes, first the *sequential* complexity classes, the one that can be defined without really taking profit of alternation:

Definition 1.1.11 (Complexity classes)**.**

$$\mathbf{L} = \mathbf{STA}(\log, *, \mathbf{d}) \tag{1.1}$$

$$\mathbf{NL} = \mathbf{STA}(\log, *, \exists) \tag{1.2}$$

$$\mathbf{co\text{-}NL} = \mathbf{STA}(\log, *, \forall) \tag{1.3}$$

$$\mathbf{P} = \mathbf{STA}(\log, *, *) = \mathbf{STA}(*, \mathrm{poly}, \mathbf{d}) \tag{1.4}$$

[10] Some hypothesis are required, we refer for instance to [4, Chapter 4: Space complexity] for a precise statement.

We introduced here the *complementary* of a complexity class, namely **co-NL**. Let P be a problem, *PComp* is $\{n \in \mathbb{N} \mid n \notin P\}$. For any complexity class **C**, **co-C** denotes the class $\{PComp \mid P \in \mathbf{C}\}$. If a problem $P \in \mathbf{C}$ can be decided with a deterministic machine, it is trivial that *PComp* can be decided with the same resources, it is sufficient to inverse acceptance and rejection, and so $\mathbf{C} = \mathbf{co\text{-}C}$.

We chose here to *define* the classical parallel classes of complexity **AC** and **NC** in terms of Alternating Turing Machines, the other definition in terms of *Boolean circuits* being postponed to Definition 2.1.5. The correspondence between those two definitions is a mix of classical results [15, p. 14] (eq. (1.6)), [25], [123], [115] (eq. (1.8)).

Definition 1.1.12 (Alternation and parallel complexity classes)**.** For all $i \geqslant 1$

$$\mathbf{AC^0} = \mathbf{STA}(*, \log, \mathbf{1}) \tag{1.5}$$
$$\mathbf{NC^1} = \mathbf{STA}(*, \log, *) \tag{1.6}$$
$$\mathbf{AC^i} = \mathbf{STA}(\log, *, \log^{\mathbf{i}}) \tag{1.7}$$
$$\mathbf{NC^i} = \mathbf{STA}(\log, \log^{\mathbf{i}}, *) \tag{1.8}$$

We define $\cup_i \mathbf{AC^i} = \mathbf{AC}$ and $\cup_i \mathbf{NC^i} = \mathbf{NC}$.

Theorem 1.1.1 (Hierarchy)**.**

$$\mathbf{AC^0} \subsetneq \mathbf{NC^1} \subseteq \mathbf{L} \subseteq \mathbf{NL} = \mathbf{co\text{-}NL} \subseteq \mathbf{AC^1} \subseteq \mathbf{NC^2} \subseteq \ldots \subseteq \mathbf{AC} = \mathbf{NC} \subseteq \mathbf{P}$$

For all $i \in \mathbb{N}$, $\mathbf{AC^i} \subseteq \mathbf{NC^{i+1}}$ is quite evident, for a bound of $\log^{i+1}$ on time is sufficient to perform at least $\log^i$ alternations. That $\mathbf{AC^0} \subsetneq \mathbf{NC^1}$ is a major result of Furst, Saxe, and Sipser [44] and made the Boolean circuits gain a lot of attractiveness, for separation results are quite rare!

The proof of $\mathbf{NL} \subseteq \mathbf{NC^2}$ is rather nice, and one may easily find a clear explanation of it [135, p. 41, Theorem 2.9]: a matrix whose dimension is the number of configurations of a log-space Turing Machine is built in constant depth,[11] and a circuit of depth $\log^2$ computes its transitive closure in parallel.

At last, $\mathbf{co\text{-}NL} = \mathbf{NL}$ is a major result of Immerman [78]. It was proven by showing how a log-space bounded Turing Machine could build the computation graph of another log-space bounded Turing Machine and parse it to determine if there is a path from the initial configuration to an accepting configuration. That $\mathbf{co\text{-}L} = \mathbf{L}$ is trivial, since every deterministic complexity class is closed by complementation.

Turing Machines indeed share a lot with graphs, especially when they are log-space bounded. We illustrate that point with two (decision) problems that we will use later on. They both are linked to connectivity in a graph, and can be also refereed to as "reachability" or "maze-problem". Numerous papers are devoted to list the problems complete for a class, we can cite among them Jenner, Lange, and McKenzie [81] and Jones, Lien, and Laaser [85].

Problem A: *Undirected Source-Target Connectivity* (*USTConn*)

`Input`: An undirected graph $G = (V, E)$ and two vertices $s, t \in E$
`Output`: Yes if s and t in the same connected component in G, i.e. if there is a path from s to t.

[11] That is to say with a family of constant-depth unbounded Boolean circuit. It might just be understand for the time being as "with $\mathbf{AC^0}$ resources", a more precise definition will be given in Chapter 2.

Reingold [114] proved that *USTConn* is **L**-complete (hardness was proved by Cook and McKenzie [28]). We will use in Chapter 2 its bounded variant, where G is of degree at most n, $USTConn_2$. We will also use in Appendix B a small variant of its complementary, where s and t are not parameters, but always the first and the last nodes of V.

This problem also exists in a directed fashion:

Problem B: *(Directed) Source-Target Connectivity* (*STConn*)

`Input:` A directed graph $G = (V, A)$ and two arcs $s, t \in A$
`Output:` Yes if there a path from s to t in G.

STConn is **NL**-complete.

The proof that every **NL**-problem can be reduced to *STConn* is really interesting, for it links closely log-space computation and graphs.[12] Let $P \in \mathbf{NL}$, there exists non-deterministic log-space bounded Turing Machine M that decides it (otherwise there would be no evidence that $P \in \mathbf{NL}$). There exists a constant-depth boolean circuit that given $n \in \mathbb{N}$ and the description of M outputs $G_{M(n)}$ the configuration graph of M. Then $n \in P$ iff there is a path between the initial configuration and a configuration accepting in $G_{M(n)}$, a problem that *is STConn*.

1.2 Linear Logic

Linear Logic (LL) was invented by Girard[47] as a refinement of classical logic. Indeed, (classical) logic is a pertinent tool to formalise "the science of mathematical reasoning", but when we affirm that a property A entails a property B ($A \Rightarrow B$), we do not worry about *the number of times A is needed to prove B*. In other words, there is no need to take care of how resources are handled, since we are working with ideas that may be reproduced for free.

We just saw that if we have complexity concerns, one should pay attention to duplication and erasure, for it has a cost. The implication $\Rightarrow$ superimposes in fact two operations that Linear Logic distinguishes: first, there is the *linear implication* $A \multimap B$ that has the meaning of "A has to be consumed once to produce B", and the *of course modality* $!A$ that has the meaning of "I can have as many copies (including 0) of A as I want". Thus, the classical implication $A \Rightarrow B$ is encoded as $(!A) \multimap B$. From this first remark, numerous tracks were followed: we will begin by recalling the main ideas of Linear Logic, and then focus on proof nets before presenting the Geometry of Interaction (GoI) program.

The reference to study Linear Logic remains *Proofs and Types*[62], and the reader not used to this subject could read with benefits the 10 pages of Appendix B, "What is Linear Logic?", written by Lafont. For anyone who reads french, Joinet's *Habilitation à diriger les recherches*[83] is an excellent starting point to put Linear Logic into philosophical and historical perspective. The two volumes of Girard's textbook [55, 56] were translated in English[58] and are the most complete technical introduction to the subject.

Main concepts and rules

We will tend during the course of this work to dwell on the syntactical properties of Linear Logic, but we should however remark that some of its elements appeared first in Lambek [94]'s linguistic considerations. Linear Logic is a world where we cannot duplicate, contract and erase occurrences

[12] For a more advanced study of those links, one could have a look at the work led by Wigderson [136] and to its important bibliography.

$$\frac{}{\vdash A, A^\perp}\ ax. \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\ \otimes \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\ der. \qquad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}\ !$$

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\ cut \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}\ ? \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\ ctr.$$

From left to right, the names and the group of the rules are

Group	**Structural**	**Multiplicative**	**Exponential**	
1st line	Axiom	Tensor introduction	Dereliction	Promotion
2nd line	Cut	Par introduction	Weakening	Contraction

Figure 1.2: The rules of MELL

of formula for free. This led to the division of classical logic[13] into three *fragments*: additive, whose rules are context-sensitive, multiplicative, whose rules are reversible, and exponential, which handles the modalities. The negation of a formula is encoded thanks to the *linear negation* $^\perp$, which is an involution.

We will only use the MELL fragment (without units),[14] i.e. the multiplicative and exponential connectives, whose formula are, for α a *literal* (or *atom*, a variable taken for a countable set), using the inductive *Backus Normal Form* notation:

$$A \coloneqq \alpha \mid \alpha^\perp \mid A \otimes A \mid A \parr A \mid !A \mid ?A.$$

The $\otimes$ connective is called *tensor*, the $\parr$ connective is called *par*, ! and ? are respectively named *of course* (or *bang*) and *why not*. Dualities are defined with respect to De Morgan's laws, that allows us to let the negation flows down to the literals:

$$(A \otimes B)^\perp \coloneqq A^\perp \parr B^\perp \qquad (A \parr B)^\perp \coloneqq A^\perp \otimes B^\perp \qquad (!A)^\perp \coloneqq ?(A^\perp) \qquad (\alpha^\perp)^\perp \coloneqq \alpha$$

The connective $\multimap$ is defined as $A \multimap B \equiv (A^\perp) \parr B$. We let Γ and Δ be two multisets of formulae, and for $\diamond \in \{!, ?\}$, if $\Delta = A_1, \dots A_n$, we let $\diamond\Delta$ be $\diamond A_1, \dots, \diamond A_n$.

We borrow from Mogbil [106, p. 8] the following presentation of rules and proofs, which is especially handy. A *sequent* $\vdash \Gamma$ is a multiset of formulae of MELL, the order of the occurrences of formula plays no role for the time being. We let S be the set of sequents of MELL. For $n \geqslant 0$, a n-ary *rule* R_n with n hypothesis and one conclusion is a $n+1$ relation on S, i.e. $R \subseteq S^n \times S$. The set of rules of MELL is given in Figure 1.2, where the *principal formula* is distinguished in the conclusion sequent.

A *proof of height* 0 in MELL is the unary rule *ax.*. For $n \neq 0$, a *proof of height* $h \neq 0$ D_h is a subset of $(D_{<h})^n \times R_n$ such that

- the set $D_{<h}$ is the set of proofs of height strictly inferior to h,
- the n conclusions of the n proofs of $D_{<h}$ are the n premises of R_n,
- the conclusion of D_h is the conclusion of R_n.

[13] For a complete presentation of the variations of Linear, Classical and Intuistionistic Logic, we refer for instance to Laurent [95].

[14] In a nutshell, it is natural to associate to the multiplicative fragment a parallel handling of the resources, whereas the additive fragment is rather to be conceived as highly non-deterministic.

A sequent $\vdash \Gamma$ is derivable in MELL if it admits a proof, i.e. if there exists a proof π whose conclusion is $\vdash \Gamma$. The *η-expansion* of a proof π is a proof π' of same conclusion where the axioms introduce only literals. Such a proof π' always exists, and it is unique. A proof π is α-equivalent[15] to a proof π' if π' can be obtained from π by replacing the occurrences of the literals in π.

One of the fundamental theorem of logic is *Gentzen's Hauptsatz*, or cut-elimination theorem, proved first by Gentzen [45][16] for classical and intuitionistic logic. This theorem applies as well to MELL, and we sometimes say that a system *eliminates cuts* if it entails this theorem:

Cut-elimination Theorem. *For every proof π of $\vdash \Gamma$ in MELL, there exists a proof π' of same conclusion that does not uses the rule* cut.

The Curry-Howard-correspondence (*a.k.a.* the *proof-as-program* correspondence) tells us that the operation of rewriting a proof with cuts into a proof without cuts corresponds to the execution of a program written in λ-calculus. This link between computer programs and mathematical proofs justifies the use of Linear Logic as a typing system on programs, and by extension, its use as a tool to modelise complexity. Before studying in depth how this gateway was used, we still need to introduce two other key-concepts of Linear Logic: proof nets and Geometry of Interaction (GoI). They correspond respectively to the model we develop in Chapter 2, and to the model we develop in Chapter 3.

Proof Nets

Proof Nets are a parallel syntax to express proofs that get rid of some lumbering order in the application of rules. For instance, one should not feel the urge to distinguish between the following two proofs obtained from the same proof π:

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi}{\vdash A,B,C,D}}{\vdash A \parr B, C, D}\parr}{\vdash A \parr B, C \parr D}\parr \qquad\qquad \dfrac{\dfrac{\dfrac{\vdots\,\pi}{\vdash A,B,C,D}}{\vdash A, B, C \parr D}\parr}{\vdash A \parr B, C \parr D}\parr$$

It was also introduced by Girard[51] and provoked numerous studies, and yet there is to our knowledge no "classical definition". The most accurate one can be found in Mogbil's *Habilitation à diriger les recherches* [106, Definition 2.11, p. 11], it defines proof nets as hyper-multigraphs.

We will present in details and massively use proof nets for an unbounded version of Multiplicative Linear Logic in Chapter 2. So despite the tradition to present some drawing to introduce them, we will just express some global ideas and refer the reader to Section 2.1 for a complete presentation. We will use them only as examples and hints toward a sharper understanding in Chapter 3, and they do not appear in Chapter 4.

A proof net is built inductively by associating to every rule of Figure 1.2 a rule on drawings. We obtain by doing so a labeleld graph, where the nodes are endowed with a connective of MELL, and the edges represents sonships between occurrences of formulae. This parallel writing of the proofs gives a quotient on proofs that "reduces the syntactical noise" and by doing so let the proofs be more readable. Every derivation infers a proof net, and by applying some quotient on the different representations of the same proof net, a proof net may be inferred by several proofs.

[15] We won't define this notion precisely here, for it is very tedious to do so, as attested by the 1st chapter of Krivine [92].
[16] The interested reader may find a translation [46].

One should remark that none of those rules is local in this setting, for there is always (except for the axiom rule) a constraint on the membership of the principal occurrences to the same context or to two different contexts, not to state the constraint on the presence of a modality. This is the drawback of this representation: the correctness of a proof is not tested locally any more, as opposed to any other syntactical system that existed in the history. We call *pseudo nets* the structures freely-generated, that is, such that we don't know if there exists a proof that infers it. To determine whether a pseudo net is a proof net, we need a global criteria to test its "legacy". Those criterion are called criterion of correctness and we refer once again to Mogbil's work [106, p. 16] for a complete list. We will use one of them in Theorem 2.2.1.

We left aside the presentation of those (yet very important) tools to focus on a crucial point for the well understanding of this thesis. As we presented it, the sequents of MLL are not ordered. An order could be introduced thanks to a bit tedious notion of *exchanges* of occurrences of formulae, allowing us to keep track of their positions. It is handy not to take care of this aspect, but we are forced to re-introduce it when dealing with proof nets: suppose given a proof π of conclusion $\vdash ?A, ?A, ?A, B, B$. The proof π can be completed as follows:

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi}{\vdash ?A, ?A, ?A, B, B}}{\vdash ?A, ?A, B, B}\,ctr.}{\vdash ?A, ?A, B \parr B}\,\parr$$

The question is: if we label the conclusion of π with $\vdash ?A_1, ?A_2, ?A_3, B_1, B_2$, what is the conclusion of the previous proof? It could be any sequent among

$$\begin{array}{lll} \vdash ?A_1, ?A_2, B_1 \parr B_2 & \vdash ?A_1, ?A_3, B_1 \parr B_2 & \vdash ?A_2, ?A_3, B_1 \parr B_2 \\ \vdash ?A_1, ?A_2, B_2 \parr B_1 & \vdash ?A_1, ?A_3, B_2 \parr B_1 & \vdash ?A_2, ?A_3, B_2 \parr B_1 \end{array}$$

In fact, by dealing with non-ordered sequents, we already introduced a first quotient on our proofs, which is not compatible with proof nets. Remark that it does not contradict the fact that proof nets quotient proofs, for we may for this example performs the contractions and the par-introduction in parallel.

But in the following, we will encode the truth-value of a boolean type in the order of the the premise of a par-rule (Definition 2.2.1). The principal formulae of the contractions will later on encode the value of a bit in a string (Section 3.3). This distinction, which by the way appears in Geometry of Interaction, is crucial since it may be the only difference between the proofs corresponding to two different binary lists.

Getting closer to a Geometry of Interaction

Geometry of Interaction (GoI) tended to be used as a motto to qualify several approaches to reconstruct logic. As Joinet [83, p. 43] points it, the heading "Geometry of Interaction " is used at the same time to describe

- a way to rebuild logic from computational interaction,
- a formalization of the preceding idea in a functional analysis framework.

Their common point is a strong will to "rebuild logic from the inside", by letting the cut-elimination —and by extension the dynamic— be the the primary notion from which logic is built, and hence to obtain immediately computational properties. This subsection does not recollect all the previous attempts to work in the GoI setting, but gives some hints of its goal, its links with complexity being postponed to the next section and to Section 3.4.

The name GoI was first introduced by Girard [49] and it was accompanied with a clear justification that we cite entirely:

> "*The only extant semantics for computation are denotational, i.e. static. This is the case for the original semantics of Scott [118], which dates back to 1969, and this remains true for the more recent* coherent semantics *of the author [47]. These semantics interpret proofs as functions, instead of actions. But computation is a dynamic process, analogous to —say— mechanics. The denotational approach to computation is to computer science what statics is to mechanics: a small part of the subject, but a relevant one. The fact that denotational semantics is kept constant during a computational process should be compared to the existence of static invariants like mass in classical mechanics. But the core of mechanics is dynamics, where other invariants of a dynamical nature, like energy, impulsion etc. play a prominent role. Trying to modelize programs as actions is therefore trying to fill the most obvious gap in the theory. There is no appropriate extant name for what we are aiming at: the name 'operational semantics' has been already widely used to speak of step-by step paraphrases of computational processes, while we are clearly aiming at a less* ad hoc *description. This is why we propose the name*
>
> ***geometry of interaction***
>
> *for such a thing.*"
>
> (Girard [49, p. 26])

Starting from the consideration that "linear negation behaves like transposition in Linear Algebra" [49, p. 10], a community started to work on the building of the Logic in a Linear Algebra framework. One leading idea was to define *ex nihilo* the concept of type, and to replace cut-elimination by the action of some operators on the Hilbert space. Proofs were interpreted as symmetrical matrices, i.e. equal to their own transpositions, and they were, so to say, "the written trace of underlying geometrical structures" [49, p. 1]. What was really interesting was the ability to express strong normalisation as nilpotency thanks to the "Execution formula". GoI was naturally conceived as a "new form of semantics" and expressed within a C^*-algebra.

Yet, coherent and Banach spaces had their limitations:

> "quantum coherent spaces *[...] have [...] two major drawbacks: that of a categorical interpretation, whence unable to explain dynamics, thus complexity; and, at a deeper level, their incompatibility with infinite dimension, the latter drawback being related to the former.*"
>
> (Girard [57, p. 26])

So the von Neumann algebra framework was developed, but we save this part for Chapter 3. One could however mention that this approach is a more general theory, for the subtle notions of finiteness helps to understand infinity in logic and in complexity.

A concrete version of the interpretation of cut-elimination by actions was provided by the context semantics of proof nets. This could be interpreted as the wandering of token, starting from the conclusions of the proof net and eventually exiting by some conclusion of the graph. Those pebbles moving in the structure without modifying it led to the study of *paths* in proof net [5], the normal form of the proof net being given by *persistent paths*. This led to the study of several kinds of *token machine* [96] and to the development of the study of algebra weight.

1.3 Previous Gateways between Linear Logic and Complexity

We will rapidly scan the wide spectrum of links between Linear Logic and complexity. This small section has two purposes: to present the numerous previous approaches, and that should ease the understanding of our contribution. We won't precise the settings that were developed, but gives several key-references for the reader who would like to push this study further.

First of all, Linear Logic and complexity are linked because some problems regarding Linear Logic arise naturally, and we can try to establish the complexity of those problems. This approach led to numerous interesting problems, and allowed a better understanding of the nature of Linear Logic.

Cut-elimination Given two proofs and a confluent cut-elimination procedure, do they have the same cut-free form? Mairson and Terui [100] established a hierarchy, proving that this problem is **P**-complete for MLL and **co-NP**-complete for MALL.

Correctness Given a proof-structure, is it a proof net? This problem was proven to be **NL**-complete no matter what the fragment considered is by Jacobé de Naurois and Mogbil [80].

Provability Given a formula, is it provable? This problem was proved to be **NP**-complete for MLL by Kanovich [87]. The reader may also have a look at Lincoln [98] and its consequent bibliography to learn more about the complexity of theorem proving.

Those problems give insights on the complexity of Linear Logic, but other approaches have been developed to address this question. One can use Linear Logic to deliver certification to programs, attesting that they will "behave well" no matter the context. This needs to define subsystems of LL and to establish that any program typable with this subsystem will be of such or such complexity.

To get rid of a *machine-based* characterisation of complexity classes, one could also perform the opposite operation: to design a "logic language of programming". We get sure that any program written in this syntax will be "pre-constrained"

The constraint on Linear Logic one could develop to moderate its expressivity was naturally the modalities. One of the oldest—and yet, very accurate— formulation of this idea is the following:

> "*[T]he rule for* '!' *indicates unlimited* possibilities *of duplication, but not a concrete one [...]. The clarification of this point could be of great importance: consider for instance* bounded exponentials $!_\alpha A$, $?_\alpha A$, *that could be added to linear logic with the intuitive meaning of 'iterate* α *times'. [T]here is some underlying polynomial structure in the exponentials. Now, it is not always the case that we can associate polynomials to all exponentials occuring in a proof of standard linear logic, especially when we have to deal with cut ; hence the proofs admitting such polynomial indexings are very peculiar.*"
>
> (Girard [49, p. 93])

This idea was later on developed by Girard, Scedrov, and Scott [63] under the name of "Bounded Linear Logic". It was later on refined [52], made more precise by Danos and Joinet [37] with the introduction of *Elementary Linear Logic* (ELL) and generalized by the introduction of *Linear Logic by level* by Baillot and Mazza [13]. Those works managed to characterise elementary and deterministic poly-time computation in Linear Logic, using sequents and proof nets. Schöpp [117] adapted this idea to second-order quantifier with benefits, for it gave a characterization of **L**.

Numerous studies were also developed with a purely semantical approach, but there is a conflict between their will to "characterise all the programs that compute the same function" and our objective to "characterise *the* good program that computes a function". The semantics is somehow "talkative", whereas the syntax focuses on the good programs, the more significant. A quote —with our emphasis— may clarify this point:

> "*A different approach to the semantics of bounded time complexity is possible: the basic idea is to measure by semantic means the execution of* any *program, regardless to its computational complexity. The aim is to compare different computational behaviours and to learn afterwards something on the very nature of bounded time complexity.*"
>
> (de Carvalho, Pagani, and Tortora de Falco [39])

One should remark moreover that this setting is not an Implicit Complexity approach at all.

Few works has been led in the GoI setting, which should recollect the advantages of the syntax and the advantages of semantics. Among them, it is worth referring to Baillot and Pedicini [14], which proposed an interpretation of Elementary Linear Logic by an algebra of clause equipped with resolution. We can also, to conclude this small scanning of the previous approaches, mention the work led by Dal Lago and Schöpp [35]. It is relevant for it introduces the functional programming language IntML that rely on Linear Logic. What is interesting is that it led to an actual implementation, whose source may be found at `https://github.com/uelis/IntML`.

In the Following...

The "level-by-level" approach has now reached a mature state. There is of course some enhancement that could be developed, but this structure begins to be quite well understood, capturing more and more programs, being less constraining. The work led by Schöpp [117] and IntML [35] are to our knowledge the only attempts to capture classes below **P**.

The two approach we are going to analyse do not rely heavily on the modalities to constrain the power of Linear Logic. In fact, they do not even appears in Boolean proof nets, which handle the computation by a kind of "hard-wire" setting: everything will be handled by the crossing of edges, or... wires. This will take benefit from the parallel writing of proofs as proof nets, but we still have to define a parallel execution for them. The next chapter will expose this setting and link it both to Boolean circuits and Alternating Turing Machines.

The other approach will rely on the GoI program, for we will encode proofs of integers in a von Neumann algebra and interpret nilpotency as normalisation, or in this case, acceptance. It will be wildly non-deterministic, for the computation will take place in product of matrices computing in parallel. Surprisingly, this framework can be "decided" with log-space resources, and that will conclude our Chapter 3.

To prove that we capture log-space computation, we will use in Chapter 4 abstract machines that manipulates pointers. This is a long-standing goal, as attested by a quotation that will conclude this introductory chapter:

> "*[G]etting rid of syntax [...] is the ultimate aim, which has been achieved only for multiplicatives [...] by Vincent* Danos *and Laurent* Regnier *[38]. In this case, the basic underlying structure turns out to be permutation of pointers, and the notion of a cyclic permutation plays a prominent role (*Danos *and* Regnier *replaced cyclicity by conditions involving connected acyclic graphs, i.e. trees). It seems that the general solution (not yet known, even for quantifiers) could be something like permutations of pointers, with variable addresses, so that some (simple form of) unification could be needed for the composition of moves : yet another connection with logic programming !*"
>
> (Girard [49, p. 101])

CHAPTER 2

Proof circuits

Contents

Proof nets are the traditional way of representing proofs of Linear Logic in parallel. And yet, they were not until the work by Terui[130] really used to model parallel computation. However, it is rather easy to define a parallel cut-elimination procedure and to extend the Proof-as-program correspondence to parallel evaluation in this setting. The ideal target to this extension are Boolean circuits, which are the classical parallel model of computation: they can be seen as electronic or digital circuits made of logic gates computing simple functions.

One of the computational property of Boolean circuits is that they are able to duplicate temporary values previously computed and to distribute them to several logic gates. We saw that duplication could be handled in an elegant way in Linear Logic by the ! and ? modalities, and yet we won't use this tool in the setting we propose here. Instead, we will build proof nets as hardware devices, where everything is explicit and where the duplication is "pre-constrained".

The model of computation we are going to define is close to graph rewriting, where the proof nets are made of only the multiplicative fragment of Linear Logic, and so every operation will be local. Because of the linearity of this logic, we are forced to keep track of the partial results generated by the evaluation that are unused in the result. It is a low-level of computation and every algorithm that can be written on Boolean circuits will be "in hard" translated into Boolean proof nets.

Regarding this first link, we will use the approach developed by Mogbil and Rahli [107, 105], who focused on uniform classes of complexity. We will present a novel translation in $\mathbf{AC}^0$ from Boolean proof nets to Boolean circuits by focusing on a simpler restricted notion of uniform Boolean proof nets—proof circuits. That will allow us to compare complexity classes below log-s pace, which were out of reach with the previous translations.

The second bridge we are going to build is between proof circuits and Alternating Turing Machines (ATMs). It will help us to present ATMs as a convenient model of parallel computation, and to prove that in some cases, logarithmic space is enough to establish the normal form of a proof net (see Section 2.4). Last but not least, this results exhibits a difference between our unbounded proof circuits and the bounded ones, something that could not be seen with the mere Boolean proof nets.

Regarding parallel complexity, a gentle introduction (in french) to this subject may be found in *Les petits cailloux* [112], but the most complete handbooks remains *Introduction to Circuit Complexity: A Uniform Approach*[135] and *Limits to Parallel Computation : P-Completeness theory* [65].

2.1 Basic Definitions: Boolean Circuits and MLL_u

We begin by introducing Boolean circuits: knowing them is not mandatory to understand how MLL_u proof nets compute, but it is useful to bear in mind notions like *uniformity*, *families of Boolean circuits* or *depth* of a Boolean circuit. Proof nets for MLL_u are an efficient way to model parallel computation with Linear Logic.

Boolean circuits

Boolean circuits (Definition 2.1.2) were first defined and popularized as a parallel model of computation by Borodin [19]. One of their feature is that they work only on inputs of fixed length, and that forces us to deal with *families* of Boolean circuits—and there arises the question of *uniformity* (Definition 2.1.4).

Definition 2.1.1 (Boolean function)**.** A *n-ary Boolean function* f^n is a map from $\{0,1\}^n$ to $\{0,1\}$. A *Boolean function family* is a sequence $f = (f^n)_{n\in\mathbb{N}}$ and a *basis* is a set of Boolean functions and Boolean function families. We define the two classical basis[1]:

$$\mathfrak{B}_b = \{\mathscr{N}_{\mathrm{eg}}, \vee^2, \wedge^2\} \text{ and } \mathfrak{B}_u = \{\mathscr{N}_{\mathrm{eg}}, (\vee^n)_{n\geqslant 2}, (\wedge^n)_{n\geqslant 2}\}$$

The Boolean function $\mathrm{USTConn}_2$ solves the problem *USTConn* $_2$ (Problem A), i.e. given in input the coding of an undirected graph G of degree at most 2 and two names of gates s and t, this function outputs 1 iff there is a path between s and t in G.

The purpose of this function will be detailed later on, and illustrated in Figure 2.4.

Definition 2.1.2 (Boolean circuit)**.** Given a basis $\mathfrak{B}$, a *Boolean circuit over* $\mathfrak{B}$ *with n inputs* C is a directed acyclic finite and labelled graph. The nodes of fan-in 0 are called *input nodes* and are labelled with $x_1, \ldots, x_n, 0, 1$. Non-input nodes are called *gates* and each one of them is labelled with a Boolean function from $\mathfrak{B}$ whose arity coincides with the fan-in of the gate. There is a unique node of fan-out 0 which is the *output gate*.[2] We indicate with a subscript the number of inputs: a Boolean circuit C with n inputs will be named C_n.

[1] Sometimes denoted respectively $\mathfrak{B}_0$ and $\mathfrak{B}_1$, but we adopt here the initials of *bounded* and *unbounded* as a reminder.

[2] We take into account in this definition only *decision circuits*, that is circuits that compute the characteristic function of a language. We will later on compose Boolean circuits with more than one output but took the liberty not to define them formally.

The *depth of a Boolean circuit* C_n, denoted by $d(C_n)$, is the length of the longest path between an input node and the output gate. Its *size* $|C_n|$ is its number of nodes. We will only consider Boolean circuits of size $n^{\mathcal{O}(1)}$, that is to say polynomial in the size of their input.

We say that C_n *accepts (resp. rejects) a word* $w \equiv w_1 \dots w_n \in \{0,1\}^n$ iff C_n evaluates to 1 (resp. 0) when $w_1, \dots, w_n$ are respectively assigned to $x_1, \dots, x_n$. A *family of Boolean circuits* is an infinite sequence $\mathbf{C} = (C_n)_{n\in\mathbb{N}}$ of Boolean circuits, $\mathbf{C}$ *accepts a language* $X \subseteq \{0,1\}^*$ iff for all $w \in X$, $C_{|w|}$ accepts w, and for all $w' \notin X$, $C_{|w'|}$ rejects w'.

Boolean circuits are "the" classical model of parallel computing, because we consider that their evaluation is linear in their depth: every gate is an independent processor evaluating its inputs as they are "ready". All the gates at the same depth are taken to compute at the same time, so depth is often compared to time on a sequential model of computation: the depth reflects the number of intermediate processing needed to output a value.

We now recall the definition of the *Direct Connection Language* of a family of Boolean circuits, an infinite sequence of tuples that describes it totally.

Definition 2.1.3 (Direct Connection Language Ruzzo [115, p. 371])**.** Given $\overline{(\cdot)}$ a suitable coding of integers and $\mathbf{C}$ a family of Boolean circuits over a basis $\mathfrak{B}$, its *Direct Connection Language* $L_{DC}(\mathbf{C})$ is the set of tuples $< 1^y, \overline{g}, \overline{p}, \overline{b} >$, such that g is a gate in C_y, labelled with $b \in \mathfrak{B}$ if $p = \epsilon$, else b is its pth predecessor.

We use the unary encoding to stay within suitable bounds for the uniformity, whose definition follows.

Definition 2.1.4 (Uniformity Barrington, Immerman, and Straubing [15])**.** A family $\mathbf{C}$ is said to be *DLOGTIME*-uniform if there exists a deterministic Turing Machine with random access to the input tape[3] that given $L_{DC}(\mathbf{C})$, 1^n and $\overline{g}$ outputs in time $\mathcal{O}(\log(|C_n|))$ any information (position, label or predecessors) about the gate g in C_n.

We can adopt the fine tuning of *DLOGTIME* many-one reductions developed by Regan and Vollmer [113]. This notion preserves transitivity and the membership to individual levels of hierarchy. Despite the fact that a *DLOGTIME* Turing Machine has more computational power than a constant-depth circuit, "a consensus has developed among researchers in circuit complexity that this *DLOGTIME* uniformity is the 'right' uniformity condition" (Hesse, Allender, and Barrington [70, p. 698]) for small complexity classes. Some proposal towards a circuit uniformity "sharper" than *DLOGTIME* were made by Bonfante and Mogbil [18], but we will stick to this well-accepted notion of uniformity and any further reference to uniformity in this chapter is to be read as *DLOGTIME* uniformity.

Uniformity allows us to be sure that there is a "sufficiently simple" way of describing the conception of any member of the family of Boolean circuits. Without it, even the problem not computable becomes computable, simply define a family of Boolean circuits that solves this problem: as you don't have to describe the way they are built —what uniformity is—, you don't have to bother if they may be constructed or not! Defining non-uniform classes of complexity led to some interesting questions and results, but it won't be the topic here, for we focus on a more tractable approach of algorithms.

[3] Meaning that the Turing Machine, given $n \in \mathbb{N}$, can in one step read the value written in the nth cell. For some fine tuning of this notion, we refer to Cai and Chen [23].

Definition 2.1.5 ($\mathbf{AC}^i$, $\mathbf{NC}^i$)**.** For all $i \in \mathbb{N}$, given $\mathfrak{B}$ a basis, a language $X \subseteq \{0,1\}^*$ belongs to the class $\mathbf{AC}^i(\mathfrak{B})$ (resp. $\mathbf{NC}^i(\mathfrak{B})$) if X is accepted by a uniform family of polynomial-size, $\log^i$-depth Boolean circuits over $\mathfrak{B}_u \cup \mathfrak{B}$ (resp. $\mathfrak{B}_b \cup \mathfrak{B}$). We set $\mathbf{AC}^i(\emptyset) = \mathbf{AC}^i$ and $\mathbf{NC}^i(\emptyset) = \mathbf{NC}^i$.

Of course this definition of $\mathbf{AC}^i$ and $\mathbf{NC}^i$ is equivalent to the definition *via* Alternating Turing Machines we gave previously in Definition 1.1.12.

Unbounded Multiplicative Linear Logic

We will here briefly expose the Logic underneath our proof circuits, a fragment of Linear Logic without modalities. We introduce those notions rapidly, for they were already developed in Section 1.2 and are quite classical.

Rather than using Multiplicative Linear Logic (MLL), we work with *unbounded*[4] *Multiplicative Linear Logic* (MLL_u) which differs only on the arities of the connectives but better relates to circuits (see the remark about "unfolding MLL_u" for a better insight). The following definitions are partially *folklore* since the introduction of "general multiplicative rule" by Danos and Regnier [38].

Definition 2.1.6 (Formulae of MLL_u)**.** Given α a literal and $n \geqslant 2$, formulae of MLL_u are:

$$A \coloneqq \alpha \mid \alpha^\perp \mid \otimes^n(A_1,\dots,A_n) \mid \parr^n(A_n,\dots,A_1)$$

We write $\overrightarrow{A}$ (resp. $\overleftarrow{A}$) for an ordered sequence of formulae $A_1,\dots,A_n$, (resp. $A_n,\dots,A_1$). Duality is defined with respect to De Morgan's law:

$$(A^\perp)^\perp \coloneqq A \qquad (\otimes^n(\overrightarrow{A}))^\perp \coloneqq \parr^n(\overleftarrow{A^\perp}) \qquad (\parr^n(\overleftarrow{A}))^\perp \coloneqq \otimes^n(\overrightarrow{A^\perp})$$

As for the rest of this chapter, consider that A, B and D will refer to MLL_u formulae. The result of the substitution of all the occurrences of B by an occurrence of D in A will be written $A[B/D]$, $A[D]$ if $B = \alpha$. We will sometimes write $A \parr B$ (resp. $A \otimes B$) for $\parr^2(A,B)$ (resp. $\otimes^2(A,B)$).

Definition 2.1.7 (Sequent calculus for MLL_u)**.** A *sequent* of MLL_u is of the form $\vdash \Gamma$, where Γ is a multiset of formulae. The *inference rules* of MLL_u are as follow:

$$\frac{}{\vdash A, A^\perp}\ ax. \qquad \frac{\vdash \Gamma_1, A_1 \quad \dots \quad \vdash \Gamma_n, A_n}{\vdash \Gamma_1,\dots,\Gamma_n, \otimes^n(\overrightarrow{A})}\ \otimes^n$$

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\ cut \qquad \frac{\vdash \Gamma, \overleftarrow{A}}{\vdash \Gamma, \parr^n(\overleftarrow{A})}\ \parr^n$$

Proofs of MLL_u are built with respect to those rules, following the construction explained page 13. MLL_u has neither weakening nor contraction, but admits implicit exchange and eliminates cuts. The formulae A and $A^\perp$ in the rule *cut* are called the *cut formulae*.

[4] That has nothing to do with Bounded Linear Logic: both have complexity concern, it is just that in our case we take *the connectives* to be unbounded, and not *the modalities*, that are by nature unbounded.

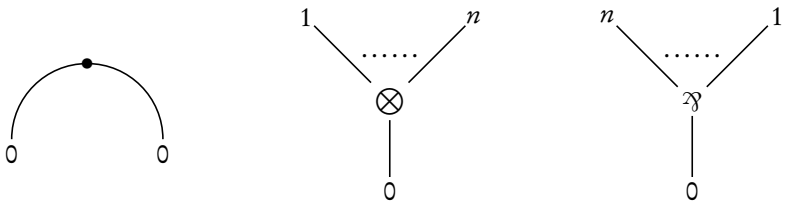


Figure 2.1: The components of a proof net: *ax.*-link, $\otimes^n$-link and $\parr^n$-link

Proof Nets for MLL$_u$

Proof nets are a parallel syntax for MLL$_u$ that abstracts away everything irrelevant and only keeps the structure of the proofs. We introduce measures (Definition 2.1.10) on them in order to study their structure and complexity, and a parallel elimination of their cuts (Definition 2.1.11).

Definition 2.1.8 (Links)**.** We introduce in Figure 2.1 three sorts of *links* —•, $\otimes^n$ and $\parr^n$– which correspond to MLL$_u$ rules.

Every link may have two kinds of *ports*: *principal* ones, indexed by 0 and written below, and *auxiliary* ones, indexed by $1,\dots,n$ and written above. The auxiliary ports are ordered, but as we always represent the links as in Figure 2.1, we may safely omit the numbering. Axiom links have two principal ports, both indexed with 0, but we can always differentiate them (for instance by naming them 0_r and 0_l) if we need to. Remark that there is no cut link: a cut is represented with an edge between two principal ports.

Definition 2.1.9 (Proof nets for MLL$_u$)**.** Given a proof of MLL$_u$, we may as usual build a *proof net* from it.

The *type of a proof net* $\mathcal{P}$ is Γ if there exists a proof of $\vdash \Gamma$ that infers it. A proof net always has several types, but up to α-equivalence (renaming of the literals) we may always assume it has a unique *principal type*. If a proof net may be typed with Γ, then for every A it may be typed with $\Gamma[A]$. By extension we will use the notion of type of an edge.

Remark. The same proof net—as it abstracts proofs— may be inferred by several proofs. Conversely, several graphs —as representations of proof nets— may correspond to the same proof net: we get round of this difficulty by associating to every proof net a single drawing among the possible drawings with the minimal number of crossings between edges, for the sake of simplicity. Two graphs representing proof nets that can be obtained from the same proof are taken to be equal.

Terui uses a notion of *decorated proofs* to build his proof nets directly from proofs. It allows not to take into account *pseudo nets* and to avoid having to define a correction criterion. But it has a major drawback: in fact Terui does not use proof nets to compute, but stick on this level of "description". They are a rather convenient linear description of the objects we manipulate, but our setting allows us to skip this intermediate step, and to truly compute with proof nets.

Definition 2.1.10 (Size and depth of a proof net)**.** The size $|\mathcal{P}|$ of a proof net $\mathcal{P}$ is the number of its links. The depth of a proof net is defined with respect to its type:

- The depth of a formula is defined by recurrence:

$$d(\alpha) = d(\alpha^\perp) = 1 \qquad d(\otimes^n(\overrightarrow{A})) = d(\parr^n(\overleftarrow{A})) = 1 + \max(d(A_1),\dots,d(A_n))$$

- The depth $d(\pi)$ of a proof π is the maximum depth of cut formulae in it.
- The depth $d(\mathcal{P})$ of a proof net $\mathcal{P}$ is $\min\{d(\pi) \mid \pi \text{ infers } \mathcal{P}\}$. The depth $d(\mathcal{P})$ of a proof net depends on its type, but it is minimal when we consider its principal type.

Now that we defined measurements on our objects, we can justify the use of *unbounded* Multiplicative Linear Logic. Terui justifies this choice by writing that "MLL_u just gives us a depth-efficient way of writing proofs" and refer to the classical work on generalized multiplicatives previously mentioned [38]. This last approach is a classical work in Linear Logic and a major outbreak in the study of Linear Logic, but it focused on logical, mathematical and somehow combinatorial topics. The following remark explains from a complexity point of view this choice.

Remark (Unfolding MLL_u, folding MLL). We take the more "depth-efficient" way of encoding MLL into MLL_u, namely:

$$\otimes^i(A_1,\dots,A_i) \equiv (\dots((A_1 \otimes A_2) \otimes (A_3 \otimes A_4)) \otimes \dots \otimes ((A_{i-3} \otimes A_{i-2}) \otimes (A_{i-1} \otimes A_i))\dots)$$

But we adopt MLL_u not only for convenience, for going back and forth between MLL_u and MLL has a cost. To unfold a proof net of MLL_u, we have to perform the two operations in Figure 2.2 for every $⅋^i$- and $\otimes^i$-link with $i > 2$. They are strictly local and thus can be performed with $\mathbf{AC}^0$ resources.

Doing the opposite operation —folding a MLL proof net into a MLL_u proof net— is a more global operation, for it has to identify the maximal chains of $⅋$ or $\otimes$ that can be folded and hence cannot be performed efficiently in parallel. But a really simple algorithm could be written to perform such folding, and it can be proved that log-s pace resources would be enough to execute it.[5]

It could be easily proved that this (un)folding operation is sound with respect to the type and the normalisation of the proof nets.

Concerning the bounds of such proof nets, let $\mathcal{P}$ be a MLL_u proof net, $d(\mathcal{P})$ its depth and $|\mathcal{P}|$ its size. Let $|c_i|_\mathcal{P}$ be the number of links with i auxiliary ports in $\mathcal{P}$, we have $|\mathcal{P}| \geqslant \sum_{i \geqslant 2} |c_i|_\mathcal{P}$. Now let $\mathcal{Q}$ be the MLL un-folded version of $\mathcal{P}$, it is clear that $|\mathcal{Q}| \geqslant \sum_{i \geqslant 2} |c_i|_\mathcal{P} \times \log(i)$. In the worst case the maximal arity of a link in $\mathcal{P}$ is $\mathcal{O}(|\mathcal{P}|)$, and we know that in this case $|\mathcal{Q}|$ is $\mathcal{O}(|\mathcal{P}| \times \log(|\mathcal{P}|))$. For the same reason, the depth of $\mathcal{Q}$ is in the worst case $\mathcal{O}(d(\mathcal{P}) \times \log(|\mathcal{P}|))$.

Roughly, we lose a lot regarding the depth if we stick to MLL proof nets, because we will take our proof nets to be of depth logarithmic in the size of the input. But as soon as the proof nets for MLL develop a computational power superior or equal to $\mathbf{L}$, we may "for free" rewrite them as MLL_u proof nets.

Definition 2.1.11 (Cuts and parallel cut-elimination)**.** A cut is an edge between the principal ports of two links. If one of these links is an *ax.*-link, two cases occur:

if the other link is an *ax.*-link, we take the maximal chain of *ax.*-links connected by their principal ports and defines this set of cuts as a *t-cut*,

otherwise the cut is an *a-cut*.

Otherwise it is a *m-cut* and we know that for $n \geqslant 2$, one link is a $\otimes^n$-link and the other is a $⅋^n$-link.

We define on Figure 2.3 three rewriting rules on the proof nets. For $r \in \{t, a, m\}$, if $\mathcal{Q}$ may be obtained from $\mathcal{P}$ by erasing all the r-c uts of $\mathcal{P}$ in parallel, we write $\mathcal{P} \rightrightarrows_r \mathcal{Q}$. If $\mathcal{P} \rightrightarrows_t \mathcal{Q}$, $\mathcal{P} \rightrightarrows_a \mathcal{Q}$ or

[5] Those two algorithms and complexity bounds would require to be stated formally the Direct Connection Language for proof nets, whose definition is to come. As it is not of major interest, we take the liberty to skip this red tape.

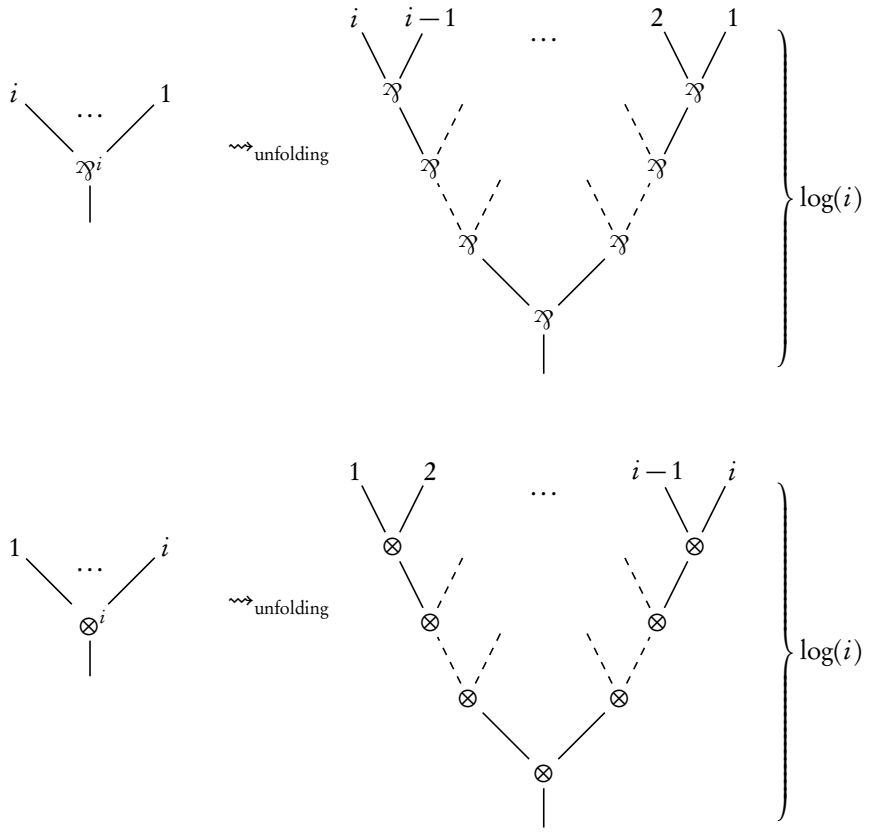


Figure 2.2: Unfolding a MLL_u proof net

For all $\circ \in \{(\parr^n)_{n\geqslant 2}, (\otimes^n)_{n\geqslant 2}\}$, $\circ$ may be $\bullet$ in $\to_m$.

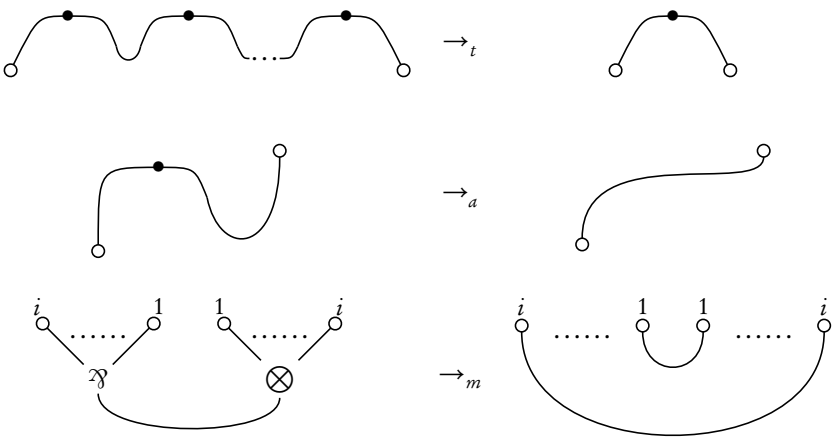


Figure 2.3: Cut-elimination in parallel: t-, a- and m-reductions

$\mathcal{P} \rightrightarrows_m \mathcal{Q}$, we write $\mathcal{P} \rightrightarrows \mathcal{Q}$. To *normalise a proof net* $\mathcal{P}$ is to apply $\rightrightarrows$ until we reach a cut-free proof net. We define $\rightrightarrows^*$ as the transitive reflexive closure of $\rightrightarrows$.

The t-cut is usually not defined, as it is just a special case of the a-c ut. It was introduced by Terui in order to be able to eliminate the cuts of a proof net in parallel. We know since "Linear logic" [47] that the number of steps needed to normalise a proof net is linear in its size. This is unsatisfactory for us in terms of parallelization: we would like the number of steps to be relative to the depth of the proof net, as it is the case for the Boolean circuits.

If we try roughly to reduce in parallel a cut between two ax.-links, we are faced with a critical pair, as exposed in Figure 2.4. Terui avoids this situation by using this t-reduction (*tightening reduction*) which eliminates in one step all those critical pairs. Of course, this rule re-introduce some sequentialization in the parallel cut-elimination[6] but after applying it, we can then safely reduce all the other cuts in parallel.

Theorem 2.1.1 (Parallel cut-elimination Terui [130])**.** *Every proof net $\mathcal{P}$ normalises in at most $\mathcal{O}(d(\mathcal{P}))$ applications of $\rightrightarrows$.*

The idea of the proof is quite simple: after the application of $\rightrightarrows_t$ and $\rightrightarrows_a$, no t-cut or a-cut remains in the proof net, because $\rightrightarrows_a$ cannot create a t-cut. So either the proof net obtained is cut-free, and the proof is done, either some m-cuts are present. In this latter case, an application of $\rightrightarrows_m$ eliminates at least one m-cut and the depth of the whole proof net lowers by 1.

By iterating the procedure, we get the expected result. This procedure allows us to make the most of the computational power of proof nets, by achieving a "speed-up" in the number of steps needed to normalise them.

Now that we know that one could compute efficiently, in parallel, with proof nets, we should wonder *what* we can compute!

2.2 Boolean Proof Nets

We will begin with a reminder of *how* a boolean value may be represented and computed with proof nets. Next we will introduce our contribution, proof circuits, and make sure that they are proof nets.

Proof Nets that compute Boolean functions

We just saw that proof nets are accompanied by a mechanism of evaluation, but this is not sufficient to represent actual computation. We have now to define how proof nets represent Boolean values (Definition 2.2.1) and Boolean functions (Definition 2.2.2). To study them in a uniform framework we define their Direct Connection Language (Definition 2.2.3). This will allow us later on to use them as "true" model of computation, i.e. a "reasonable" model if we remember the Invariance Thesis.

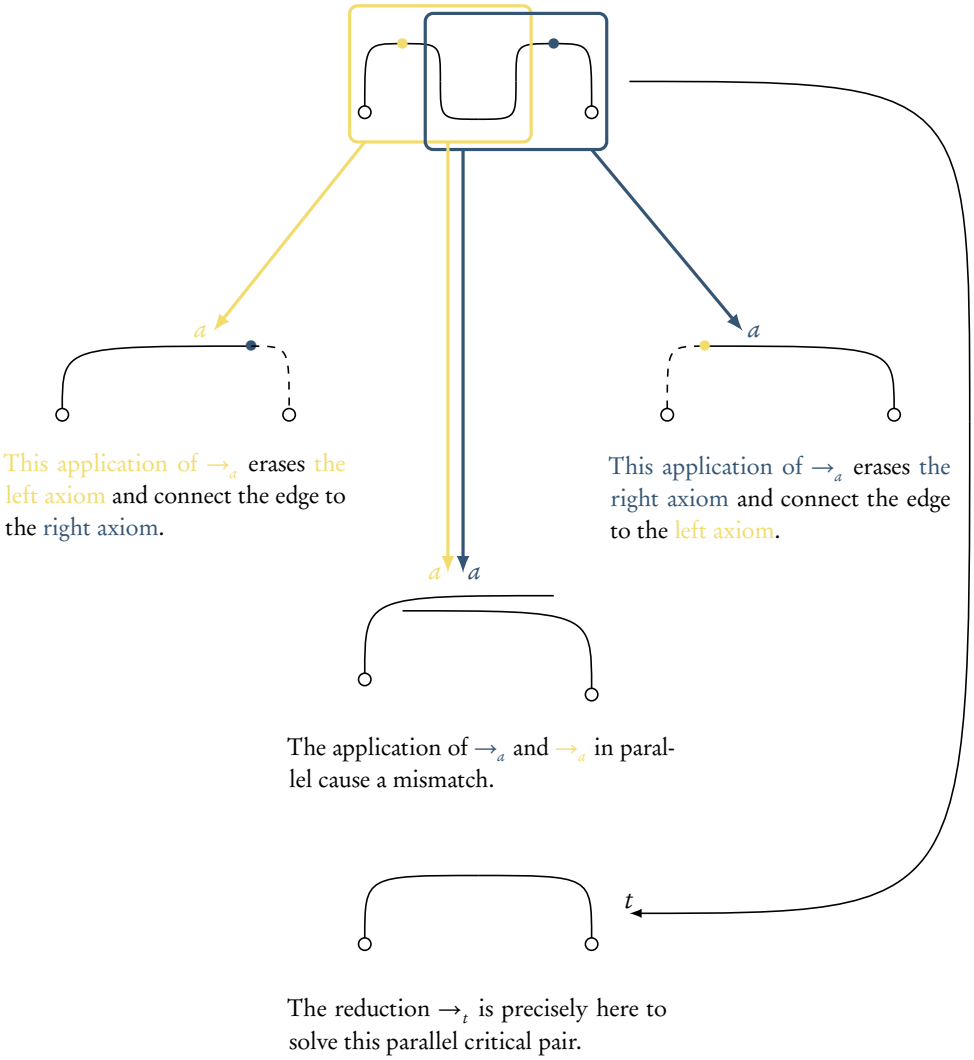
We cannot in MLL_u derive the classical representation of the Boolean values, i.e. $\alpha \multimap \alpha \multimap \alpha$, but we use its ternary variant, without weakening.

Definition 2.2.1 (Boolean type, b_0 and b_1 Terui [130])**.** Let $\mathbf{B}$ be the boolean type of MLL_u:

$$\mathbf{B} \coloneqq \parr^3(\alpha^\perp, \alpha^\perp, (\alpha \otimes \alpha))$$

Its proof is

[6] The process of identifying maximal chains of axioms with cuts between them is necessarily done sequentially.



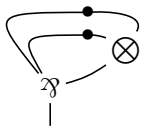
The proof net at the top of the figure has a single cut between two axioms, but it can be reduced in two different ways, depending on the axiom we "keep". This is illustrated by the two rectangles, which emphasize the two premisses of the application of $\to_a$. If we perform the two possible applications in parallel, we do not get the expected result.
The t-r eduction will handle this situation, but it requires the $\mathrm{USTConn}_2$ function to be performed in parallel, in order to identify the maximal chains of axioms with a cut between them.

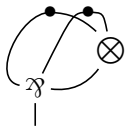
Figure 2.4: An obstacle to cut-elimination in parallel

$$\dfrac{\dfrac{\dfrac{}{\vdash \alpha, \alpha^\perp}\,\text{ax.} \quad \dfrac{}{\vdash \alpha, \alpha^\perp}\,\text{ax.}}{\vdash \alpha^\perp, \alpha^\perp, (\alpha \otimes \alpha)}\,\otimes^2}{\vdash \parr^3(\alpha^\perp, \alpha^\perp, (\alpha \otimes \alpha))}\,\parr^3$$

But if we look closely, this proof is not unique: does the first literal $\alpha^\perp$ of the $\parr$ connective comes from the right hand-side axiom or from the left hand-side axiom[7]?

Depending on the answer to this question, we obtain the two proof nets of type $\mathbf{B}$: b_0 and b_1, respectively used to represent false and true.

$b_0 \coloneqq$ 

$b_1 \coloneqq$ 

We write $\overrightarrow{b}$ for $b_{i_1}, \dots, b_{i_n}$ with $i_1, \dots, i_n \in \{0, 1\}$.

As we can see, b_0 and b_1 differ only on their planarity: the proof nets exhibit the exchange that was kept implicit in the proof. This difference of planarity is the key of the encoding of the boolean values into proof nets.

We present briefly two examples of computation: Figure 2.5 shows how to inverse the planarity and Figure 2.6 how conditional can be simulated thanks to planarity.

Definition 2.2.2 (Boolean proof nets)**.** A *Boolean proof net with n inputs* is a proof net $\mathsf{P}_n(\overrightarrow{p})$ of type

$$\vdash \mathbf{B}^\perp[A_1], \dots, \mathbf{B}^\perp[A_n], \otimes^{1+m}(\mathbf{B}[A], D_1, \dots, D_m)$$

The tensor in this type is the *result tensor*: it collects the result of the computation on its first auxiliary port and the *garbage* —here of type $D_1, \dots, D_m$– on its other auxiliary ports which will be called the *garbage ports*.

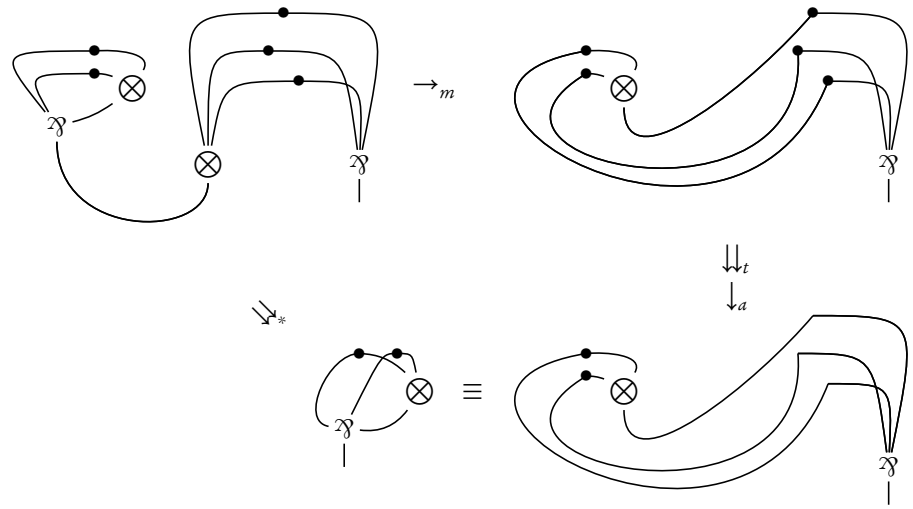
Given $\overrightarrow{b}$ of length n, $\mathsf{P}_n(\overrightarrow{b})$ is obtained by connecting with cuts $\mathbf{B}^\perp[A_j]$ to b_{i_j} for all $1 \leq j \leq n$. We have that $\mathsf{P}_n(\overrightarrow{b}) \rightrightarrows^* \mathsf{Q}$ where Q is unique, cut-free and of type $\otimes^{1+m}(\mathbf{B}[A], D_1, \dots, D_m)$. Let b_r be the proof net of type $\mathbf{B}$ connected to the first port of $\otimes^{1+m}$. We know since Definition 2.2.1 that the only two[8] normal proof nets of type $\mathbf{B}$ are b_0 and b_1, so $r \in \{0, 1\}$ is considered as the result of this computation, and we write $\mathsf{P}_n(\overrightarrow{b}) \to_{\text{ev.}} b_r$, see Figure 2.7 for a better insight.

As for the Boolean circuits, a Boolean proof net can only receive a fixed size of input, so we need to work with families of Boolean proof nets $\mathbf{P} = (\mathsf{P}_n)_{n \in \mathbb{N}}$. We say that a Boolean proof net family $\mathbf{P}$ *represents a Boolean function* f^n if for all $w \equiv i_1 \dots i_n \in \{0, 1\}^n$,

$$\mathsf{P}_n(b_{i_1}, \dots, b_{i_n}) \to_{\text{ev.}} b_{f(w)}$$

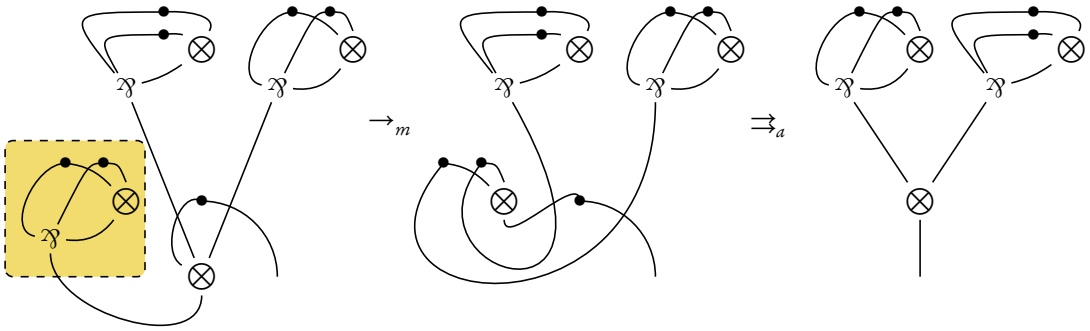
[7] This ambiguity rests of course on the implicit exchange. In fact, there could be more exchanges, but we can up to reordering consider only this one.

[8] Up to η-expansion, but we can safely ignore this fact by assuming that the type chosen is minimal, i.e. that b_r is of type $\mathbf{B}$ and not of type $\mathbf{B}[C]$ for any $C \neq \alpha$.



The proof net we present is the result of the "branching", i.e. the connection *via* a cut, of b_0 with a proof net that will be defined later on as $\mathcal{N}_{\text{eg}}$. This proof net adds during the cut-elimination a cross between two edges. One can simply check that if the input was b_1, this proof net would have removed the crossing between the edges. Therefore, this small proof net is the function mapping 0 to 1 and 1 to 0: the negation.

Figure 2.5: Adding or removing a crossing: how negation works



The input (here in a dashed rectangle) is a proof net of type $\mathbf{B}[\mathbf{B}]$, and it "selects" —according to its planarity or non-planarity— during the normalisation which one of b_0 or b_1 is connected to the first auxiliary port of the tensor and will be considered as the result —the other being treated as garbage.
We will in the following replace b_0 and b_1 with more complex proof nets, let's say $\mathcal{P}$ and $\mathcal{Q}$, and we can remark that this proof net compute "if the input is true then $\mathcal{Q}$ else $\mathcal{P}$".

Figure 2.6: The conditional, the core of the computation

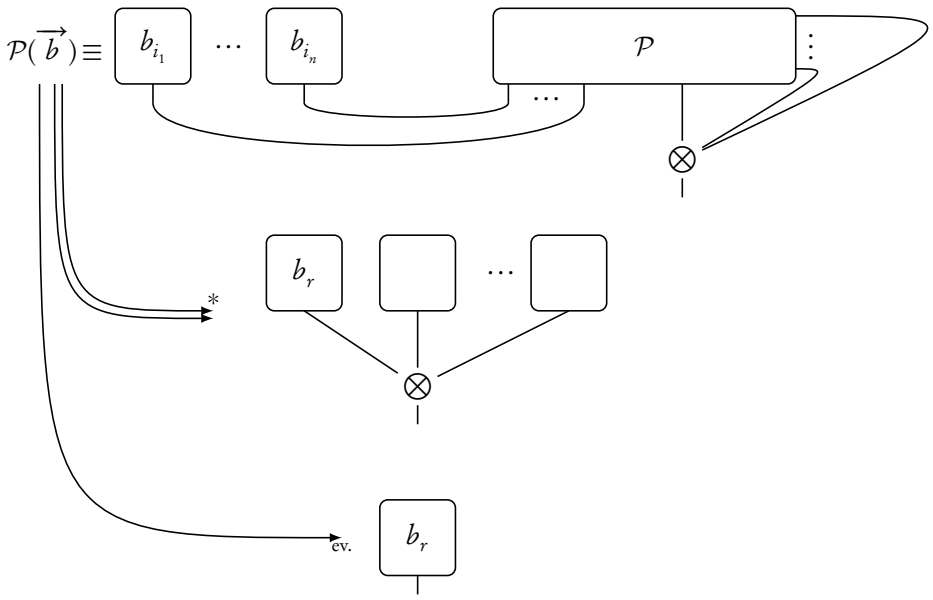


Figure 2.7: Branching variables on a Boolean proof net and evaluating it

Remark that the Boolean type may contains more than literals, thanks to the substitution. Without substitutions, Boolean proof nets would be *flat*, i.e. necessarily of type

$$\mathbf{B}^{\perp}[\alpha],\ldots,\mathbf{B}^{\perp}[\alpha],\otimes^{1+m}(\mathbf{B}[\alpha],D_1,\ldots,D_m)$$

and Terui proved that in this case we could not compute more than $Parity^n$ or its negation!

We may easily define *language accepted by a family of Boolean proof nets* and *uniformity*, thanks to the definition of Direct Connection Language for proof nets that follows.

Definition 2.2.3 (Direct Connection Language for proof nets Mogbil and Rahli [107])**.** Given $\mathbf{P} = (\mathsf{P}_n)_{n\in\mathbb{N}}$ a family of Boolean proof nets, its *Direct Connection Language* $\mathrm{L}_{\mathrm{DC}}(\mathsf{P})$ is the set of tuples $< 1^y, \overline{g}, \overline{p}, \overline{b} >$ where g is a link in P_y, of sort b if $p = \epsilon$ else the pth premise of g is the link b.

If $< 1^y, \overline{p}, 0, \overline{b} >$ or $< 1^y, \overline{b}, 0, \overline{p} >$ belong to $\mathrm{L}_{\mathrm{DC}}(\mathsf{P})$, there is a cut between b and p in P_y.

Of course this definition is really close to the definition of Direct Connection Language for Boolean circuits (Definition 2.1.3), for the object are in fact very similar. We will make them even more similar by constraining how Boolean proof nets are built.

Proof Circuits

Proof circuits were introduced in my *mémoire de* Master 2[7] and appears in a paper published in the Proceedings of the Second Workshop on Developments in Implicit Computational Complexity [8]. They make it simpler to grasp the mechanisms of computation at work in the Boolean proof nets. They are studied in a uniform framework, following the work initiated by Mogbil and Rahli [107, 105], but they help to avoid a complex and tedious composition and are closer to Boolean circuits; also strictly speaking, for we extend the correspondence between them to sublogarithmic classes of complexity.

A proof circuit is a Boolean proof net (Theorem 2.2.3) made out of pieces (Definition 2.2.4) which represents Boolean functions, constants or duplicates values. Garbage is manipulated in an innovative way, as shown in Figure 2.9. The mechanisms of computation explained in Figure 2.5 and Figure 2.6 remain valid in this setting. Apart from the constants and the $\mathscr{N}_{\text{eg}}$ piece, all pieces uses the conditional presented in Figure 2.6 with a "built-in" composition.

From now on every edge represented by  is connected on its right to a free (unconnected) garbage port of the result tensor: it carries a piece of garbage. As there is no modality in MLL_{u}, the computation cannot erase the useless values generated during the course of the computation, so we handle them differently.

Patrick Baillot argued that proof circuits should rather be seen as a target to the translation from Boolean circuits to Boolean proof nets, as a technical tool of a proof, rather than as an object to define classes of complexity. It is true that the original motivation that led to introduce them was to sharpen the translation to be developed in Section 2.3. Yet, as a by-product, they allow to define a framework where the fan-in of the *pieces* is bounded, and so to get an equivalent to the bounded Boolean circuits. It is true that their nature makes them look rather as Boolean circuits, but the bridging they propose between those two worlds is the key to the new correspondence developed in Section 2.4.

Definition 2.2.4 (Pieces)**.** We present in Table 2.8 the set of *pieces* at our disposal. Entries are labelled with e, exits with s and garbage with g. Edges labelled b_k —for $k \in \{0,1\}$– are connected to the edge labelled s of the piece b_k.

A *piece* $\mathscr{P}$ *with* $i \geqslant 0$ *entries,* $j \geqslant 1$ *exits and* $k \geqslant 0$ *garbage* is one of the pseudo nets in Table 2.8, where i edges are labelled with $e_1, \dots, e_i$, j edges are labelled with $s_1, \dots, s_j$ and k edges labelled with $g_1, \dots, g_k$ go to the garbage ports of the result tensor.

Table 2.8: Pieces of Proof Circuits

We set $2 \leqslant j \leqslant i$. If $i = 2$, the edge a *is* the edge s in the pieces $\mathscr{D}^i_{\text{isj}}$ and $\mathscr{C}^i_{\text{onj}}$.

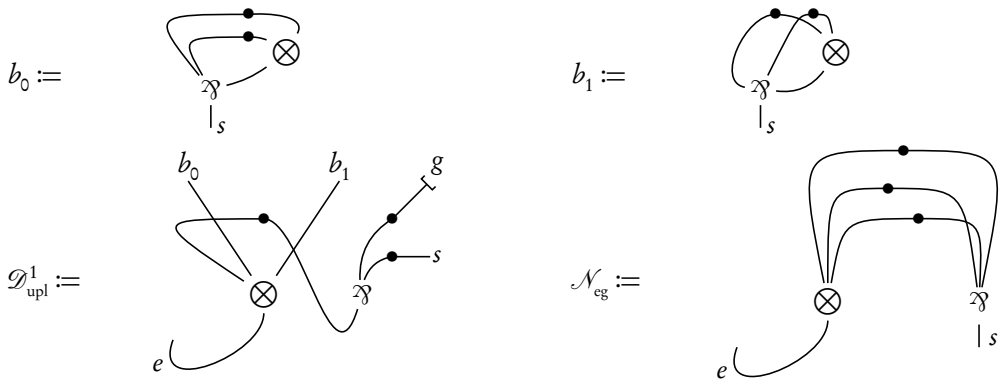
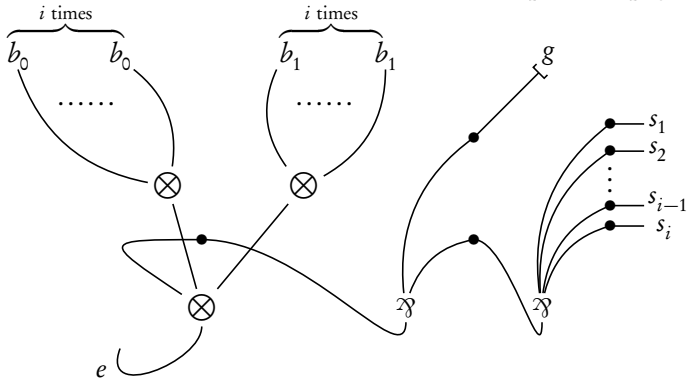
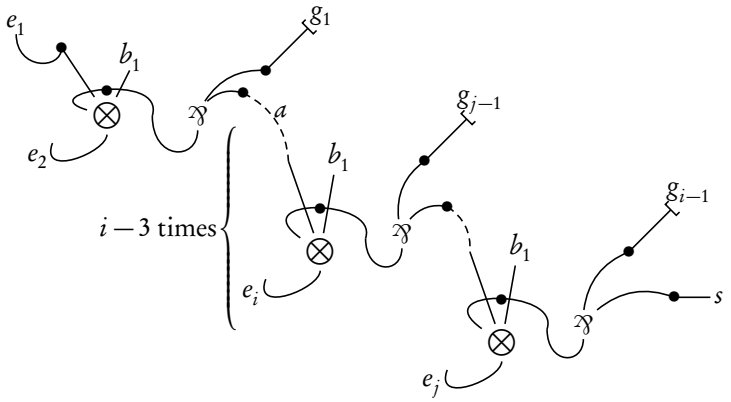


Table 2.8: Pieces – continued from previous page

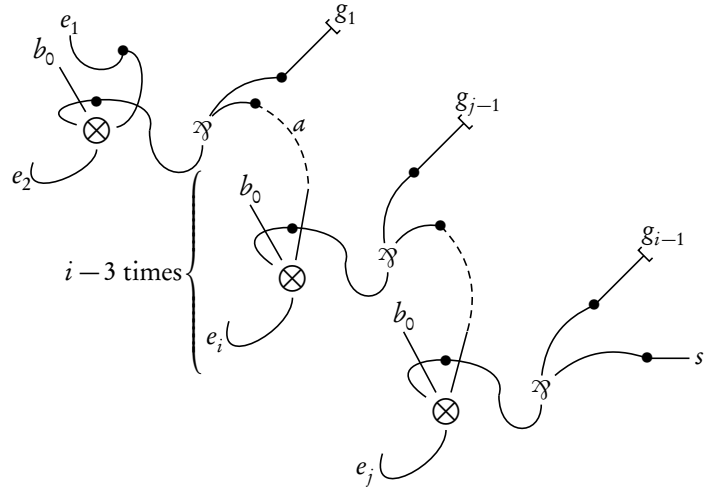
$\mathscr{D}^i_{\text{upl}} :=$



$\mathscr{D}^i_{\text{isj}} :=$



$\mathscr{C}^i_{\text{onj}} :=$



We define two sets of pieces:

$$\mathscr{P}_u = \{b_0, b_1, \mathscr{N}_{\text{eg}}, \{\mathscr{D}^i_{\text{upl}}\}_{i\geqslant 1}, \{\mathscr{D}^i_{\text{isj}}\}_{i\geqslant 2}, \{\mathscr{C}^i_{\text{onj}}\}_{i\geqslant 2}\}$$

and

$$\mathscr{P}_b = \{b_0, b_1, \mathscr{N}_{\text{eg}}, \{\mathscr{D}^i_{\text{upl}}\}_{i\geqslant 1}, \mathscr{D}^2_{\text{isj}}, \mathscr{C}^2_{\text{onj}}\}$$

To *compose two pieces* $\mathscr{P}_1$ and $\mathscr{P}_2$, we connect an exit of $\mathscr{P}_1$ to an entry of $\mathscr{P}_2$. It is not allowed to loop: we cannot connect an entry and an exit belonging to the same piece. All the same, we cannot have $\mathscr{P}_1$ connected to $\mathscr{P}_2$ connected to $\mathscr{P}_3$ ... connected to $\mathscr{P}_1$.

An entry (resp. an exit) that is not connected to an exit (resp. an entry) of another piece is said to be *unconnected*.

A piece with i entries and j exists will sometimes be referred to as having *fan-in* i and *fan-out* j, as for the gates of the Boolean circuits.

Definition 2.2.5 (Proof circuits)**.** A *proof circuit* $\mathsf{P}_n(\overrightarrow{p})$ *with n inputs and one output* is obtained by composing pieces such that n entries and one exit are unconnected. If no garbage is created we add a $\mathscr{D}^1_{\text{upl}}$ piece connected to the unconnected exit to produce some artificially. Then we add a result tensor whose first auxiliary port is connected to the exit —which is also the output of the proof circuit— and whose garbage ports are connected to the garbage of the pieces. We then label every unconnected entries with $p_1, \dots, p_n$: those are the inputs of the proof circuit.

Given $\overrightarrow{b}$ of length n, $\mathsf{P}_n(\overrightarrow{b})$ is obtained by connecting with cuts p_j to b_{i_j} for all $1 \leq j \leq n$. The Figure 2.7 can be used as it to describe the branching of the inputs.

Proof circuits come in two flavours: they may be composed of *unbounded* pieces (taken in the set $\mathscr{P}_u$ or of *bounded* (or *binaries*) pieces (taken from the set $\mathscr{P}_b$), as for the Boolean circuits over the basis $\mathfrak{B}_u$ or $\mathfrak{B}_b$. Yet the Linear Logic we use is in both cases the unbounded MLL_u, and the $\mathscr{D}^i_{\text{upl}}$ pieces can be of any fan-out.

Without loss of generality, one may consider that the $\mathscr{N}_{\text{eg}}$ piece are only connected to the inputs of the proof circuit and that a $\mathscr{D}^i_{\text{upl}}$ piece is never connected to another $\mathscr{D}^i_{\text{upl}}$ piece: we do not lose computational power if we negate only the entries, and we don't need to duplicate i times the values that have just been duplicated j times, for we might as well duplicate it $i+j$ times directly. This comes from a semantic of evaluation of those proof circuits that will come soon.

There are two main novelties in this way to define Boolean proof nets:

First, the composition is handled by the pieces themselves. If you recall Figure 2.6, you may notice that the value computed, the output, is connected to this first port of the tensor. So to access this value, one has to cut the tensor with a par, to "extract" the output to be able to compute with it. Here the garbage is evacuated as soon as it is produced. We present in Figure 2.9 the difference of representation of the disjunction of arity 4 between Terui's framework and our setting.

We define directly proof nets that can be built easily and where mechanisms of computation are simpler to grasp. This allow us to lower the computational power needed to build them, as we will see in Section 2.3.

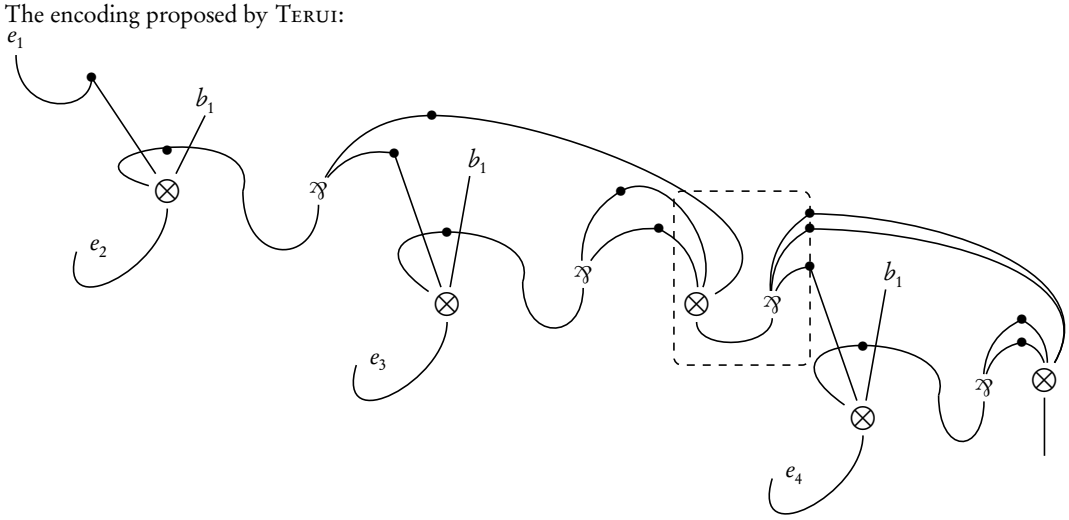
But first we must make sure that they are proof nets, and for that we will need the following definition.

Definition 2.2.6 (Height of a piece)**.** Let P be a proof circuit, we define for any of its pieces $\mathscr{P}$, its *height* denoted by $h(\mathscr{P})$:

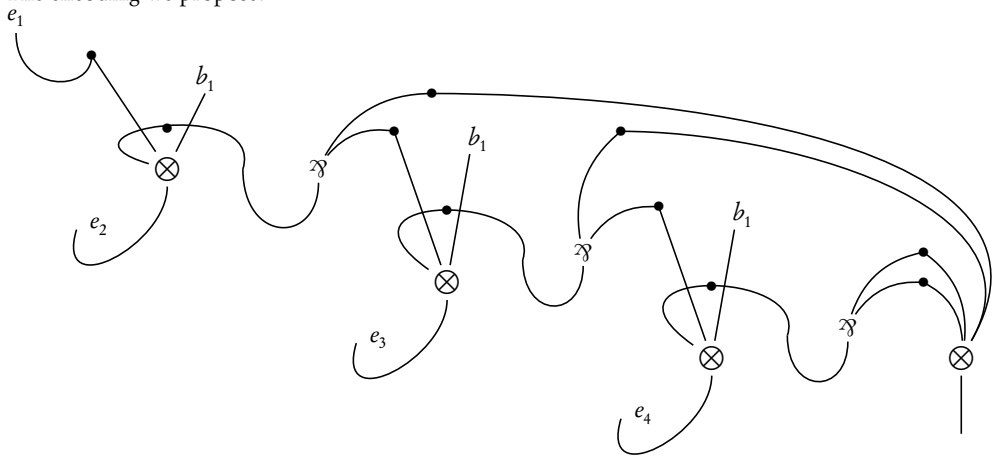
If the exit of $\mathscr{P}$ is connected to the result tensor, then $h(\mathscr{P}) = 0$.

If $\mathscr{P}$ has a single exit and it is connected to an entry of a piece $\mathscr{P}'$, then $h(\mathscr{P}) = h(\mathscr{P}') + 1$.

We give below two encodings of $\vee^4$, the disjunction of arity 4.
The encoding proposed by Terui:

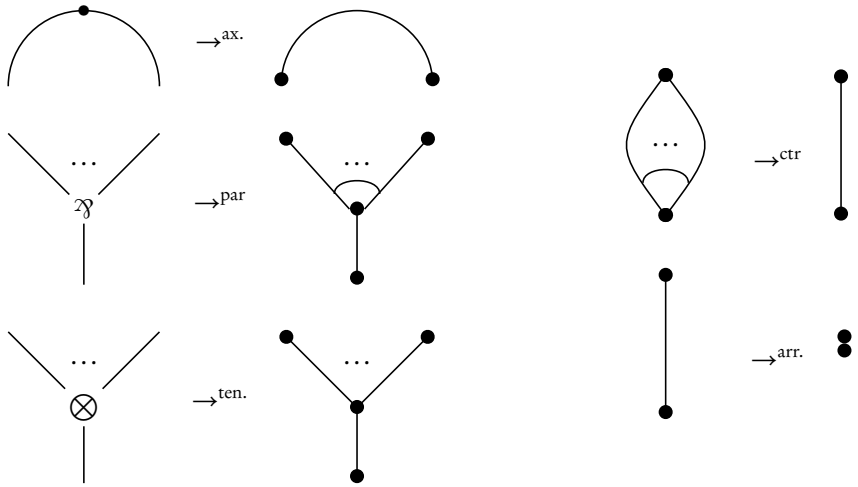


The encoding we propose:



Note that the couple $\otimes/\parr$ in the first encoding (in a dashed rectangle) disappears in the second encoding, exactly as if we had reduced this cut. It might look quite a minimal improvement, but it has to be conceived at the scale of the whole Boolean proof net. We will show in the proof of Theorem 2.3.3 that it allows us to drastically lower the size of the manipulated object.

Figure 2.9: The improvement of "built-in" composition



The three rules on the left goes from pseudo nets to dotted graphs, the two on the right from dotted graphs to dotted graphs. The dots must be different in the case of $\to^{\text{ctr}}$ and $\to^{\text{arr.}}$. In the $\to^{\text{arr.}}$ case, there should be no semi-arc coming from the relabelling of a $\parr$.

Figure 2.10: Contractibility rules: relabelling and contracting a pseudo net

If $\mathscr{P}$ has $i > 1$ exits connected to entries of $\mathscr{P}_1, \dots, \mathscr{P}_i$, then $h(\mathscr{P}) = \max_{n \leqslant i}(h(\mathscr{P}_n)) + 1$.

Why this definition ? For two reasons: as we don't know yet that proof circuits are proof nets, we do not know that they may be typed, and yet we need to be able to measure the degree of "overlapping" of their pieces. The second reason is that the depth do not increase each time we cross a piece, as it will be shown in Lemma 2.3.1, but we need a criterion to reason inductively with. We use this notion in the following subsection, to prove Theorem 2.2.3.

Contractibility criterion: Proof Circuits are Boolean Proof Nets

For the time being, proof circuits are only freely-generated by connecting pieces, and we have no insurance whatsoever that they are actually proof nets. They are only pseudo nets, for we don't know yet if they are typable, if every one of them may be linearized as a proof.

To prove that all proof circuit is a proof net, we use a generalized version of the contractibility criteria of Danos [36]. We adopt its variant introduced by Lafont [93] and refined by Guerrini and Masini [67] who use *parsing boxes*.

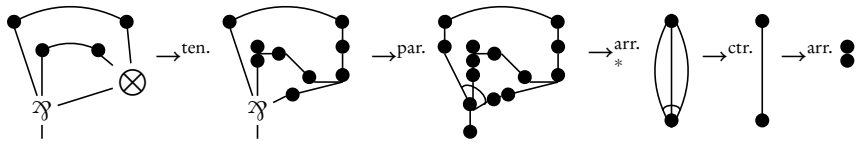
The contraction rules —adapted to the unbounded case of MLL_{u}— are reminded in Figure 2.10. Those rules have a different role: some of them ($\to^{\text{ax.}}$, $\to^{\text{par}}$ and $\to^{\text{ten.}}$) transform a pseudo net into a dotted graph —i.e. they are only a relabelling of the pseudo net—, and the others ($\to^{\text{arr.}}$, $\to^{\text{ctr.}}$) contract the dotted graph, constitutes a rewriting system. Let $\to^{\text{r}}$ be one of those rules, we write $\to^{\text{r}}_{*}$ its transitive reflexive closure.

There is no need to define a relabelling for the cuts, for links eventually transforms into dots, and an edge between two dots —that used to be a cut— will be contracted with a single application of $\to^{\text{arr.}}$.

Theorem 2.2.1 (Contractibility criterion, adapted from Danos [36]). *A pseudo net is a proof net of MLL_u iff after applying $\to^{\text{ax.}}$, $\to^{\text{par}}$ and $\to^{\text{ten.}}$ as many time as possible, the dotted graph obtained can be reduced to a single dot applying $\to^{\text{arr.}}$ and $\to^{\text{ctr.}}$.*

We will in the following write that a pseudo net "reduces" or "contracts" to a dot to express that it is possible to obtain a dot from it by applying the rules of Figure 2.10.

Definition 2.2.7 ($\to_*^{\text{b}}$). The proof nets representing the boolean values, b_0 and b_1, reduce themselves easily into a dot. After an application of $\to_*^{\text{ax.}}$, b_0 reduces itself into a dot by applying the following sequence:



The case for b_1 is the same up to a minor permutation of two wires, and we define $\to_*^{\text{b}}$ as the reduction in one step of all the pieces b_0 and b_1 into dots.

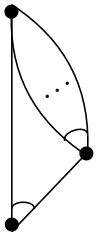
Lemma 2.2.2. *A "piece" whose exits and garbage are connected to a single dot*[9] *reduces to a dot.*

Proof. Let $\mathscr{P}^*$ be the piece $\mathscr{P}$ where all exits and garbage are connected to the same dot. We study the different cases:

b_0 and b_1 After applying $\to^{\text{b}}$ and $\to^{\text{arr.}}$, b_0^* as b_1^* contracts to a dot.

$\mathscr{N}_{\text{eg}}$ After applying $\to_*^{\text{ax.}}$, $\to^{\text{ten.}}$, $\to^{\text{par.}}$, $\to_*^{\text{arr.}}$, $\to^{\text{ctr.}}$ and $\to^{\text{arr.}}$, $\mathscr{N}_{\text{eg}}^*$ reduces to a dot.

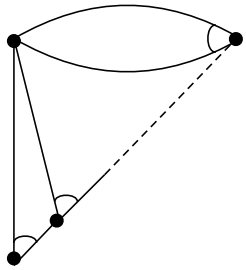
$\mathscr{D}^i_{\text{upl}}$ After applying $\to_*^{\text{b}}$, $\to_*^{\text{ax.}}$, $\to_*^{\text{ten}}$, $\to_*^{\text{arr.}}$, $\to_*^{\text{par.}}$ and then $\to_*^{\text{arr.}}$, $\mathscr{D}^{i*}_{\text{upl}}$ is reduced to



If we apply $\to^{\text{ctr.}}$, $\to^{\text{arr.}}$, $\to^{\text{ctr.}}$ and $\to^{\text{arr.}}$, we get the expected result.

$\mathscr{C}^i_{\text{onj}}$ and $\mathscr{D}^i_{\text{isj}}$ After applying $\to_*^{\text{b}}$, $\to_*^{\text{ax.}}$, $\to_*^{\text{ten}}$, $\to_*^{\text{par}}$ and $\to_*^{\text{arr.}}$, $\mathscr{C}^{i*}_{\text{onj}}$ as well as $\mathscr{D}^{i*}_{\text{isj}}$ is

[9] Note that we are speaking of a structure midway between the pseudo net and the dotted graph. It is not strictly speaking a piece, but this lemma is suited to prove Theorem 2.2.3 later on.



If we apply $i-1$ times the pattern $\to^{\text{ctr.}}$, $\to^{\text{arr.}}$, we get the expected result.

Q.E.D.

Theorem 2.2.3. *Every proof circuit is a Boolean proof net.*

Proof. We will prove this by using the height of the pieces (recall Definition 2.2.6). Given P a proof circuit, every of his piece has a height $n \in \mathbb{N}$, we prove that the set of pieces of height equal or inferior to n contracts itself into a dot.

If $\mathbf{n} = 0$ The set of pieces whose height is 0 is by definition a singleton. His exit and his garbage are connected to the result tensor. There might be other garbage that come from piece of height > 0 connected to the result tensor, but we may safely ignore them. All we want is to fulfill the hypothesis of Lemma 2.2.2, and this is the case after applying $\to^{\text{ten}}$ and $\to^{\text{arr.}}_{*}$, so this piece contracts itself into a dot.

If $\mathbf{n} > 0$ By definition, all pieces of height n are connected to pieces of height strictly inferior to n. By induction, we know that this set of pieces of height inferior to n contracts itself into a single dot. So all pieces of height n have their exits and garbage connected to a single dot, and by Lemma 2.2.2 this set contract itself into a dot.

A proof circuit is made of pieces and of a result tensor, there always exists a n such that all its pieces are of height inferior or equal to n, so it contracts itself into a single dot. The Theorem 2.2.1 gives us that all proof circuits are proof nets.

Regarding the type of the proof circuit P: we know that there exists several derivations that infer it. The only conclusions are:

- unconnected entries of pieces, that is input,
- the result tensor.

We know that the type of an entry of a piece is —up to substitution— $\mathbf{B}^{\perp}$ and that the result tensor recollect on its first port a formula of type $\mathbf{B}$. Hence, any proof circuit can be typed with

$$\vdash \mathbf{B}^{\perp}[A_1], \dots, \mathbf{B}^{\perp}[A_n], \otimes^{1+m}(\mathbf{B}[A], D_1, \dots, D_m)$$

so it is a Boolean proof net. **Q.E.D.**

Patrick Baillot questioned this use of the contractibility criterion. His argument is that as proof circuits are defined inductively, one could directly build its derivation by induction. This argument is perfectly valid, and exhibiting concretely the derivation of a proof circuit would give us almost for free Corollary 2.3.2 to come. Yet, we maintain that this proof technique is the "right" one, because it highlights the graphical structure of our object, its parallel nature. Moreover, it dispenses us from maintaining a high stack of variables for the number of inputs, the fan-in and fan-out of the piece under study, the size of the result tensor, etc. Finally, it seems to us that this proof is in better adequation with the situation, for it testifies of the role of the result tensor to "recollect" all the pending conclusion regarding garbage.

Anyway, this theorem establishes that proof circuits are proof nets, so we can use the same evaluation mechanism, and benefit from Theorem 2.1.1 regarding the parallel cut-elimination. We can define *families of proof circuits* $\mathbf{P} = (\mathsf{P}_n)_{n\in\mathbb{N}}$ as for the Boolean circuits, and the notions of evaluation and representation of a Boolean function can be used "as it" from Definition 2.2.2. As proof circuits are just a special case of Boolean proof nets, we can adopt for them the same Direct Connection Language we introduced in Definition 2.2.3. At last, concerning uniformity, simply substitutes "gate" by "link" and "C" by "P" in Definition 2.1.4 to obtain the suitable definition.

This theorem also mean that a type or a formula can be associated to every edge of a proof circuit. In the case of the entry or the exit of a piece, we will speak of a *transported value*: we know that this edge "carries" a boolean type that can be associated to b_0 or b_1. This notion will turn out to be crucial when we will parse proof circuits with Alternating Turing Machines to determine what are the values carried along the edges without normalising the proof circuit. We refer to Section 2.4 and more precisely to Theorem 2.4.3 for a demonstration of the utility of this notion.

We are now ready to study the computational power of proof circuits: we quickly expose how to compose two proof circuits and then we define a *Proof Circuit Complexity* (**PCC**) class of complexity.

Remark. To *compose two proof circuits* P_1 and P_2, we remove the result tensor of P_1, identify the unconnected exit of P_1 with the selected input of P_2, and recollect all the garbage with the result tensor of P_2. We then label the unconnected entries anew and obtain a proof circuit. The result of this composition is illustrated in Figure 2.11.

Definition 2.2.8 (**PCC**$^{\mathsf{i}}$ (resp. **mBN**$^{\mathsf{i}}$, Mogbil and Rahli [107])). A language $X \subseteq \{0,1\}^*$ belongs to the class **PCC**$^{\mathsf{i}}$ (resp. **mBN**$^{\mathsf{i}}$) if X is accepted by a polynomial-size, $\log^i$-depth uniform family of proof circuits (resp. of Boolean proof nets).

If the pieces are taken from $\mathscr{P}_b$, we write **bPCC**$^{\mathsf{i}}$[10], elsewhere the fan-in of the pieces is unbounded.

Remark. For all $i \in \mathbb{N}$, **bPCC**$^{\mathsf{i}} \subseteq$ **PCC**$^{\mathsf{i}} \subseteq$ **mBN**$^{\mathsf{i}}$.

The first inclusion comes from the fact that every piece of $\mathscr{P}_b$ is a piece of $\mathscr{P}_u$. The second comes from Theorem 2.2.3.

Remark that **PCC**$^{\mathsf{i}} \subseteq$ **bPCC**$^{\mathsf{i+1}}$ with the same argument that proves **AC**$^{\mathsf{i}} \subseteq$ **NC**$^{\mathsf{i+1}}$: every piece of fan-in $n > 2$ can be replaced by $n-1$ pieces of fan-in 2, organized in a "depth-efficient way",[11] thus multiplying the total depth of the proof circuit by a logarithm.[12]

We are now ready to compare proof circuits with other models of computation: first come Boolean circuits, and in the next section we will study the links with Alternating Turing Machines.

[10] See errata.
[11] With the same arrangement that the "unfolding procedure" of Figure 2.2.
[12] Another encoding, proving that **PCC**$^{\mathsf{i}} =$ **bPCC**$^{\mathsf{i}}$, is given in the errata.

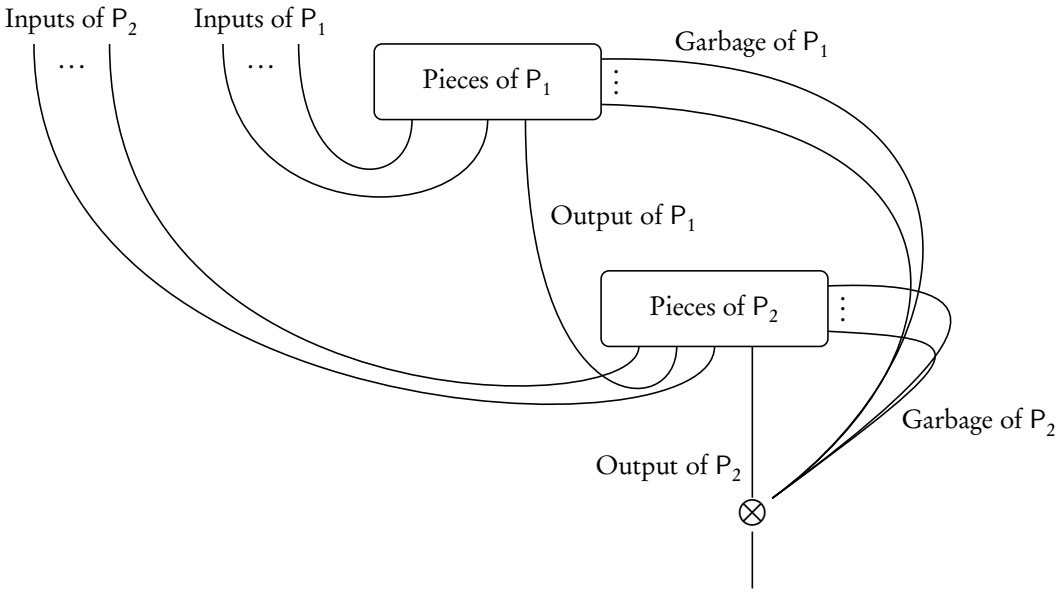


Figure 2.11: Composing two proof circuits

2.3 Correspondence With Boolean Circuits

The first part of this section is an improvement compared to the previous works about Boolean proof nets, the second is just a reminder of those previous works.

Translation from Boolean Circuits to Proof Circuits

By using our proof circuits we prove anew the translation from Boolean circuits to proof circuits, but lower its complexity and by doing so, we extend the results to sublogarithmic classes of complexity. We begin with an intermediate lemma and one of its corollaries.

Lemma 2.3.1. *Let* P *be a proof circuit,* $\mathscr{P}$ *one of its piece with* $n \geqslant 1$ *entries and* $m \geqslant 1$ *exits. Let* $A_{e_1}, \dots, A_{e_n}$ *(resp.* $A_{s_1}, \dots, A_{s_m}$*) be the type associated to its entries (resp. exits), we have for* $j \in \{1, \dots, n\}$

$$d(A_{e_j}) \leqslant d(\boldsymbol{B}) + \max(d(A_{s_1}), \dots, d(A_{s_m}))$$

Proof. We simply consider the different cases and refers to Table 2.8:

- $\mathscr{P} \notin \{b_0, b_1\}$ according to the hypothesis.
- If $\mathscr{P} = \mathscr{N}_{\text{eg}}$, $d(A_{e_1}) = d(A_{s_1})$.
- If $\mathscr{P} = \mathscr{D}^i_{\text{isj}}$ or $\mathscr{C}^i_{\text{onj}}$, then $d(A_{e_1}) = d(A_s)$ and $d(A_{e_j}) = d(\mathbf{B}) + d(A_s)$ for $2 \leqslant j \leqslant i$.

- If $\mathscr{P} = \mathscr{D}^i_{\text{upl}}$, then

$$\begin{aligned} d(A_{e_1}) &= d(\otimes^3((\parr^n(\overleftarrow{A^\perp_{s_n}})\parr(\parr^n(\overleftarrow{A^\perp_{s_n}})), \otimes^n(\overrightarrow{A_{s_n}}), \otimes^n(\overrightarrow{A_{s_n}}))) \\ &= 1 + \max(d((\parr^n(\overleftarrow{A^\perp_{s_n}}))\parr(\parr^n(\overleftarrow{A^\perp_{s_n}}))), d(\otimes^n(\overrightarrow{A_{s_n}}))) \\ &= 2 + \max(d(\otimes^n(\overrightarrow{A_{s_n}}))) \\ &= 3 + \max(d(A_{s_1}), \ldots, d(A_{s_n})) \end{aligned}$$

As $d(\mathbf{B}) = d(\parr^3(\alpha^\perp, \alpha^\perp, (\alpha \otimes \alpha)) = 3$, the lemma is proved.

Q.E.D.

The "less or equal" sign should not misguide the reader: this lemma teaches us that the depth slowly *increase* from the exits to the entries i.e. that the formulae associated to the entries are bigger than the formulae associated to the exists. We should also remark that the depth increases independently from the fan-in of the piece and that duplication has an impact on the depth. But since we cannot "duplicate more than we compute", the impact will only be a multiplication by a constant factor (see proof of Theorem 2.3.3 for a precise statement).

Corollary 2.3.2. *For all proof circuit* $\mathsf{P}_n(\overrightarrow{p})$ *and all* $\overrightarrow{b}$*, the cuts at maximum depth in* $\mathsf{P}_n(\overrightarrow{b})$ *are between the entry of a piece and an input* b_{i_j} *for some* $1 \leq j \leq n$*.*

Proof. The formulae in the garbage are never cut, and in all pieces the cuts that do not connect an entry or an exit ("the inner cuts") are always of depth inferior or equal to cuts connecting the entries.

We just proved that the depths of the cut formulae slowly increase from the exit to the entry, and as the entries that are not connected to other pieces are connected to values, this lemma is proved.

Of course, an entry could also be connected to a constant b_0 or b_1, but we may always consider that two constants b_0 and b_1, eventually duplicated, are part of the input, as for the Boolean circuits.

Q.E.D.

Problem C: Translation from Boolean circuits to proof circuits

`Input`: $\mathrm{L}_{\mathrm{DC}}(\mathbf{C})$ for $\mathbf{C}$ a uniform family of polynomial-size and $\log^i$-depth Boolean circuits over $\mathfrak{B}_u$.

`Output`: $\mathrm{L}_{\mathrm{DC}}(\mathbf{P})$ for $\mathbf{P}$ a uniform family of polynomial-size and $\log^i$-depth proof circuits over $\mathscr{P}_u$, such that for all $n \in \mathbb{N}$, for all $\overrightarrow{b} \equiv b_{i_1}, \ldots, b_{i_n}$, $\mathsf{P}_n(\overrightarrow{b}) \to_{\text{ev.}} b_j$ iff $\mathrm{C}_n(i_1, \ldots, i_n)$ evaluates to j.

Theorem 2.3.3. *Problem C belongs to* $\mathbf{AC}^0$.

Proof. The translation from $\mathbf{C}$ to $\mathbf{P}$ is obvious, it relies on coding: for every n, a first constant-depth circuit associates to every gate of C_n the corresponding piece simulating its Boolean function, i.e. $\wedge^k$ is translated to $\mathscr{C}^k_{\text{onj}}$, $\vee^k$ by $\mathscr{D}^k_{\text{isj}}$ and $\neg$ by $\mathscr{N}_{\text{eg}}$. If the fan-out of this gate is $k > 1$, a $\mathscr{D}^k_{\text{upl}}$ piece is associated to the exit of the piece, and the pieces are connected as the gates. The input nodes are associated to the inputs of P_n. A second constant-depth circuit recollects the single free exit and the garbage of the pieces and connects them to the result tensor. The composition of these two Boolean circuits produces a constant-depth Boolean circuit that builds proof circuits in parallel.

It is easy to check that $\mathscr{C}^k_{\text{onj}}$, $\mathscr{D}^k_{\text{isj}}$ and $\mathscr{N}_{\text{eg}}$ represent $\wedge^k$, $\vee^k$ and $\neg$ respectively, that $\mathscr{D}^k_{\text{upl}}$ duplicates a value k times, and b_0 and b_1 represent 0 and 1 by convention. The composition of these pieces does not raise any trouble: P_n effectively simulates C_n on every input of size n.

Concerning the bounds: the longest path between an entry or a constant and the result tensor goes through at most $2 \times d(\mathsf{C}_n)$ pieces and we know by Corollary 2.3.2 that the increase of the depth is linear in the number of pieces crossed. We conclude that $d(\mathsf{P}_n) \leqslant 2 \times 3 \times d(\mathsf{C}_n)$ and we may also notice that P_n normalises in $\mathcal{O}(d(\mathsf{C}_n))$ parallel steps.

Concerning the size, by simply counting we know that a gate of fan-in n and fan-out m is simulated by two pieces[13] made of $\mathcal{O}(m+n)$ links. As the number of edges in C_n is bounded by $|\mathsf{C}_n|^2$, the size of P_n is at most $\mathcal{O}(|\mathsf{C}_n|^2)$. **Q.E.D.**

Corollary 2.3.4. *For all $i \in \mathbb{N}$*

$$\mathbf{AC}^i \subseteq \mathbf{PCC}^i$$

Note also that if the family $\mathbf{C}$ is over $\mathfrak{B}_b$, $\mathbf{P}$ is made of pieces of $\mathscr{P}_b$, and so $\mathbf{NC}^i \subseteq \mathbf{bPCC}^i$ for $i > 0$, as the translation is in $\mathbf{AC}^0$.

A Boolean circuit with unbounded (resp. bounded) arity of size s is translated to a proof circuit of size quadratic (resp. linear) in s, whereas Terui considers only unbounded Boolean circuits and translate them with Boolean proof nets of size $\mathcal{O}(s^5)$. The previous translation [107, Theorem 2, p. 8] was in $\mathbf{L}$ and so the inclusion could not be applied to sublogarithmic classes. Our translation —thanks mostly to the easier garbage collection— needs less computational power, is more clear and beside lowers the size of the Boolean proof nets obtained.

Simulation of Proof Circuits by Boolean Circuits

Now we will prove the "reverse" inclusion, i.e. that $\mathbf{PCC}^i \subseteq \mathbf{AC}^{i+1}$. This part will not be proved with a translation: we do not build a Boolean circuit family that computes as a proof circuit family. Rather, we define a Boolean circuit family that can *evaluate* any proof circuit, i.e. decide given an input and the Direct Connection Language of its family if it evaluates to b_0 or b_1.

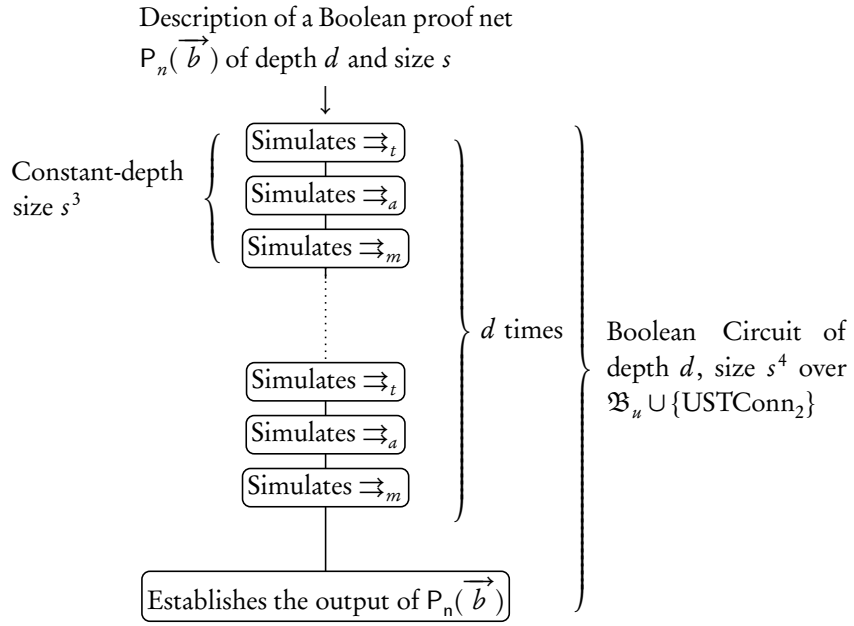
In other words, we define a family of Boolean circuits that is able to normalise in parallel any Boolean proof net, and so they need to perform t-reductions. This operation needs to identify maximal chains of axioms with cuts between them, and that require to solve the problem *USTConn* $_2$ (Problem A). We simply add USTConn$_2$-gates to the basis of the Boolean circuits that will in constant depth identify the t-cuts and reduce them.

Problem D: Simulation from Boolean proof nets to Boolean circuits

`Input`:	$\mathrm{L}_{\mathrm{DC}}(\mathbf{P})$ for $\mathbf{P}$ a polynomial-size, $\log^i$-depth uniform Boolean proof net family.
`Output`:	$\mathrm{L}_{\mathrm{DC}}(\mathbf{C})$ for $\mathbf{C}$ a uniform family of Boolean circuits $\mathbf{C}$ over $\mathfrak{B}_u \cup \{\mathrm{USTConn}_2\}$, such that for all $\overrightarrow{b} \equiv b_{i_1} \dots b_{i_n}$, $\mathsf{P}_n(\overrightarrow{b}) \to_{\text{ev.}} b_j$ iff $\mathsf{C}_n(i_1, \dots, i_n)$ evaluates to j.

Terui managed to perform this simulation thanks to *configurations*, another kind of description of the Boolean proof net that is updated during the course of the normalisation. We refer to [130, 107] for a technical presentation of this tool, and only give the sketch of the proof below. The Figure 2.12 could help the reader to grasp the general structure of the Boolean circuit we are building.

[13] One for the computation, one for the duplication of the result.



Note that given a Boolean proof net in input, d steps of normalisation are enough to obtain the description of its normal form. The t-reduction needs $\mathrm{USTConn}_2$ nodes to be made efficiently.

Figure 2.12: The simulation of a Boolean proof net by a Boolean circuit

For $r \in \{t, a, m\}$, an unbounded fan-in constant-depth Boolean circuit with $\mathcal{O}(|\mathsf{P}_n|^3)$ gates —with $\mathrm{USTConn}_2$ gates to identify chains of axioms if $r = t$– is able to reduce all the r-cuts of P_n in parallel.

A first constant-depth circuit establishes the configuration from $\mathrm{L}_{\mathrm{DC}}(\mathsf{P})$ and constant-depth circuits update this configuration after steps of normalisation. Once the configuration of the normal form of P_n is obtained, a last constant-depth circuit identifies the first proof net connected to the result tensor and establishes if it is b_0 or b_1 —that is to say if the result of the evaluation is *false* or *true*.

As all the circuits are of constant depth, the depth of C_n is linear in $d(\mathsf{P}_n)$. The size of C_n is $\mathcal{O}(|\mathsf{P}_n|^4)$: every circuit simulating a parallel reduction needs $\mathcal{O}(|\mathsf{P}_n|^3)$ gates and in the worst case —if $d(\mathsf{P}_n)$ is linear in the size of the proof circuit— $\mathcal{O}(|\mathsf{P}_n|)$ steps are needed to obtain the configuration of the cut-free Boolean proof net.

Theorem 2.3.5 ([107, Theorem 5, p. 9]). *Problem D is in* **L**.

Corollary 2.3.6. $\mathbf{PCC}^{\mathbf{i}} \subseteq \mathbf{mBN}^{\mathbf{i}} \subseteq \mathbf{AC}^{\mathbf{i+1}}$ *for all* i.

Proof. We know since [136] that *USTConn* is **L**-complete, but there is no proof that a weaker problem as *USTConn* $_2$ belongs to a weaker class of complexity. But since $\mathbf{L} \subseteq \mathbf{AC}^1$, we could replace all $\mathrm{USTConn}_2$ gates by log-depth Boolean circuits, thus yielding $\mathbf{AC}^{\mathbf{i}}(\mathrm{USTConn}_2) \subseteq \mathbf{AC}^{\mathbf{i+1}}$. The previous theorem and Theorem 2.2.3 complete the chain of inclusion. **Q.E.D.**

The simulation is slightly different from the translation: the Boolean circuit does not have to identify the pieces or any mechanism of computation of P_n, but simply to apply $\rightrightarrows_t$, $\rightrightarrows_a$ and $\rightrightarrows_m$ until it reaches a normal form and then look at the obtained value.

Remark. We have focused on sublogarithmic classes of complexity, but we can draw more general conclusions by re-introducing $\mathrm{USTConn}_2$. Terui proved that $\mathrm{USTConn}_2$ could be represented by Boolean proof nets of constant-depth and polynomial size. So we could define "$\mathrm{USTConn}_2$-piece", $\mathscr{P}_{u+s} = \mathscr{P}_u \cup \{\mathrm{USTConn}_2\}$ and $\mathbf{s}+\mathbf{PCC}^{\mathbf{i}}$ as the class of languages recognized by $\log^i$-depth, poly-size proof circuits over $\mathscr{P}_{u+s}$. It would be easy to conclude that for all $i \in \mathbb{N}$, $\mathbf{s}+\mathbf{PCC}^{\mathbf{i}} = \mathbf{AC}^{\mathbf{i}}(\mathrm{USTConn}_2) = \mathbf{mBN}^{\mathbf{i}}$.

2.4 Correspondence With Alternating Turing Machines

Alternating Turing Machines (ATMs) are "the" historical way of characterizing parallel computation. Cook [27], in 1985, *defines* $\mathbf{AC}^{\mathbf{k}}$ as "the class of all problems solvable by an ATM in space $\mathcal{O}(\log n)$ and alternation depth $\mathcal{O}(\log^k n)$." Remark by the way that the "A" of **AC** stands for "alternation".

In this setting, there is no worries to have regarding uniformity, but some slight changes will be made in order to define "families of computation graphs" for ATMs (Definition 2.4.2). That will require that all the runs of an ATM on inputs of the same size have the same "shape" (Definition 2.4.1). This will help us to get closer to the computational behaviour of the proof circuits and to establish bridges.

This other characterization of the computational power of proof circuits has several advantages:

- it helps to grasp in a different manner parallel computation,
- it makes uses of log-space computation, and helps us to introduce some oddities of this calculus,
- it uses the paradigm "ATMs as graphs" that we sketched in Section 1.1, for instance in Definition 1.1.5
- and, last but not least, it refines the bounds we proposed in the previous section.

Recall that we proved, for all $i \in \mathbb{N}$,

$$\mathbf{AC}^{\mathbf{i}} \subseteq \mathbf{PCC}^{\mathbf{i}} \subseteq \mathbf{AC}^{\mathbf{i+1}} \qquad \mathbf{NC}^{\mathbf{i}} \subseteq \mathbf{bPCC}^{\mathbf{i}} \subseteq \mathbf{PCC}^{\mathbf{i}} \subseteq \mathbf{bPCC}^{\mathbf{i+1}}$$

Here we will prove[14] for $j > 1$ and $k > 0$,

$$\mathbf{NC}^{\mathbf{j}} \subseteq \mathbf{PCC}^{\mathbf{j}} \qquad \mathbf{bPCC}^{\mathbf{k}} \subseteq \mathbf{AC}^{\mathbf{k}}$$

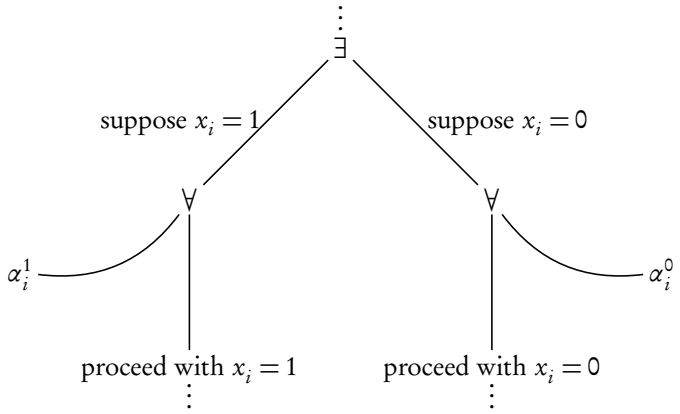
Whereas the first inclusion could be obtained from the previous section, the second inclusion is a novelty and enlightens the position of bounded proof circuits.

Unfortunately, the simulation and translation we will define in the following are quite expansive, and those results do not apply to constant-depth circuits. Anyway, by combining the results of the previous section and the results[15] to come, we obtain for all $i > 1$,

$$\mathbf{NC}^{\mathbf{i}} \subseteq \mathbf{bPCC}^{\mathbf{i}} \subseteq \mathbf{AC}^{\mathbf{i}} \subseteq \mathbf{PCC}^{\mathbf{i}} \subseteq \mathbf{bPCC}^{\mathbf{i+1}}$$

The results in this section are unpublished and new, they are the result of a strong motivation to link together three classical models of parallel calculus. We begin with some remarks on ATMs, we

[14] See errata.
[15] See errata.



Rather than directly reading x_i, the ATM pursue the computation in parallel: on one way it makes as if x_i was 0, on the other it makes as if it was 1. The state α_i^j leads to acceptance iff $x_i = j$, so it discriminate between the correct and the wrong guess.

Figure 2.13: The access to the input in an input normal form ATM

will refine them to obtain more easily the results we expect. We take the liberty to represent the ATM program as a graph, following Definition 1.1.5.

Definition 2.4.1 (Input normal form)**.** An ATM is in *input normal form*,[16] if for every input $x \equiv x_1 \dots x_n$, for all $1 \leqslant i \leqslant n$, all accesses of M to x_i are of the form illustrated in Figure 2.13.

This mechanism is the core of the parallelism of ATMs: here, thanks to the random access to the input, checking if $x_i = j$ has a constant cost, but imagine that we have to perform a computation with a significant cost in terms of space, time or alternation. With this "guess and check" routine, we can safely perform this checking and the course of the computation *in parallel*. Moreover, it is possible to guess values longer than a single bit (cf. Definition 2.4.4).

We will massively use this procedure in the following, but we dwell for now on one of its other advantages: the course of the computation is now relative to the size of the input. This comes from the remark that if M an ATM is in input normal forms and $|x| = |y|$, then $G_{M(x)} = G_{M(y)}$. The difference will rest on the labelling of the graph (recall Definition 1.1.3), leading to acceptance or rejection.

Definition 2.4.2 (Family of computation graphs)**.** Given M an ATM, its *family of computation graphs* $G_M = (G_{M,n})_{n\in\mathbb{N}}$ is such that for every $p \in \mathbb{N}$, $G_{M,p}$ is the computation graph of M on input of size p. We will write $\alpha \vdash_{M,n} \beta$ if there is an edge between α and β in $G_{M,n}$.

The computation graph associated to M on input of size $|x|$, $G_{M,|x|}$, is of depth at most $\mathcal{O}(t(|x|))$ if M is bounded in time by t.[17] If M is bounded in space by s, the number of configurations of M is bounded by $\mathcal{O}(2^{s(|x|)})$ and so is the number of nodes of $G_{M,|x|}$. The bounds on alternation has no consequence on the shape of the graph.

[16] This suppose that our ATMs have random access to their inputs [135, p. 58].
[17] This comes from the simple remark that it is always possible to halt the computation after $\mathcal{O}(t(|x|))$ transitions.

The *depth of a node* is the length of the computation path from the initial configuration to the configuration corresponding to this node. It will be the equivalent of the height of a piece as defined in Definition 2.2.6.

Translation from Alternating Turing Machines to Proof Circuits

Definition 2.4.3 (Translation, inspired from Vollmer [135, p. 63])**.** We define inductively a translation $(\widehat{\cdot})$ from a family of computation graphs G_M to a family of proof circuit $\mathbf{P}$.

Given a size of input n, we proceed as follow:

- For $1 \leqslant j \leqslant n$, there exists $a \in \mathbb{N}$ such that the input x_j is tested a times. If $a = 0$, we simply connect an edge labelled with x_j to a garbage port of the result tensor, otherwise we display a $\mathscr{D}^a_{\text{upl}}$ piece whose entry is labelled with x_j.
- We then consider the sinks (nodes of height 0), which are final configurations or tests. Suppose that the configuration is α and there exists $\beta_1, \dots, \beta_i$ such that for $1 \leqslant k \leqslant i$, $\beta_k \vdash_{M,n} \alpha$.
  - If $g(\alpha) = 0$ (resp. 1), the node is translated with b_0 (resp. b_1),
  - if α tests the if the jth bit of the entry is 0 (resp. 1),[18] the node is translated with the piece $\mathscr{N}_{\text{eg}}$ connected (resp. by directly connecting it) to one of the free output of the $\mathscr{D}^a_{\text{upl}}$ piece connected to x_j.

  In both cases the output of the piece is connected to $\mathscr{D}^i_{\text{upl}}$.
- If the height of the node is $h \geqslant 1$, we proceed as follow: suppose the configuration associated is α. There exists $j, k \in \mathbb{N}^*$, $\beta_1, \dots, \beta_j$ such that for all $1 \leqslant m \leqslant j$, $\beta_m \vdash_{M,n} \alpha$, and $\gamma_1, \dots, \gamma_k$ such that for $1 \leqslant l \leqslant k$, $\alpha \vdash_{M,n} \gamma_l$. By induction, we already know pieces translating $\gamma_1, \dots, \gamma_k$, because their height is strictly inferior to h. We know that $g(\alpha) \notin \{0, 1\}$ since α is not final.

  If $g(\alpha) = \exists$ (resp. $\forall$), the node associated to α is $\mathscr{D}^k_{\text{isj}}$ (resp. $\mathscr{C}^k_{\text{onj}}$), with its k input connected to the translations of $\gamma_1, \dots, \gamma_k$. Its output is connected to $\mathscr{D}^j_{\text{upl}}$, which transmits the value to the pieces translating $\beta_1, \dots, \beta_j$.

We then add a result tensor connected to the output of the translation of the initial configuration and to the garbage. Since every computation graph is finite and presents no cycle, this translation ends. As we simply connected together pieces and added a result tensor, the structure obtained is a proof circuit—hence a proof net— and it has n edges labelled with $x_i, \dots, x_n$.

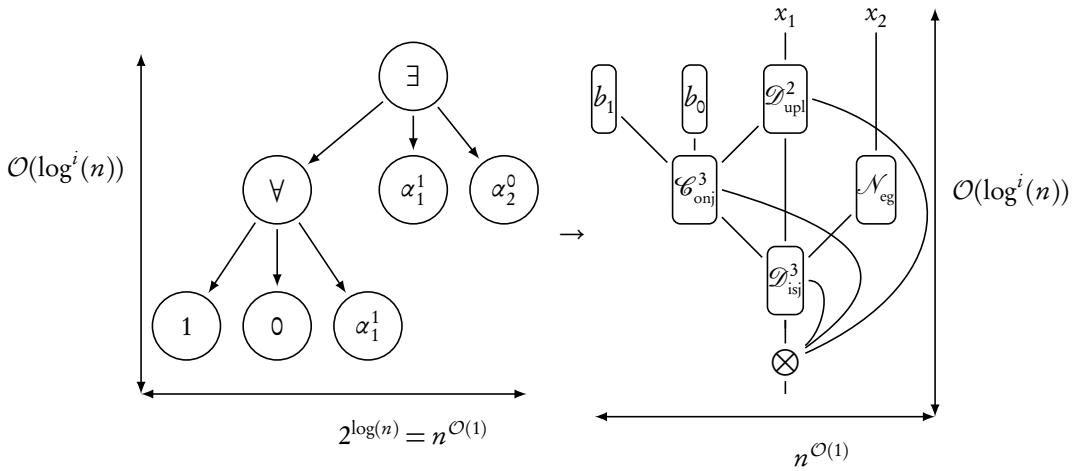
The Figure 2.14 might help the reader to check how simple this translation is. However, even if this algorithm is simple to understand, it has a log-space complexity, as shown below.

Problem E: Translation from ATM to proof circuits

`Input`:	A log-space, $\log^i$-time bounded ATM M.
`Output`:	The Direct Connection Language of a uniform family of $\log^i$-depth proof circuits $\mathbf{C}$ such that for all $n \in \mathbb{N}$, for all $\overrightarrow{b} \equiv b_{i_1}, \dots, b_{i_n}$, $\mathsf{C}_n(\overrightarrow{b}) \to_{\text{ev.}} b_1$ iff $M(i_1, \dots, i_n)$ accepts.

Theorem 2.4.1. *Problem E belongs to* **L**.

[18] Something we wrote α^0_i (resp. α^1_i) in Figure 2.13.



We take the Alternating Turing Machine here to be log-space and $\log^i$-time bounded, and only specify the state type of every configuration (recall Definition 1.1.2). The configuration α_i^j is labelled with 1 iff $x_i = j$, 0 elsewhere. For convenience we represented the edges of the proof circuit from top to bottom, but of course the pieces are connected with cuts.

Figure 2.14: The translation from Alternating Turing Machines to proof circuits

Proof. First, remark that we can always suppose that the ATM is in input normal form: if it is not, there exists an ATM that accepts the same language with the same bounds.[19] Secondly, remember that we sketched in Section 1.1 how a deterministic Turing Machine with logarithmic space could output the computation graph $G_{M(i_1,\dots,i_n)}$ of a Turing Machine M. From that, it is easy to output with logarithmic resources $\widehat{G_{M,n}}$ a proof circuit that will reduce as expected.

Concerning the bounds: we know that the number of configurations of $M(x)$ —and so the size of $G_{M,n}$– is bounded by $2^{\mathcal{O}(\log(n))}$. Every configuration with m predecessors and k successors is translated in $\widehat{G_{M,n}}$ with a piece of size $\mathcal{O}(m+k)$ and a number linear in n of $\mathscr{D}_{\text{upl}}$ piece may be added to duplicate the inputs. As $|\widehat{G_{M,n}}| = \mathcal{O}(|G_{M,n}|) = 2^{\mathcal{O}(\log(n))} = n^{\mathcal{O}(1)}$, $|\widehat{G_{M,n}}|$ is polynomial in n.

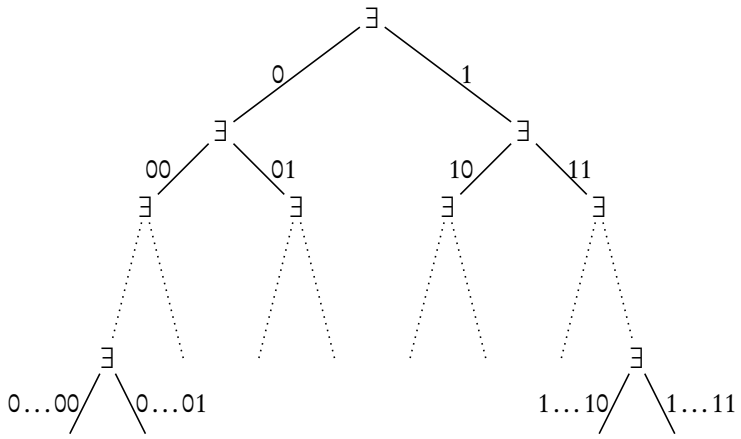
Regarding the depth, the longest computational path form the initial configuration to a final configuration is of length at most $\log^i(n)$. Every node is translated with at worst two pieces, so the depth of $\widehat{G_{M,n}}$ is linear in $\log^i(n)$ by Lemma 2.3.1 **Q.E.D.**

Corollary 2.4.2. *For all $i > 1$,*

$$\mathbf{STA}(\log, \log^{\mathbf{i}}, *) \subseteq \mathbf{PCC}^{\mathbf{i}}$$

Unfortunately this result could not be extended to the sublogarithmic case, as the reduction developed in the previous proof belongs to **L**. This result concerns only $\mathbf{NC}^{\mathbf{2}} = \mathbf{STA}(\log, \log^2, *)$ and above, which has more resources than **L**.

[19] Recall that being in input normal form has a cost in alternation only, but we say nothing regarding this bound in this theorem.



The tree has a depth equal to the size of the guessed value.

Figure 2.15: How a value is guessed by an Alternating Turing Machine

Simulation of Proof Circuits by an Alternating Turing Machine

We will perform here a typical log-space-computation: the size of a Boolean proof net is polynomial in the size of its input, so the Alternating Turing Machine cannot rewrite the description of the Boolean proof net until it reaches a cut-free form, because it simply does not have enough space. Instead, the Alternating Turing Machine will parse, or peruse, its Direct Connection Language in order to find directly the result of the normalisation. We save space by doing that way and are able to write down an algorithm *à la* log-space. This procedure shares a lot with the pointer machinery, a topic that will be the center of Chapter 4.

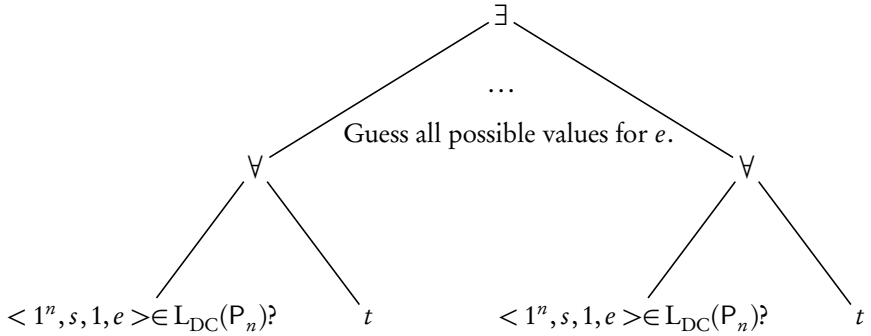
It will rely at the same time on the Direct Connection Language of the Boolean proof nets (recall Definition 2.2.3), on their unifomity, and for the last part on the constrained structure of a proof circuit. We will use the same mechanism as in the guessing of the input by an input normal form ATM to guess information regarding the proof circuit we are trying to normalise.

Definition 2.4.4 (How to guess?)**.** We show how to guess numbers of links; to guess other values of constant size is essentially the same mechanism. Given an input $\overrightarrow{b}$ of size n, it takes place $\log(|\mathsf{P}_n|)$ to write the number of a link in binary, and as $|\mathsf{P}_n| = n^{\mathcal{O}(1)}$, that is to say $\mathcal{O}(\log(n))$.

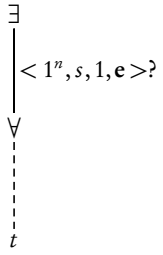
The semantic of the guessing is the following : "there is a 1st bit which can be 0 or 1, there is a 2nd bit which can be 0 or 1, ..., there is a ith bit which can be 0 or 1". The Figure 2.15 shows how it works. It takes time i, space i and has a constant cost in alternation to guess a value of size i.

Once the value we are looking for is guessed, we have to check that it is the *actual* value that we are looking for. In other terms, if we are looking for the number of a link e such that e is connected to the first auxiliary port of the link number s in the Boolean proof net P_n, we have to make sure that $< 1^n, s, 1, e > \in \mathrm{L}_{\mathrm{DC}}(\mathsf{P}_n)$. This question —once e is guessed— takes time $\mathcal{O}(\log(n))$ on a deterministic Turing Machine to be answered, since our proof circuits are uniform.

We present in Figure 2.16 the composition of the guessing and of the checking by the uniformity machine. Remark that an important amount of useless work is done: this is typical of a log-space



We guess all possible values for e by guessing all the binary string of length $\log(|\mathsf{P}_n|)$. Then, for every of them, we test if it is the value we are looking for, by asking the uniformity machine if $< 1^n, s, 1, e > \in \mathrm{L_{DC}}(\mathsf{P}_n)$.
Among all the values guessed for e, only one will satisfy this belonging, for the first auxiliary port of the link numbered e in P_n is connected to only one other link. So all the branches except one will reject, and the computation will go on with the actual value for e.
We will write in Figure 2.17 this routine as



Where e is in bold because it is the value we are looking for, and a dashed line means that the uniformity machine is been asked. A constant number of values may be guessed and checked at the same time, with only one alternation, as long as there is enough space to store the values.

Figure 2.16: A shorthand for the "guess and check" routine

computation. As we do not have enough space to memorize the number of the links, we just calculate them again and again, when we need them.

Now that we defined how to obtain with efficiency some information regarding a Boolean proof net, we have to show how an ATM can use them wisely to establish the outcome of its normalisation. To try to establish an algorithm with logarithmic resources for any Boolean proof net is a bit naive, for the complexity of cut-elimination is quite high: it has been proven by Mairson and Terui [100] that the cut elimination procedure is **P**-complete for IMLL—the intuitionist fragment of MLL. As they write it [100, p. 24], "This result contradicts folkloric intuitions that **MLL** proofnets could be normalized in logarithmic space—that is, with only a finite number of pointers into the proofnet, presumably following paths in the style of the geometry of interaction."

We proved it anyhow for a constrained fragment of the proof nets of $\mathrm{MLL_u}$: the proof circuits. In this setting, we always know "what to expects", as proof circuits are built following the simple pattern of composing pieces. To each entry and exit of a piece, we may associate a value, i.e. the edges labelled e and s in Table 2.8 may be typed with **B**, and they will "transport" a boolean value. Our algorithm will find for every piece, starting from the exit of the proof circuit, what are the values carried by those

edges. We cannot associate to "the value carried is 0" rejection and to "the value carried is 1" acceptance, because we rely on tests, on guess and checks, where rejection means "you did not made the right hypothesis, I am not even able to perform this computation".

So we define a recursive function f that given the number of a link and an expected value, leads to acceptance if and only if the edge connected to the principal port of that link will "transport" the expected value. That distinguishes "rejection cause the test fails" and "rejection because the value is 0".

Definition 2.4.5 (f : number of a link $\times$ value expected $\to \{\mathbf{accept}, \mathbf{reject}\}$)**.** Given s the number of a link and $v \in \{0,1\}$, f will accept iff the proof net connected to the principal port of s will normalise to b_v. This function is recursive and we give in Figure 2.17 a representation of the computational graph of an ATM computing this function: it massively uses the "guess and check" procedure and the shorthand we introduced in Figure 2.16. It does not cover the case when s is connected to an input: in this case, simply accept if this input is b_v, reject elsewhere.

We give some explanation on how this function computes: it takes as input $v \in \{0,1\}$ —the "expected" value— and s —the number of a link, that may safely be identified with the edge connected to its principal port, i.e. with the edge labelled s in Table 2.8, the exit of a piece.

Then f proceeds by cases, with $\overline{v} = 0$ if $v = 1$, 0 elsewhere:

1. If s is the output of a piece b_i ($i \in \{0,1\}$) accepts if $i = v$, rejects elsewhere.
2. If s is the output of a piece $\mathscr{N}_{\text{eg}}$ whose entry is connected to the link number e, do as $f(\overline{v}, e)$.
3. If s is the output of a piece $\mathscr{D}^i_{\text{upl}}$ whose entry is connected to the link number e, do as $f(v, e)$.[20]
4. If s is the output of a piece $\mathscr{C}^2_{\text{onj}}$ whose entries are connected to the links number e_1 and e_2, and if $v = 1$ (resp. $v = 0$), universally (resp. existentially) branch on $f(v, e_1)$ and $f(v, e_2)$.
5. If s is the output of a piece $\mathscr{D}^2_{\text{isj}}$ whose entries are connected to the links number e_1 and e_2, and if if $v = 1$ (resp. $v = 0$), existentially (resp. universally) branch on $f(v, e_1)$ and $f(v, e_2)$.

We prove that this procedure ends thanks to an inductive proof on the depth and by using Lemma 2.3.1: in the $\mathscr{N}_{\text{eg}}$ case, the call to f is performed on a link at same depth. In the $\mathscr{D}^i_{\text{upl}}$ case, the call is performed at a greater depth. In the $\mathscr{C}^2_{\text{onj}}$ and $\mathscr{D}^2_{\text{upl}}$ cases, two calls are made: one at the same depth, the other at superior depth. In the b_i case, no recursive call is made.

As the $\mathscr{N}_{\text{eg}}$ pieces are limited to the entries (recall Definition 2.2.5), we can consider this case only as a constant factor. Except for this case, each iteration performs at least one call to f with a link of depth superior to the previous call. So the procedure ends, and in the worst case $\mathcal{O}(d(\mathsf{P}_{|n|})$ calls are performed. That f leads to accept with input v, s iff the link v will transport the value v is just a matter of common sense.

The following theorem and its proof are wrong, see errata.

Theorem 2.4.3. *For all $i > 0$.*

$$\mathbf{bPCC}^{\mathbf{i}} \subseteq \mathbf{STA}(\log, *, \log^{\mathbf{i}})$$

Proof. We build our ATM M as follows:

[20] A remark should be made about this treatment of the duplication: there is no sharing. Every time we want to know the output of a duplication, we test its input. So to say, the circuit is "splited" and treated as it was a tree.

$$i \in \{2,3\}, \overline{i} = \begin{cases} 2 \text{ if } i = 3 \\ 3 \text{ elsewhere} \end{cases} \quad v \in \{0,1\}, \overline{v} = \begin{cases} 0 \text{ if } v = 1 \\ 1 \text{ elsewhere} \end{cases} \quad \text{case1} = \begin{cases} i = 2 \text{ and } v = 0 \\ i = 3 \text{ and } v = 1 \end{cases} \quad \text{case2} = \begin{cases} i = 2 \text{ and } v = 1 \\ i = 3 \text{ and } v = 0 \end{cases}$$

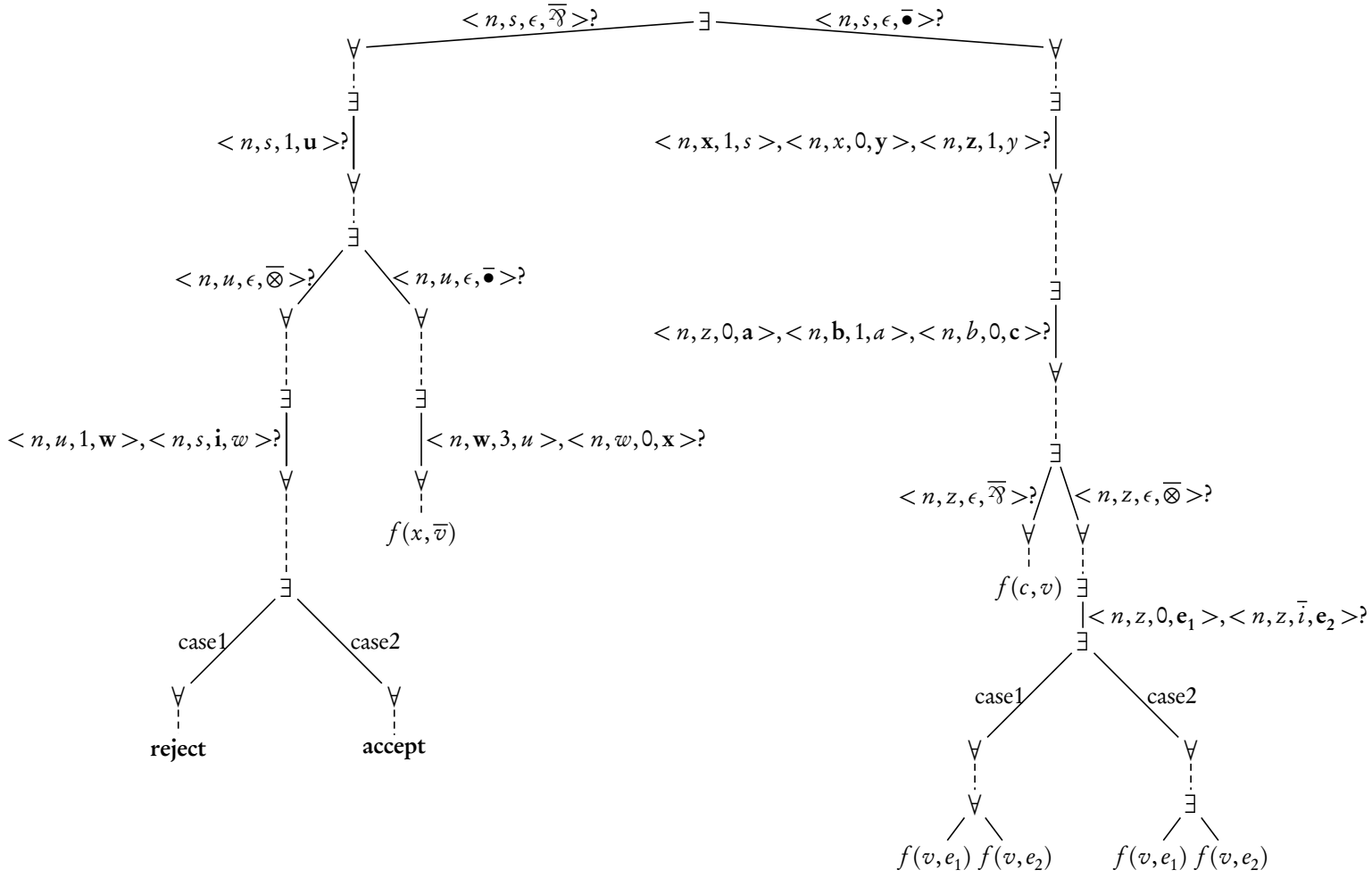


Figure 2.17: A function to compute the value carried by an edge

- It takes as input $L_{DC}(\mathbf{P})$ for $\mathbf{P}$ a uniform, polynomial-size, $\log^i$-depth proof circuit family over $\mathscr{P}_b$, and an input $\overrightarrow{b} = i_1, \ldots, i_n$.
- It identifies the number of the result tensor of P_n.[21]
- It guesses the number of the link connected to the first auxiliary port of the result tensor, and it runs f with this value and 1.

It should be clear that M accepts iff $\mathsf{P}_n(\overrightarrow{b}) \to_{\text{ev.}} b_1$, and stops rejecting elsewhere.

Regarding the bounds:

Space At every moment M has only to store a constant number of addresses of links. We can at each transition forget every data except the expected value and the number of the link under treatment. A logarithmic amount of space is sufficient to store those values.

Alternation Each call to f uses a constant number of alternations, and $\mathcal{O}(d(\mathsf{P}_n))$ calls are made. As P_n is of depth $\log^i(n)$, $\mathcal{O}(\log^i(n))$ calls are made.

Q.E.D.

Two remarks should be done:

- This algorithm cannot be extended as it to Boolean proof nets, even if they are uniform. As our pieces are "normed", the ATM knows what to expect, and so we can encode in the algorithm itself the behaviour to adopt. A Boolean proof net in all generality is a too complex object to be treated the same way.
- We work in the bounded case for complexity reasons. Our algorithm could work with just a slight modification in the unbounded case, but the bounds would be problematic. We would need to introduce a routine that guess and checks the arity of every $\mathscr{D}_{\text{isj}}$ and $\mathscr{C}_{\text{onj}}$ pieces. But that would re-introduce linearity in our algorithm, for the ATM needs to establish the number of the $\otimes$ corresponding to the $i+1$th entry before establishing the number of the $\otimes$ corresponding to the ith entry. As in the worst case, the fan-in of the piece is relative to the size of the proof circuit, we obtain a bound on time relative to the *size* of our Boolean proof net.

Results and perspectives

What have we done?

We recalled a reliable way of representing Boolean values and Boolean functions with MLL_u proof nets. We then restated a depth-efficient way of computing with them, letting them able to be normalised by Boolean circuits, thus proving (Corollary 2.3.6), for all $i \in \mathbb{N}$,

$$\mathbf{PCC}^i \subseteq \mathbf{AC}^{i+1}$$

[21] That is to say to find the only $\otimes$-link with its principal port unconnected. That can be done in parallel, or more simply we can by convention says that the first link described in the Direct Connection Language of the proof net is the result tensor.

We then constrained the structure of the Boolean proof nets by introducing proof circuits and gave an $\mathbf{AC}^0$-reduction from them to Boolean circuits, establishing —even when $i = 0$ (Corollary 2.3.4)

$$\mathbf{AC}^i \subseteq \mathbf{PCC}^i$$

We adapted the classical representation of Alternating Turing Machines in Boolean circuits toward proof circuits, and that gave us (Corollary 2.4.2), a new proof that for $k > 1$

$$\mathbf{NC}^k \subseteq \mathbf{PCC}^k$$

At last, we deeply used the parallelism of Alternating Turing Machines to prove that they were able with log-space resources to obtain the cut-free form of any proof circuits with bounded fan-in. By combining Theorem 2.4.3 with the equality $\mathbf{AC}^i = \mathbf{STA}(\log, *, \log^i)$ for all $i \geqslant 1$, we obtain[22], for $k > 1$

$$\mathbf{bPCC}^k \subseteq \mathbf{AC}^k$$

We managed to lighten the simulation of the Boolean functions by proof nets and extended previous results to sublogarithmic classes of complexity. We also managed to build strong gateways between Boolean proof nets and ATMs, and even if those results does not apply to the very small classes, it shed a new light on the links between logarithmic space computation and proof nets.

What could be done?

This work could be pushed further in several directions:

1. Girard [49, p. 8] was very enthusiast regarding non-commutative Linear Logic, "which should play a prominent role in the future". This fragment does not forbid exchange, but constrains them by allowing only circular permutations, that is to say planar proof nets. This particular framework led to numerous fruitful studies, including specific correctness criterion [103]. Her, our modelization of computation used permutations, something that was at a hidden level in the sequent calculus. We made explicit those exchanges thanks to the proof nets, but other approaches has been developed, for instance by Andreoli, Maieli, and Ruet [3]. Although the main interest of our setting is to compute in parallel, it could be interesting to see how computation may be represented by cyclic linear logic, which has a part of non-determinism in it.

2. Non-determinism was also simulated with the Boolean proof nets by introducing a part of the additive fragment by Mogbil [105]. The simplicity of the proof circuits allows to write quite directly "by hands" proof nets and to compute with them. By introducing small doses of non-determinism in the pieces, it could be possible to understand precisely what is at stake in this setting. Some promising results between proof nets and non-deterministic were obtained by Maurel [101]. But non-deterministic computation in this setting looks more like a juxtaposition of deterministic computation than true non-determinism.

3. Semi-Unbounded-Fanin $\mathbf{AC}$ ($\mathbf{SAC}^k$) is the class of decision problems solvable by a family of $\log^k(n)$-depth circuits with unbounded-fanin $\vee$ and bounded-fanin $\wedge$ gates. Of course, we take as a constraint that negations are only allowed at the input level. It was introduced by Borodin et al. [20] and

[22] See errata.

proven that this class was closed under complement for every $k > 0$. We know also that $\mathbf{SAC}^1 = \mathbf{LOGCFL/poly}$ since the work of Venkateswaran [134]. To investigate in this direction could offer a very nice framework to grasp non-uniform classes of complexity, recently sketched by Mazza [102].

CHAPTER 3

Computing in the Hyperfinite Factor

Contents

In this chapter, we are going to define another model of computation inspired from the Linear Logic, and more precisely from the Geometry of Interaction (GoI). Although the inspiration comes from Linear Logic, the objects we are going to manipulate are purely mathematical. This comes from the GoI program, and more precisely from its fifth version[57], that tells us that operators in the II_1 hyperfinite von Neumann algebra describe totally the computational behaviour of proofs. Others variants of the GoI programs were already used with complexity concerns, and we present some of them in Section 3.1, before making clear that our approach here is a total novelty.

We build patiently in Section 3.3 the representation of binary words with matrices starting from proofs, and the ELL representation in particular. From this construction, it is rather easy to build a representation of integers in the hyperfinite factor, and then to define programs that will interact with

them (or, as we call them later, *observations*), in Section 3.4. We take the time in the second subsection of Section 3.4 to explain and motivate GIRARD's choices, and to dwell on the difference of the approach we developed with Thomas SEILLER.

Once the representation of computation in this setting is defined, we fix a parameter (the *normative pair*) and constraint the program on a mathematical level, with the positiveness of the coefficients and the 1-norm. We then define a particular normative pair, which relies heavily on permutations. The group of finite permutations over $\mathbb{N}$ is well-known to the Linear Logic community for it was used to interpret the multiplicative fragment of Linear Logic.

The last part is devoted to prove that the computation that takes place in the Hyperfinite factor can be grasped with logarithmic space. This can be achieved thanks to a complex lemma presented in Section 3.5, whose purpose can easily be explained as the trip back from hyperfinite factor to finite matrices. It is the first step toward a *mathematical* characterisation of the log-space computation. It borrows some concepts to the Implicit Computatonal Complexity (ICC) approach, but we will stick in this chapter to a mathematical level.

This setting was first proposed by GIRARD [59] and enhanced by AUBERT and SEILLER[10, 11] recently. This chapter borrows a large part of the content of those works, but presents it in a innovative ways, with examples and a more pedagogical development (see in particular Section 3.2, when the construction is explained step-by-step for the unary representation of the integers).

This chapter is to be read at the light of its dual Chapter 4, where the computational behaviour of the observations in a particular normative pair is compared to pointer machines. It is largely in debt to SEILLER's precious intuitions, his patience and well understanding of the von Neumann algebras. Even if it uses some quite complex mathematical tools, as the von Neumann algebra or the crossed-product, we preferred to define those technical tools in the Appendix A. The technical details are not what is important in this innovative construction, so we felt free to give rather global ideas to explain it. Yet, some useful element are reminded in the second moment of Section 3.1.

3.1 Complexity and Geometry of Interaction

Past approaches

Linear Logic always had a tight link to complexity, thanks to its resource-awareness. Geometry of Interaction, while giving a dynamics semantics to Linear Logic, preserved that concern and gave new tools to express cut-elimination. The famous "execution formula" [48, p. 222] defined a way to express strong normalisation as the nilpotency of a set of operators.

This first approach gave rise to numerous semantics for various fragments of Linear Logic, much of them being sensitive to the cost of normalization in terms of time: the studies of IMELL [33] and ELL [14] being probably the two most famous examples.This first paper use some games and ICC techniques to build an interpretation for IMELL proofs that reflects their time-complexity. The second paper is a bit closer to our work, for it uses an algebra of clauses to interpret proofs of ELL. In this setting, proof nets are represented with operators, and a novel way of executing them —independent from the nilpotency— is defined and proven to preserve some complexity properties. Context semantics is one of the model of the GoI and was used to prove some interesting computational complexity properties [31].

On a more syntactical level (even if the GoI tries to "scramble" the distinction between syntax and semantics), the GoI was often understood as a novel approach that subsumes the proof nets into a set

of permutations. Stated differently, the type of the proof net is not the center of our object anymore: all the information is contained in the *paths*. This perspective gave rise to some hope that proof nets could be parsed with pointers and described with log-space (but we saw in the previous chapter, page 50, how Mairson and Terui [100] proved that it was only "a rumour", provided of course that $\mathbf{L} \neq \mathbf{P}$). Anyway, this gave numerous works where Linear Logic was linked to complexity and apprehended thanks to various kind of token machines.

Last but not least, the bridges between Linear Logic and operator algebra led to think that Linear Logic could be a place to express quantum computation.Indeed, quantum mechanics is expressed by mathematical formulations in a complex separable Hilbert space[1]—as for the GoI. Since the pioneer work by Selinger [122] that linked λ-calculus and quantum computation,[2] numerous attempts have been made to link those two world by the entanglement of von Neumann algebras, Girard [54] being the first. Quantum complexity was not left aside, and quantum circuits were proven to have a closed link with Linear Logic by Dal Lago and Faggian [32].

Complexity with operator algebra

We define in this chapter a new way of linking Linear Logic and complexity *via* von Neumann algebras by using operators as representations of algorithms. The multiplication of the representation of an integer and the representations of a program will be the execution of the program, so there is no need to define transition functions. Despite its high degree of mathematical abstraction, the model of computation we are going to define is a low-level one: every operation can be understood in its most modest details.

This attempt can be considered as the first success toward a purely mathematical way of defining a model of computation and complexity classes[3] It uses operators acting on an infinite-dimensional (separable) Hilbert space, but can largely be understood in terms of matrices. It does not refer to any exterior measure of complexity and takes its place in the fast-growing field of ICC.

We give below some intuitions regarding the choice of those mathematical tools. The reader might refer to Appendix A for a short reminder of the algebraic and topological elements needed to grasp the usefull part of von Neumann algebras. It also contains some useful references and bibliographical indications. We simply give in the following some insights of the mathematical properties of the von Neumann algebras. We will not explore the definitions and properties of von Neumann algebras, factors, hyperfiniteness and crossed-product in this chapter. We use only the core of those notions in our construction, but they do not really play an important role. Although those tools won't be used before Section 3.4, we remind of them now, so that we won't have to interrupt the quite subtle construction that will follows.

We begin by recalling that a Banach Algebra —a $\mathbb{C}$-vector space enriched with the structure of an algebra, a norm, and complete for that norm— endowed with an involution is a C*-algebra. A von Neumann algebra $\mathfrak{M}$ is a C*-algebra of operators[4] that can be understood —depending on the reader's preferences— as

[1] This is not the place to develop such a complex subject, but the interested reader may find some useful hints in the textbook of Nielsen and Chuang [110].

[2] One may also have a look at another pionner work [133, Section 9].

[3] Ugo Dal Lago kindly made me notice that "any model of computation is defined mathematically", so that this was a bit pretentious. Yet, it seems to us that the framework we are going to define can be fully understood from a mathematical point of view, in an innovative way.

[4] An operator is a continuous or equivalently bounded linear maps between Hilbert spaces. The set of operators on a Hilbert space $\mathbb{H}$ will be denoted $\mathcal{B}(\mathbb{H})$.

a C*-subaglebra of $\mathcal{B}(\mathbb{H})$, which is closed for one of the topology among the weak operator, the strong, the ultrastrong or the ultraweak topologies;

a C*-algebra that have a predual, i.e. there exists a Banach algebra such that $\mathfrak{M}$ is its dual;

a C*-algebra equal to its bi-commutant, i.e. the set of elements of $\mathcal{B}(\mathbb{H})$ which commute with the elements that commute with themselves.

The beauty of von Neumann algebras comes partially from this plurality of definitions, which allowed transfers from a mathematical field to another.

The *hyperfinite* von Neumann algebras are of particular interest to us, because we can approximate every of their operators with finite matrices. We can also quotient the von Neumann algebras to study only their *factors*, which can be classified according to their *projections*. For most types hyperfinite factors are unique, and it is the case for the factor we will use in the following: the type II_1 hyperfinite factor.

As we have already said, permutations are a natural way of interpreting the multiplicative fragment of Linear Logic. Yet, to express the power of exponentials, we need to be able to express infinite objects. Operator algebra is quite an obvious choice, for it generalizes finite permutation thanks to *partial isometries*. A partial isometry u establishes an isometric bijection between its domain —the closed subspace associated with the projection u^*u— and its image —the closed subspace associated with the projection uu^*.[5] Partial isometries have a tight bound with *projections*, i.e. operators which give an orthonormal projection of $\mathbb{H}$ onto a closed subspace of $\mathbb{H}$. The permutation of the elements of an infinite separable Hilbert space generalise quite elegantly the permutations.

Permutations will be re-integrated in this setting on a second level, thanks to the crossed product. This operation internalizes the action of a group —in this setting, the group of finite permutations over $\mathbb{N}$— into the von Neumann algebra. This operation won't modify our object —which will remain the II_1 hyperfinite factor— but it will allow us in the next chapter to study easily the computational behaviour of our object.

3.2 First Taste: Tallies as Matrices

We will begin by explaining how integers in the unary notation ("tally sticks") can be represented as matrices *via* the Curry-Howard correspondence. It has two interests: to see the computation from a dynamics point of view, and to represent any proof (hence, integer) inside the same object. As the construction is quite complex, we felt it necessary to begin with the unary representation, simpler to grasp. We won't use it in the following, so the reader quite used to this idea may safely skip this section and go straightforward to Section 3.3. Nevertheless, most of the explanations we make in the following are still valid when applied to the binary representation.

The representation of integers as matrices goes through the following journey:

1. Represent integers with λ-terms (Church encoding).
2. Represent λ-terms with derivations (Curry-Howard isomorphism).
3. Represent derivations with proof nets ("Parallel syntax").

[5] A restatement of [58, p. 400].

$$\dfrac{\dfrac{}{\vdash X_4, X_5^\perp}\,\text{ax.} \qquad \dfrac{\dfrac{}{\vdash X_2, X_3^\perp}\,\text{ax.} \qquad \dfrac{}{\vdash X_{m-1}, X_1^\perp}\,\text{ax.}}{\vdash X_1^\perp \otimes X_2, X_3^\perp, X_{m-1}}\,\otimes}{\vdash X_1^\perp \otimes X_2, X_3^\perp \otimes X_4, X_5^\perp, X_{m-1}}\,\otimes$$

$$\vdots$$

$$\dfrac{\dfrac{}{\vdash X_{m-2}, X_m^\perp}\,\text{ax.} \qquad \vdash X_1^\perp \otimes X_2, \ldots, X_{m-3}^\perp, X_{m-1}}{\vdash X_1^\perp \otimes X_2, \ldots, X_{m-3}^\perp \otimes X_{m-2}, X_{m-1}, X_m^\perp}\,\otimes$$

$$\dfrac{\vdots}{\vdash X_1^\perp \otimes X_2, \ldots, X_{m-3}^\perp \otimes X_{m-2}, X_{m-1} \parr X_m^\perp}\,\parr$$

We gives in all generality the derivation of the unary representation of $n \in \mathbb{N}^*$, remember that we set $2n + n = m$.

Figure 3.1: The derivation of a unary integer, without contraction rule.

4. Represent proof nets with matrices (Geometry of Interaction).

We will in the following make one step further: to embed those matrices in the type II_1 hyperfinite factor. Those unary integers are not efficient from a complexity point of view: they are introduced only for pedagogical reasons, to show how data can be represented by matrices. A first version of this encoding —more in the "GoI V-style"— was developed [57, section A.3, p. 31], and then simplified [59, p. 251]. The version we propose here rather comes from this latter simplification, but has the pretension to be a bit more explanatory. We will first present it without the contraction rule, and introduce this subtlety in the next subsection.

First representation of the unary integers, without the contraction rule

Without the contraction rule, $n \in \mathbb{N}$ can be represented as a proof of

$$\underbrace{(X \Rightarrow X), \ldots, (X \Rightarrow X)}_{n \text{ times}} \vdash X \Rightarrow X$$

This representation is not exactly the Church encoding of numerals, that represent n as the higher-order function that maps any function f to $f \circ f \circ \ldots \circ f$. Due to the absence of contraction, that could allow us to get sure that *the same* function is applied n times,[6] this encoding should rather be seen as the function that takes n functions $f_1, \ldots f_n$ and returns the composition of them $f_n \circ \ldots \circ f_1$, whose corresponding λ-term would be $\lambda f_1 \ldots \lambda f_n \lambda x \cdot f_n(\ldots(f_1(x)\ldots)$. We know that they all have the same type $X \Rightarrow X$, but we have no insurance whatsoever that they are identical.

To represent $n \in \mathbb{N}$, this encoding uses $2n + 2$ times the literal X, and we set for convenience that $2n + 2 = m$. We add in the following a subscript to help us to differentiate the m different occurrences of X, but keep in mind that the one-sided Linear Logic proof that corresponds to n represented in Figure 3.1 contains a single literal.[7] The bits are "chained" with $\otimes$, and $\parr$ represents the end of the chain: it will be at the same time its beginning and its end, thus yielding a circular representation. But we have time for that and should first remark that the poverty of this proof is even more clear if we represent it as a proof net, as in Figure 3.2.

[6] This approach will be developed in the following subsection. We felt it necessary to begin with the an even simpler case.
[7] Otherwise the axiom for instance introducing at the same time X_2 and $X_3^\perp$ would be false.

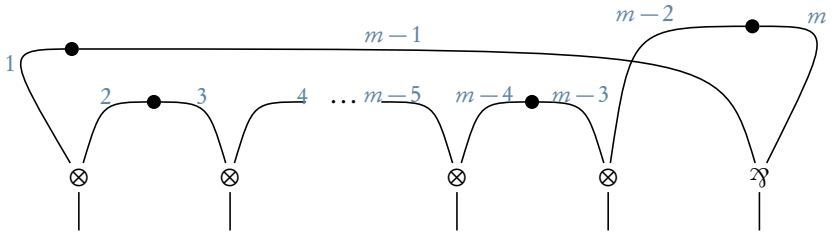


Figure 3.2: Unaries as proof net

Nevertheless, we can then represent for all $n \in \mathbb{N}^*$ its proof by a square matrix L_n,[8] with one column and one row per occurrence of X —i.e. of dimension $m \times m$— in the corresponding proof. We interpret the axiom rule

$$\frac{}{\vdash X_j, X_i^\perp}\ \text{ax.}$$

as the symmetric exchange between i and j (i.e. $\mathrm{L}_n[i,j] = 1$ iff there is an axiom between $X_i^\perp$ and X_j) and we define τ to be the anti-diagonal matrix of dimension 2. For all $n \in \mathbb{N}^*$, we interpret the proof corresponding to n with a $m \times m$ matrix:

$$\tau = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \mathrm{L}_n \coloneqq \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 & 0 & 1 & 0 \\ 0 & & & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & & \tau & \dots & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \dots & & & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & & \tau & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Of course this representation is not satisfactory, for the dimension of the matrix itself —as for the type— tells us the represented integer. It inflates the space needed to encode the information without giving any dynamics. But this is the first step toward a more efficient representation. We add in the following the first enhancement: to allow the contractions.

Maybe we should before that remark that L_n is symmetric and that its diagonal is empty. Some more interesting properties are yet to come when we will have defined the associated projections, but we keep this further step for the following subsection.

Unary integers with the contraction rule

In λ-calculus, the Church representation yields that $n \in \mathbb{N}^*$ is encoded by

$$\lambda f\, \lambda x \cdot \underbrace{f(f(\dots(f}_{n \text{ times}}\, x)\dots)$$

[8] The notation we use regarding matrices is summed up in Definition A.2.5.

and the Curry-Howard correspondence tells us that the construction of this λ-term corresponds to the derivation of the type $(X \Rightarrow X) \Rightarrow (X \Rightarrow X)$.

To get this representation, we allow the contraction rule.[9] Any proof of conclusion

$$\underbrace{(X \Rightarrow X), \ldots, (X \Rightarrow X)}_{n \text{ times}} \vdash X \Rightarrow X$$

can easily be transformed into a proof of conclusion $(X \Rightarrow X) \vdash (X \Rightarrow X)$, where X occurs only four times. This representation is isomorphic to the representation of the integers in system F,[10] which associate to $n \in \mathbb{N}^*$ a proof of $\forall X (X \Rightarrow X) \Rightarrow (X \Rightarrow X)$.

In Linear Logic, this yelds a proof of $?(X^\perp \otimes X), !(X^\perp \otimes X)$, which is obtained from the sequent derivation of the previous subsection (in Figure 3.1) as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash X_1^\perp \otimes X_2, \ldots, X_{m-5}^\perp \otimes X_{m-4}, X_{m-3}^\perp \otimes X_{m-2}, X_{m-1} \parr X_m^\perp}\,?}{\vdash ?(X_1^\perp \otimes X_2), \ldots, X_{m-5}^\perp \otimes X_{m-4}, X_{m-3}^\perp \otimes X_{m-2}, X_{m-1} \parr X_m^\perp}\,?}{\dfrac{\dfrac{\vdots}{\vdash ?(X_1^\perp \otimes X_2), \ldots, ?(X_{m-5}^\perp \otimes X_{m-4}), X_{m-3}^\perp \otimes X_{m-2}, X_{m-1} \parr X_m^\perp}\,?}{\vdash ?(X_1^\perp \otimes X_2), \ldots, ?(X_{m-5}^\perp \otimes X_{m-4}), ?(X_{m-3}^\perp \otimes X_{m-2}), X_{m-1} \parr X_m^\perp}\,?}{\vdash ?(X_1^\perp \otimes X_2), \ldots, ?(\mathbf{X}_1^\perp \otimes \mathbf{X}_2), X_{m-1} \parr X_m^\perp}\,\text{ctr.}$$

$$\dfrac{\dfrac{\vdots}{\vdash ?(\mathbf{X}_1^\perp \otimes \mathbf{X}_2), X_{m-1} \parr X_m^\perp}\,!}{\vdash ?(\mathbf{X}_1^\perp \otimes \mathbf{X}_2), !(\mathbf{X}_3 \parr \mathbf{X}_4^\perp)}\,!$$

The literals in the conclusion are in bold and with a fresh labelling, to dwell on the fact that they do not stand only for themselves: $?(\mathbf{X}_1^\perp \otimes \mathbf{X}_2)$ is made of "strata" of contracted occurrences, and $!(\mathbf{X}_3 \parr \mathbf{X}_4^\perp)$ can be reproduced, it is the result of the computation. The Figure 3.3 is the introduction of the contractions in the proof net presented in Figure 3.2.

Informally, if we give n times 1 stick, we are able to obtain 1 pack of n sticks. What a low-level of computation ! But a least we are able —and this is the precise interest of Linear Logic— to handle resources within proofs.

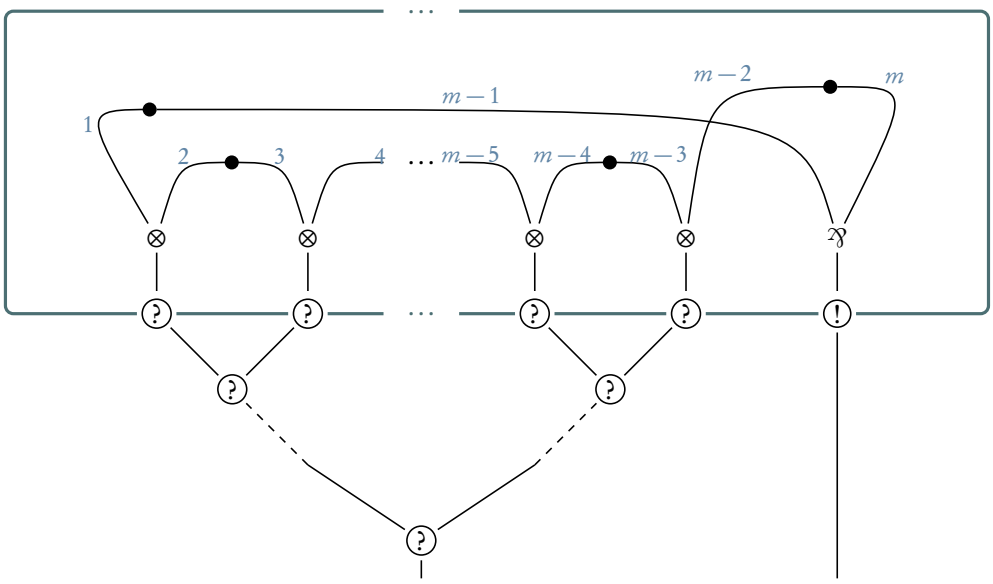
Contrary to the previous subsection, the type does not reflect all alone the integer encoded, it just reflects the mere fact that *an* integer has been encoded. More precisely, each tally stick $X_i^\perp \otimes X_j$ lives in a different *slice*, and all those slices are "compressed" into one single occurrence of $?(\mathbf{X}_1^\perp \otimes \mathbf{X}_2)$.

In this case, n occurrences of $X^\perp \otimes X$ are contracted, it should be more clear if we precise that the original proof of

$$X_1 \Rightarrow X_2, X_3 \Rightarrow X_4, \ldots X_{m-1} \Rightarrow X_{m-2} \vdash X_{m-1} \Rightarrow X_m$$

[9] As Girard likes to write, contraction is the "fingernail of infinity" [49, p. 12].

[10] The system F is the second-order λ- calculus, discovered independently by Jean-Yves Girard and John C. Reynolds, one may easily find a precise definition in textbooks [62, Chap. 11].



This proof net permutes the applications of ! and ctr. compared to the derivation we gave. The ! rule needs all the occurrences in the context to be under a ? modality, hence the "box", a classical tool we did not introduce.

Figure 3.3: Unaries as proof net, with contraction

can be renumbered as

$$X_{1.1} \Rightarrow X_{2.1}, X_{1.2} \Rightarrow X_{2.2}, \ldots X_{1.n} \Rightarrow X_{2.n} \vdash X_{3.0} \Rightarrow X_{4.0}$$

And thus we contract all the occurrences of $X_{1.y}$ into $\mathbf{X}_1$, and all the occurrences of $X_{2.y}$ into $\mathbf{X}_2$.

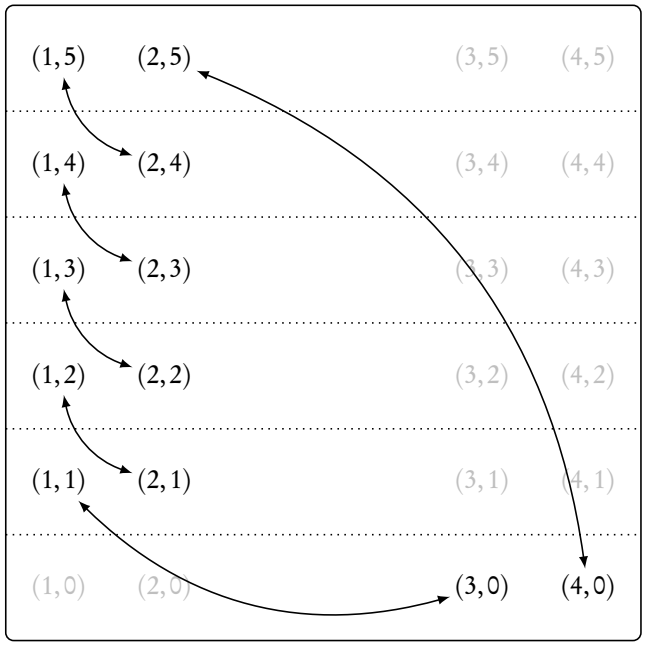
We are beginning to introduce two-dimensional considerations: we have 4 literals, and each of them contains $n+1$ "floors"[11]: the "ground floor"[12] is reserved for the result, and every tally stick lives on a different storey i.e. on a floor that is not the ground floor.

Note that nothing in the Logic prevents from displaying the first stick and the result on the same floor: we preferred to separate them, in order to clearly distinguish the beginning ($X_{3.0}$) and the end ($X_{4.0}$) of the integer represented.

We borrow from Seiller's thesis [121, p. 251] a quite explanatory figure, representing the proof of the integer 5:

[11] Although "slice" is the consecrated term for this notion, it is a bit misleading in this precise case. Contraction is handled as superimposition of copies of occurrences, as in the classical GoI framework, but that does not justify why $(3,0)$ and $(4,0)$ lives in different slices.

[12] To whom we give the number 0, as in the British convention.



It would need too much material to explain in details this figure, which actually represent a *thick graph* in the graph-theoretical representation introduced by Seiller [121]. Two literals side-by-side are in black if they are "active" (present in the proof), and for each of them there is a ⇒-connective between them in the conclusion (before the contractions). The arrows represent the axiom links: remember for instance that $X_{3.0}$ and $X_{1.1}$ were respectively numbered with X_{m-1} and X_1 previously, and refer to Figure 3.1.

Remark that $X_{1.0}$, $X_{2.0}$, $X_{3.x}$ and $X_{4.x}$ for $0 < x \leqslant n$ are necessary empty, because no contracted occurrence will ever rest on those "slots".

Now for the matricial representation, we still interpret the axiom rule as the symmetric exchange between two occurrences, but we split the information between several coefficient of M, a 4×4 matrix, as the number of occurrences of $\mathbf{X}$. Each of these coefficient is a $(n+1) \times (n+1)$ matrix, as the number of "floors" in the proof.

So far, we did not really took into account the case where $n = 0$, for it is at the same time quite different and not really interesting. Simply notice that 0 is quite an exception, for its proof is simply an axiom and a weakening, so there is only one symmetric exchange, and we represent the proof of 0 with

$$\mathrm{M}_0 \coloneqq \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The general case —i.e. $n \neq 0$— is a bit more complex, let us remember that the axiom links are between

$X_{3.0}$ and $X_{1.1}$,	$\dots$
$X_{2.1}$ and $X_{1.2}$,	$X_{2.(n-1)}$ and $X_{1.n}$,
$X_{2.2}$ and $X_{1.3}$,	$X_{2.n}$ and $X_{4.0}$.

We will represent any integer $n \in \mathbb{N}^*$ with the same canonical 4×4 matrix M:

$$\mathrm{M} \coloneqq \begin{pmatrix} 0 & v_n & u_n & 0 \\ v_n^{\mathrm{t}} & 0 & 0 & w_n^{\mathrm{t}} \\ u_n^{\mathrm{t}} & 0 & 0 & 0 \\ 0 & w_n & 0 & 0 \end{pmatrix}$$

The matrix M is such that the entry $\mathrm{M}[i,j]$ in the ith row and jth column encodes the exchanges between occurrences labelled with $X_{i.x}$ and $X_{j.y}$ for all $0 \leqslant x \leqslant n$ and $0 \leqslant y \leqslant n$. So for instance, $\mathrm{M}[i,i]$ is necessarily empty, and $\mathrm{M} = (\mathrm{M})^{\mathrm{t}}$ because the exchanges are symmetric.

The coefficients of M —u_n, v_n and w_n— are $(n+1) \times (n+1)$ matrices that encodes [13] respectively the exchanges between $X_{1.x}$ and $X_{3.x}$ —the beginning of the integer—, $X_{1.x}$ and $X_{2.x}$ —the body of the integer—, $X_{4.x}$ and $X_{2.x}$ —the end of the integer. Their size vary with n the integer represented, but they always have the same shape:

$$u_n \coloneqq \begin{pmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix} \qquad v_n \coloneqq \begin{pmatrix} 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix} \qquad w_n \coloneqq \begin{pmatrix} 0 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{pmatrix}$$

What is really interesting is their properties as matrices. We need before that to define the projections $\pi_{n,i}$ of dimension $(n+1) \times (n+1)$ with $0 \leqslant i \leqslant n$.

$$\pi_{n,i} \coloneqq \begin{array}{c} \begin{matrix} & & i & & \end{matrix} \\ \begin{pmatrix} 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ 0 & \dots & 1 & \dots & 0 \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 \end{pmatrix} \end{array} \begin{matrix} \\ \\ i \\ \\ \\ \end{matrix}$$

Remark that for all i, $\pi_{n,i}^2 = \pi_{n,i}^{\mathrm{t}} = \pi_{n,i}$ and that $\sum_{0 \leqslant j \leqslant n} \pi_{n,j} = \mathrm{Id}_{n+1}$.

[13] We take as a convention that those matrice's rows and columns are numbered from 0 to n, whereas M was numbered from 1 to 4. This may seem illegitimate, but this help us to conserve the same numbering of the literals and floors.

Then one can remark that u_n, v_n and w_n are such that:

$$\begin{aligned} u_n u_n^{\mathrm{t}} &= \pi_{n,1} \\ v_n v_n^{\mathrm{t}} &= \pi_{n,2} + \ldots + \pi_{n,n} \\ w_n w_n^{\mathrm{t}} &= u_n^{\mathrm{t}} u_n = \pi_{n,0} \\ w_n^{\mathrm{t}} w_n &= \pi_{n,n} \end{aligned}$$

It will become of greatest importance when we will represent u_n, v_n and w_n as operators: instead of the mere transposition we will conserve those equalities when we use the conjugate-transposition $(\cdot)^*$ and that will establish that those three operators are partial isometries.

For the time being, and that will conclude this first part, let us remark that the following equalities hold:

$$u_n \pi_{n,i} = \begin{cases} u_n & \text{if } i = 0 \\ 0 & \text{elsewhere} \end{cases} \qquad \pi_{n,i} u_n = \begin{cases} u_n & \text{if } i = n \\ 0 & \text{elsewhere} \end{cases}$$

$$v_n \pi_{n,0} = v_n \pi_{n,n} = \pi_{n,0} v_n = \pi_{n,n} v_n = 0$$
$$\pi_{n,n} v_n = v_n \pi_{n,n-1}$$
$$v_n = \pi_{n,2} v_n + \ldots + \pi_{n,n} v_n = v_n \pi_{n,1} + \ldots + v_n \pi_{n,n-1}$$

$$w_n \pi_{n,i} = \begin{cases} w_n & \text{if } i = n \\ 0 & \text{elsewhere} \end{cases} \qquad \pi_{n,i} w_n = \begin{cases} w_n & \text{if } i = 0 \\ 0 & \text{elsewhere} \end{cases}$$
$$w_n = u_n^{\mathrm{t}} (v_n^{\mathrm{t}})^{-1}$$

$$\pi_{n,i+1} = v_n \pi_{n,i} v_n^{\mathrm{t}} \text{ for } 1 \leqslant i < n$$

It is not of major interest for the time being to understand them precisely, they are just here to help the reader to grasp that we are able to go back and forth between projections and those coefficients: their sum "is a sort of circular permutation of $n+1$ projections $\pi_{0,n}, \ldots, \pi_{n,n}$", they "organise a 'round-trip'" (Girard [59, pp. 253–254]).

They make it possible to cycle through the encoded representation. They are the core of the representation, for we represent the data by its dynamic, and only its dynamics has to be preserved by the embedding in the hyperfinite factor. We won't develop that point here, all interesting equalities will be developed in Section 3.3 for the binary representation.

This representation, with the contraction, is a gain in term of uniformity,[14] for every integer is represented with a 4×4 matrix, but if we look closely, those matrices are equivalent to $4(n+1) \times 4(n+1)$ matrices. To obtain a true uniformity, we will embed those matrices in the II_1 hyperfinite factor $\mathfrak{R}$, as detailed in Proposition 3.4.1 for the binary representation: the operators will represents integers no matter their size, but conserve their properties.

We postpone this embedding for the case that interests us the most: the representation of the binary integers.

[14] Uniformity being understood in this case as "one object to represent them all".

3.3 Binary Integers

We make one last time the construction from integers to matrices in this section, but it is to prepare the ground for Section 3.4, which pushes the construction one step further by embedding this representation in the hyperfinite factor.

In the previous chapter, we encoded binary words as a sequence of proof nets encoding bits. The information whereas the bit was 0 or 1 was encoded in the planarity of the proof net—in the presence or the absence of a permutation, or an exchange, between two literals.

Here we will proceed differently: we will encode binary words into a single proof net and the planarity will play no role. The information whereas the ith bit is a 0 or a 1 will rest in the order of the contractions. Every binary word of arbitrary size will have the same type $?(X\otimes X^{\perp}), ?(X\otimes X^{\perp}), !(X^{\perp}\otimes X)$. Whereas the conclusion is always the same, the number of application of the contraction rule and their principal formulae will differ from one proof to another.

We identify binary words with integers, using the "binary strings to integers" classical bijection:

For $a_k a_{k-1}\dots a_2 a_1$ a binary list of length k, $a_j \in \{0,1\}$ for all $0 \leqslant j \leqslant k$,

$$a_k a_{k-1}\dots a_2 a_1 \to (a_k \times 2^{k-1}) + (a_{k-1} \times 2^{k-2}) + \dots + (a_2 \times 2^1) + (a_1 \times 2^0) = \sum_k (a_k \times 2^{k-1})$$

Here we won't have a bijection, because for instance $0a_k \dots a_1$ will be accepted as a representing the same integer as $a_k \dots a_1$. Anyway, we can always uniquely associate an integer to every binary word, and we will sometimes commit the small abuse of speaking of *the* representation of an integer as a binary word.

The reader who read the previous section should not get confused: in the unary representation, the size of the representation *was* the integer represented, so we spoke of n as well as the integer and as its size. This setting is of course different, for several integers $n_1, \dots, n_{2^k}$ can all be represented with k bits. We will always assume in the following that the length of the binary word under study is $k \neq 0$.

Binaries as proofs

Following the Church-Howard representation, the binary word $a_k \dots a_1$ for $a_1, \dots, a_k \in \{0,1\}^k$ is represented with the λ-term $\lambda f_0 \lambda f_1 \lambda x \cdot f_{a_k}(f_{a_{k-1}}(\dots(f_{a_1} x)\dots))$. Given two function f_0 —*a.k.a.* "append a 0 to the list"— and f_1 —*a.k.a.* "append a 1 to the list"—, the binary word is encoded in the order of application of those two functions. Of course, we need only one copy of each of those functions: in Linear Logic, this will correspond to contractions.[15]

In ELL,[16] binary lists are typed with

$$\forall X\ !(X \multimap X) \multimap (!(X \multimap X) \multimap !(X \multimap X))$$

To a binary integer corresponds a proof of the —annotated— Linear Logic sequent

$$\forall X \Big(?(X(0_i) \otimes X(0_o)^{\perp}) \parr \big(?(X(1_i) \otimes X(1_o)^{\perp}) \parr !(X(S)^{\perp} \parr X(E)) \big) \Big) \tag{3.1}$$

[15] Yes, to contraction, for the proof is built upside-down and is one-sided: so what could normally look like a duplication —"gives me one function and I will apply it several times"— is in reality a contraction—"I used the function several times but it will appear only once in the conclusion".

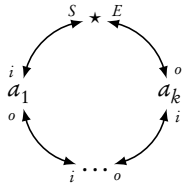
[16] This notion and all the references were introduced p. 18.

We should take some time to explain those annotations: $X(0_i) \otimes X(0_o)$ plays the role of adding a 0 at the end of the list, and it has an *input* (the original list) and an *output* (the list with a 0 appended), respectively at the left and the right of the $\otimes$-connective. The other "function" $X(1_i) \otimes X(1_o)$ plays the same role for "append a 1 to the list".

The last part of this formula $-X(S)^\perp \parr X(E)-$ is the "exterior interface": it has also an output (the "S" part, as in *start*) and an input (the "E" part, as in *end*).[17] It acts like a marker of the beginning *and* the end of the word. We use the symbol $\star$ to represent this marker, from which we may *in one step* go to the end or to the beginning of the word.

Each element is connected by its output node to its successor's input node and by its input node to its predecessor's output node. The successor of the last element and the predecessor of the first element is $\star$.

So a binary word will be circular, we represent this fact by using the notation $\star a_k \dots a_1 \star$. The bits are numbered in the reverse order: the leftmost bit is the last (the end of the word, the kth one) and the rightmost bit is the first (the start of the word). By convention, the $k+1$th and the 0th bits are $\star$. Or, more graphically, and with labels regarding the input / output structure:



One could see above that S can be read as $\star$'s output and E as $\star$'s input.

The size and the structure of the derivation of such a list change a lot in some cases: the empty list uses the weakening rule twice, and the proof representing a list containing only 0s (resp. only 1s) uses a weakening to introduce $?(X(1i) \otimes X(1o)^\perp)$ (resp. $?(X(0i) \otimes X(0o)^\perp)$). The reader may refer to earlier works [121, 10, p. 254–257] to see the proofs and the associated graphs in those cases.

We won't take them into account, for we may safely assume that we are always working with binary words containing 0 and 1. Any sequence $11\dots11$ can safely be replaced by $011\dots11$, and we can always suppose we don't need any representation of 0.

The derivation of the sequent (3.1) is given in all generality in Figure 3.4. We labelled the variables in order to distinguish between the different occurrences of the variable X. This is necessary because we need to keep track of which formulae are principal in the contraction rule. Without this information, a lot of proofs could not be distinguished.

Example 3.3.1 (Eleven (1/3))**.** We propose in Figure 3.5 an example of a binary word $\star 001011 \star$ represented as a proof net and as a circular word. This sequence —which is one of the representation of eleven— will be a running example in this first part of this chapter and should help the reader to grasp the general shape of the construction.

There is one last representation that we chose not to develop: the *interaction graphs* invented by Seiller [121, 120, 119]. This approach is enlightening as it allows us to see the Geometry of Interaction as a combinatorial object. Yet we do not develop this setting here, for it would need to introduce too many objects at the same time. A first taste of this representation was given page 65: we refer the reader

[17] "Start" and "End" rather than the more classical "Head" and "Tail", mainly because in this setting, the head *is* the tail, and that could be misleading.

We give below the derivation of the proof corresponding to $\star a_k a_{k-1} \dots a_1 \star$, which could be read backward and with the labels as $S a_{i1} a_{o1} \dots a_{i(k-1)} a_{o(k-1)} \dots a_{ik} a_{ok} E$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash X(S)^\perp, X(a_{i1})}\,\text{ax.} \quad \dfrac{}{\vdash X(a_{o1})^\perp, X(a_{i2})}\,\text{ax.}}{\vdash X(S)^\perp, X(a_{i1}) \otimes X(a_{o1})^\perp, X(a_{i2})}\,\otimes \quad \dfrac{}{\vdash X(a_{o2})^\perp, X(a_{i3})}\,\text{ax.}}{\vdash X(S)^\perp, X(a_{i1}) \otimes X(a_{o1})^\perp, X(a_{i2}) \otimes X(a_{o2})^\perp, X(a_{i3})}\,\otimes}{\vdots}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash X(S)^\perp, X(a_{i1}) \otimes X(a_{o1})^\perp, \dots, X(a_{i(k-1)}) \otimes X(a_{o(k-1)})^\perp, X(a_{ik})} \quad \dfrac{}{\vdash X(a_{ok})^\perp, X(E)}\,\text{ax.}}{\vdash X(S)^\perp, X(a_{i1}) \otimes X(a_{o1})^\perp, \dots, X(a_{i(k-1)}) \otimes X(a_{o(k-1)})^\perp, X(a_{ik}) \otimes X(a_{ok})^\perp, X(E)}\,\otimes}{\vdash X(a_{i1}) \otimes X(a_{o1})^\perp, \dots, X(a_{i(k-1)}) \otimes X(a_{o(k-1)})^\perp, X(a_{ik}) \otimes X(a_{ok})^\perp, X(S)^\perp \parr X(E)}\,\parr}{\vdash ?(X(a_{i1}) \otimes X(a_{o1})^\perp), \dots, ?(X(a_{i(k-1)}) \otimes X(a_{o(k-1)})^\perp), ?(X(a_{ik}) \otimes X(a_{ok})^\perp), !(X(S)^\perp \parr X(E))}\,!}{\vdots}\,\text{ctr.}$$

$$\dfrac{\dfrac{\dfrac{\vdash ?(X(0_i) \otimes X(0_o)^\perp), ?(X(1_i) \otimes X(1_o)^\perp), !(X(S)^\perp \parr X(E))}{\vdash ?(X(0_i) \otimes X(0_o)^\perp), (?(X(1_i) \otimes X(1_o)^\perp) \parr !(X(S)^\perp \parr X(E)))}\,\parr}{\vdash (?(X(0_i) \otimes X(0_o)^\perp) \parr (?(X(1_i) \otimes X(1_o)^\perp) \parr !(X(S)^\perp \parr X(E))))}\,\parr}{\vdash \forall X\ (?(X(0_i) \otimes X(0_o)^\perp) \parr (?(X(1_i) \otimes X(1_o)^\perp) \parr !(X(S)^\perp \parr X(E))))}\,\forall$$

The formulae $?(X(a_{ic}) \otimes X(a_{oc})^\perp)$ and $?(X(a_{id}) \otimes X(a_{od})^\perp)$ are contracted iff $a_c = a_d$. After the applications of the contraction rule, we have only two occurrences of $?(X(a_i) \otimes X(a_o)^\perp)$ for $a \in \{0, 1\}$, gathering the occurrences of 0 and the occurrences of 1.

Figure 3.4: Binary integers as derivations

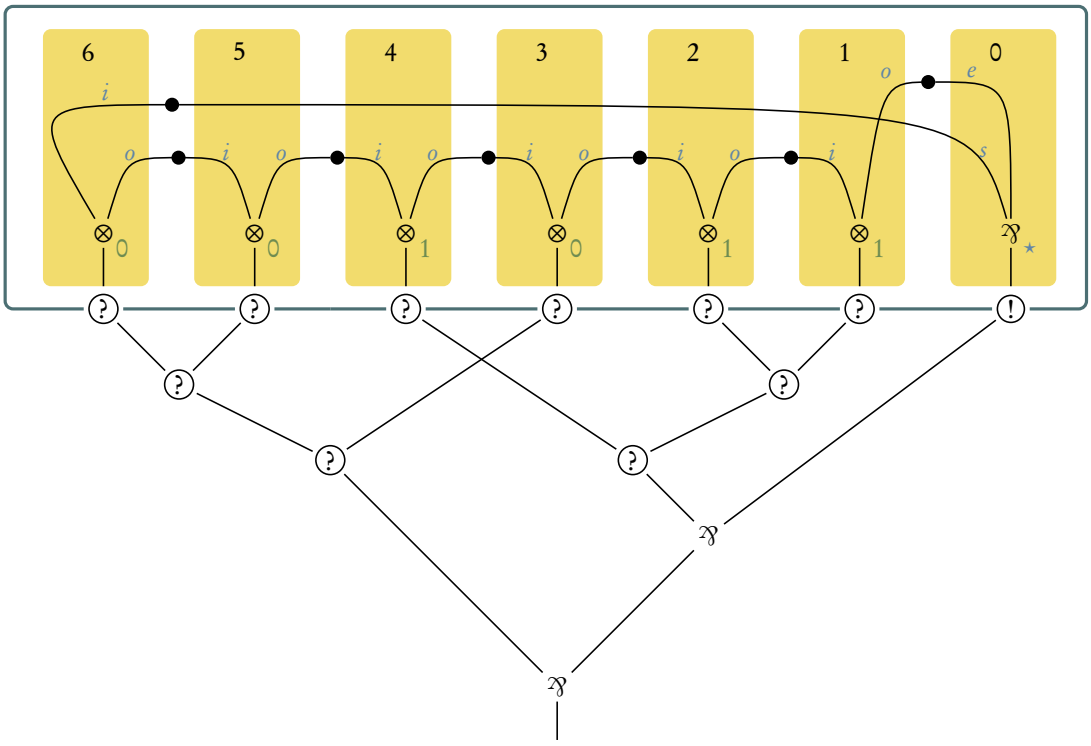
to this subsection or to an earlier version of this work [10] for a better insight on binary words as interaction graphs.

In the forthcoming setting, proof nets are not our main objects, we will use matrices as representations of (proofs of) integers. Yet, it would probably be wiser to avoid the non-planarity of our proof nets, since in this case it only depends on a convention regarding where is the input and output of our word. That would require a careful renaming of the variable corresponding to S and E in this work, but would give another property to our representation and probably make it gain in clarity. We did not make this small change —suggested by Paul-André Melliès— since our next representation is not sensitive to that property of proof nets.

Binaries as matrices

The idea to represent proofs as matrices comes from the simple remark that all cut-free proofs in LL of same conclusion differs essentially only on their axioms. The reader who looked at the previous section should be used to this idea, but it is not mandatory to understand this motivation to grasp the following construction.

Any binary word will be represented by the same 6×6 matrix where one row and one column is dedicated to a literal of Equation 3.1. We recall that those literals handle the information about the input and output structure of 0, 1 and $\star$. So for instance the coefficient at the intersection of the "input of 0" row and the "output of 1" will contains the positions in the sequence where a 1 is followed by a 0.



We indicate the "binary values" (0 and 1) next to the logical connectives ⊗, but their values comes from the order of the contractions.

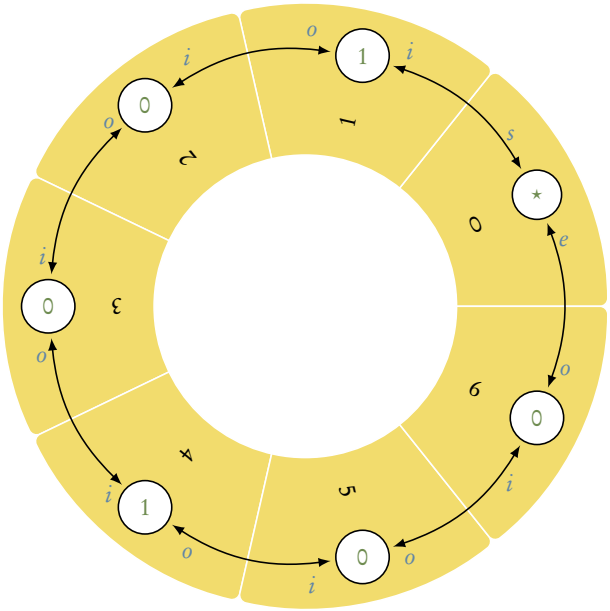


Figure 3.5: The binary sequence ⋆001011⋆ as a proof net and as a circular string

Definition 3.3.1 (Matricial representation of a list)**.** Given a binary representation $\star a_k \dots a_1 \star$ of an integer n, it is represented by M_n, a 6×6 block matrix[18] of the following form:

$$\mathrm{M}_n = \begin{pmatrix} \overbrace{0}^{0} & \mathrm{l}_{00} & \overbrace{0}^{1} & \mathrm{l}_{10} & \overbrace{\mathrm{l}_{S0}}^{\star} & 0 \\ \mathrm{l}_{00}{}^{t} & 0 & \mathrm{l}_{01}{}^{t} & 0 & 0 & \mathrm{l}_{0E}{}^{t} \\ 0 & \mathrm{l}_{01} & 0 & \mathrm{l}_{11} & \mathrm{l}_{S1} & 0 \\ \mathrm{l}_{10}{}^{t} & 0 & \mathrm{l}_{11}{}^{t} & 0 & 0 & \mathrm{l}_{1E}{}^{t} \\ \mathrm{l}_{S0}{}^{t} & 0 & \mathrm{l}_{S1}{}^{t} & 0 & 0 & 0 \\ 0 & \mathrm{l}_{0E} & 0 & \mathrm{l}_{1E} & 0 & 0 \end{pmatrix} \begin{matrix} \}0 \\ \}1 \\ \}\star \end{matrix}$$

whose entries are $(k+1) \times (k+1)$ matrices[19] defined —for $u, v \in \{0,1\}$— by:

- $\mathrm{l}_{uv}[i, i+1] = 1$ iff $a_i = u$ and $a_{i+1} = v$, $\mathrm{l}_{uv}[a,b] = 0$ otherwise;
- $\mathrm{l}_{uE}[0,1] = 1$ iff $a_k = u$, and $\mathrm{l}_{uE}[a,b] = 0$ otherwise,
- $\mathrm{l}_{Sv}[k,0] = 1$ iff $a_1 = v$, and $\mathrm{l}_{Sv}[a,b] = 0$ otherwise.

Remark that —as the output of a bit is always connected to the input of its direct neighbor— $\mathrm{l}_{uv}[a,b] = 1$ implies that $b = a+1$. This was for the body of the word. Regarding the start (resp. the end) of the word, only one among l_{S0}, l_{S1} (resp. l_{0E}, l_{1E}) is different from 0: only one value among 0, 1 may be connected to S (resp. E), that is, the sequence necessarily starts (resp. ends) with a 0 or with a 1.

A sketch in Figure 3.6 should help the reader to grasp some properties of this matrix, as well as its inner structure.

Example 3.3.2 (Eleven (2/3))**.** Our example $\star 001011 \star$ is represented by M_{11} whose coefficients[20] are as follows:

- $\mathrm{l}_{00}[a,b] = 0$ except for $\mathrm{l}_{00}[5,6] = 1$
- $\mathrm{l}_{01}[a,b] = 0$ except for $\mathrm{l}_{01}[3,4] = 1$
- $\mathrm{l}_{10}[a,b] = 0$ except for $\mathrm{l}_{10}[2,3] = \mathrm{l}_{10}[4,5] = 1$
- $\mathrm{l}_{11}[a,b] = 0$ except for $\mathrm{l}_{11}[1,2] = 1$
- $\mathrm{l}_{1E}[0,1] = 1$, $\mathrm{l}_{S0}[6,0] = 1$, and all the other coefficients of l_{1E} and l_{S0} are 0
- $\mathrm{l}_{0E} = \mathrm{l}_{S1} = 0$

This representation of binary integers is still "non-uniform": the size of the matrix —$6(k+1) \times 6(k+1)$— depends on the size k of the binary string representating the integer.

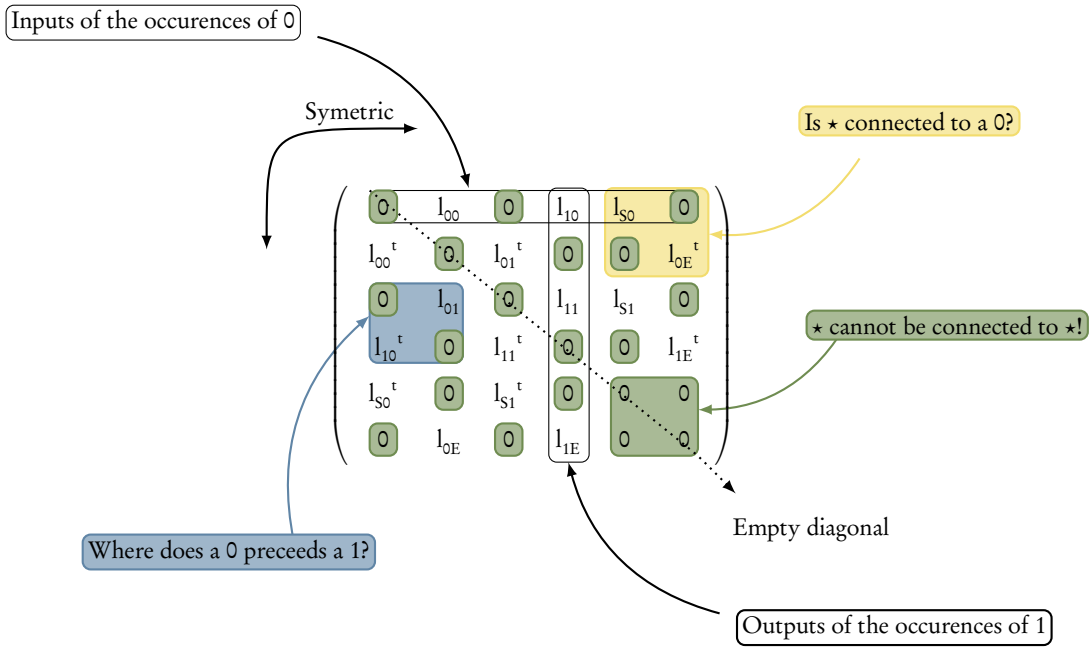
To get a uniform representation of integers, we still need to embed the coefficients of M_n in $\mathfrak{R}$. But before that, we should dwell on some properties of this matricial representation.

[18] The embraces are only pedagogical.
[19] Matrices whose rows and columns are numbered from 0 to k, for convenience.
[20] Remember that the first element of the word is the rightmost one.

The semantics of the coefficients of M_n is, given a string where a_k is the last bit, a_1 the first, a and b any two bits:

l_{ab}	Goes from the output of $a \in \{0,1\}$ to the input of $b \in \{0,1\}$.
$l_{ab}{}^t$	Goes from the input of $b \in \{0,1\}$ to the output of $a \in \{0,1\}$.
l_{Sa_k}	Goes from the output of $\star(=S)$ to the input of $a_k \in \{0,1\}$.
$l_{Sa_k}{}^t$	Goes from the input of $a_k \in \{0,1\}$ to the output of $\star(=S)$.
l_{a_1E}	Goes from the output of $a_1 \in \{0,1\}$ to the input of $\star(=E)$.
$l_{a_1E}{}^t$	Goes from the input of $\star(=E)$ to the output of $a_1 \in \{0,1\}$.

Each of those coefficient is a $(k+1)\times(k+1)$ matrix: the dimension corresponds to the size of the string encoded plus an additional room to handle $\star$. The matrix M_n is a 6×6 matrix —as the number of occurrence of literals in the conclusion of the proof— organized as follows:



The values in a rectangle are 0 because an input (resp. output) will never be connected to another input (resp. output), or because $\star$ will never be connected to itself.
The reader should of course refer to Definition 3.3.1 for a precise definition of M_n the matrical representation of n.

Figure 3.6: Some explanations about the matricial representation of a binary word

Some remarkable properties of the matricial representation

For $k \in \mathbb{N}^*$ and $0 \leqslant i \leqslant k$, we define $\pi_{k,i}$ to be the $(k+1)\times(k+1)$ matrix such that

$$\pi_{k,i}[a,b] = \begin{cases} 1 & \text{if } a = b = i \\ 0 & \text{elsewhere} \end{cases} \qquad \text{i.e.} \qquad \pi_{k,i} \coloneqq \begin{array}{c} \\ \\ \\ i \\ \\ \end{array} \overset{\begin{array}{ccccc} & & & i & \\ \end{array}}{\begin{pmatrix} 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ 0 & \dots & 1 & \dots & 0 \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 \end{pmatrix}}$$

We will in the following omit the subscript k, for the dimension of the projection should always be clear from context. Those projections shares a lot with the encoding we defined, as the small list of equalities that follow shows. This subsection can bee seen as a small break in our journey, where we look at how the objects we defined behave.

$$\pi_1 = (\mathrm{l}_{0\mathrm{E}} + \mathrm{l}_{1\mathrm{E}})(\mathrm{l}_{0\mathrm{E}} + \mathrm{l}_{1\mathrm{E}})^{\mathrm{t}}$$
$$\pi_0 = (\mathrm{l}_{0\mathrm{E}} + \mathrm{l}_{1\mathrm{E}})^{\mathrm{t}}(\mathrm{l}_{0\mathrm{E}}\mathrm{l}_{1\mathrm{E}})$$

Depending on the last bit, $\mathrm{l}_{0\mathrm{E}}$ or $\mathrm{l}_{1\mathrm{E}}$ is

$$\begin{pmatrix} 0 & 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}$$

The other being the 0 matrix. And it is quite clear that

$$\begin{pmatrix} 0 & 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}\begin{pmatrix} 0 & 1 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}\begin{pmatrix} 0 & 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}$$

We have a similar result regarding the beginning of the word represented:

$$\pi_0 = (\mathrm{l}_{\mathrm{S}0} + \mathrm{l}_{\mathrm{S}1})(\mathrm{l}_{\mathrm{S}0} + \mathrm{l}_{\mathrm{S}1})^{\mathrm{t}}$$
$$\pi_k = (\mathrm{l}_{\mathrm{S}0} + \mathrm{l}_{\mathrm{S}1})^{\mathrm{t}}(\mathrm{l}_{\mathrm{S}0} + \mathrm{l}_{\mathrm{S}1})$$

According to the value of the first bit, one among $\mathrm{l}_{\mathrm{S}0}$ and $\mathrm{l}_{\mathrm{S}1}$ is the 0 matrix, the other being

$$\begin{pmatrix} 0 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{pmatrix}$$

And a simple computation proves our claim.

$$\begin{pmatrix} 0 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{pmatrix}\begin{pmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 0 \end{pmatrix} = \begin{pmatrix} 1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 0 \end{pmatrix}\begin{pmatrix} 0 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{pmatrix} = \begin{pmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 \end{pmatrix}$$

Informally, π_0 is the bridge between the start and the end of our word. To read the beginning of the word, we should go toward π_k, whereas π_1 gives information about the end of the sequence.

The representation of the body of the word has also a tight link with the projections. First, we should remark that the sum of the matrices has always the same structure, in fact it is the same that the one used to represent unary structures.

$$(\mathrm{l}_{\mathrm{S0}} + \mathrm{l}_{\mathrm{S1}}) + (\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11}) + (\mathrm{l}_{1\mathrm{E}} + \mathrm{l}_{0\mathrm{E}}) = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 & 1 \\ 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix}$$

This is not surprising, for it simply tells us that every single bit "is defined": for every one of them, there is one and only one matrix that contains information about it. The part regarding the body of the word

$$(\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11}) = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{pmatrix}$$

has an internal structure that links it to a sum of projections:

$$(\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11})(\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11})^{\mathrm{t}} = \pi_2 + \pi_3 + \ldots + \pi_k$$
$$(\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11})^{\mathrm{t}}(\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11}) = \pi_1 + \pi_2 + \ldots + \pi_{k-1}$$

Moreover, for $k > i > 0$,

$$\pi_{i+1} = (\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11})\pi_i(\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11})^{\mathrm{t}} \tag{3.2}$$

This equality tells us somehow how to go read the next bit. Of course, this equality does not apply for the first or the last bit, because those matrices contains no information about those bits. This is

expressed by $(\mathrm{l}_{00}+\mathrm{l}_{01}+\mathrm{l}_{10}+\mathrm{l}_{11})^k = 0$. This can be easily checked:

$$(\mathrm{l}_{00}+\mathrm{l}_{01}+\mathrm{l}_{10}+\mathrm{l}_{11})^2 = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 & 0 \end{pmatrix}$$

and so

$$(\mathrm{l}_{00}+\mathrm{l}_{01}+\mathrm{l}_{10}+\mathrm{l}_{11})^{k-1} = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \end{pmatrix}$$

This dynamics is interesting not only in itself, but also because we will use it as a *definition* in the next section: what represents an integer is not its size or the shape of those matrices, but their *dynamic* when iterated or multiplied with projections, something that can be perfectly expressed by operators.

3.4 How to Compute

We will first explain how it is possible to represent integers in the hyperfinite factor, relying on the classical embedding of the matrix algebra into the type II_1 hyperfinite factor $\mathfrak{R}$. As you may guess, this embedding will strongly rely on the previous construction. A first approach on how matrices could be avoided by using C^* algebra was already developed [50, Proposition 11, p. 58].

We will then spend some time discussing the representation of computation (or more precisely, acceptance) and then explain how programs can be represented in this setting. This part gives the answers to our concerns regarding *uniformity* and *objectivity*: we want a single kind of object to represent all the integers, and at the time we want the program to interact in the same way no matter the representative chosen.

Binaries as operators

We will embed the $(k+1)\times(k+1)$ matrices of Definition 3.3.1 in the hyperfinite factor $\mathfrak{R}$ in order to have a uniform representation of the integers: an integer —no matter its size— will be represented by an operator in $\mathrm{M}_6(\mathfrak{R})$ fulfilling some properties. To express them we define, given a sequence $\star a_k \dots a_1 \star$

representing an integer n and for $j,l \in \{0,1\}$, the sets:

$$I^n_{jl} = \{m \mid a_m = j, a_{m+1} = l, 1 \leqslant m \leqslant k\}$$
$$I^n_{Sj} = \{1\} \text{ if } a_1 = j, \emptyset \text{ elsewhere}$$
$$I^n_{jE} = \{k\} \text{ if } a_k = j, \emptyset \text{ elsewhere}$$

Roughly speaking, I^n_{Sj} (resp. I^n_{jE}) tells us about the first (resp. last) bit of the sequence and I^n_{jl} is the set of sequences of a j followed by a l. We let $\blacklozenge \in \{00, 01, 10, 11, S0, S1, 0E, 1E\}$ as a shortcut.

Those sets allow us to be more precise regarding the formulation of Equation 3.2:

$$\sum_{d \in I_\blacklozenge} \mathrm{l}_\blacklozenge \pi_d = \mathrm{l}_\blacklozenge \tag{3.3}$$
$$\sum_{d \in I_\blacklozenge} \pi_{d+1} \mathrm{l}_\blacklozenge = \mathrm{l}_\blacklozenge \tag{3.4}$$

This explain in hard-to-read terms a very simple fact that our example will help to grasp.

Example 3.4.1 (Eleven (3/3))**.** Our sequence $\star 001011 \star$ yields

$$I_{S0} = \emptyset \quad I_{S1} = \{1\} \quad I_{0E} = \{6\} \quad I_{1E} = \emptyset \quad I_{00} = \{5\} \quad I_{01} = \{3\} \quad I_{11} = \{1\} \quad I_{10} = \{2,4\}$$

Remember (Example 3.3.2) for instance that in this case,

$$\mathrm{l}_{10} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This comes from the fact that in the 2nd and the 4th positions (counting from right to left) of our sequence there is a 1 followed by a 0. And so Equation 3.3, when instantiated to $I_\blacklozenge = I_{10}$, gives

$$\begin{aligned} \sum_{d \in I_{10}} \mathrm{l}_{10} \pi_d &= \sum_{i=2,4} \mathrm{l}_{10} \pi_i \\ &= (\mathrm{l}_{10} \pi_2) + (\mathrm{l}_{10} \pi_4) \\ &= \mathrm{l}_{10} \end{aligned}$$

This is simple to check, as

$$\mathrm{l}_{10} \pi_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and as

$$\mathrm{l}_{10}\pi_4 = \begin{pmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0 \end{pmatrix}\begin{pmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0 \end{pmatrix} = \begin{pmatrix} 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0 \end{pmatrix}$$

The "converse" Equation 3.4 can easily be checked the same way.

Now for the representation by operators: we will keep the same notation, simply change the family (from roman to *italic*), i.e. the operator representing l_{10} will be denoted by l_{10}. The only change will be that instead of taking the mere transposed of a matrix $(\cdot)^{\mathrm{t}}$, we will use the the conjugate-transpose of operators, denoted by $(\cdot)^*$. The Proposition 3.4.1 will justify this identical notation.

Definition 3.4.1 (Binary representation of integers)**.** An operator $N_n \in \mathrm{M}_6(\mathfrak{R})$ is a *binary representation* of an integer n if there exists projections $\pi_0, \pi_1, \ldots, \pi_k$ in $\mathfrak{R}$ that satisfy $\sum_{i=0}^{k} \pi_i = 1$ and

$$N_n = \begin{pmatrix} 0 & l_{00} & 0 & l_{10} & l_{S0} & 0\\ l_{00}^* & 0 & l_{01}^* & 0 & 0 & l_{0E}^*\\ 0 & l_{01} & 0 & l_{11} & l_{S1} & 0\\ l_{10}^* & 0 & l_{11}^* & 0 & 0 & l_{1E}^*\\ l_{S0}^* & 0 & l_{S1}^* & 0 & 0 & 0\\ 0 & l_{0E} & 0 & l_{1E} & 0 & 0 \end{pmatrix}$$

The coefficients of N_n are partial isometries fulfilling the following equations (where $\pi_{k+1} = \pi_0$).

$$l_\blacklozenge = \sum_{i \in I_\blacklozenge^n} \pi_{i+1} l_\blacklozenge \pi_i \qquad (\blacklozenge \in \{00, 01, 10, 11, S0, S1, 0E, 1E\}) \tag{3.5}$$

$$\pi_0 = (l_{S0} + l_{S1})(l_{00} + l_{01} + l_{10} + l_{11})^{k-1}(l_{0E} + l_{1E}) \tag{3.6}$$

Equation 3.5 and Equation 3.6 are the core of the computational behaviour we expect, they sum up the equations of the previous subsection and *define* our representation of integers as operators.

For all $i \in \{1, \ldots, k\}$, let $u_i = ((l_{00} + l_{01} + l_{10} + l_{11})^i (l_{0E} + l_{1E}))$. We should remark that $u_i u_i^* = \pi_i$ and that $u_i^* u_i = \pi_0$, i.e. that π_0 is equivalent[21] to π_i.

Proposition 3.4.1 (Binary and matricial representations)**.** *Given $N_n \in \mathrm{M}_6(\mathfrak{R})$ a binary representation of the integer n, there exists an embedding $\theta : \mathrm{M}_{k+1}(\mathbb{C}) \to \mathfrak{R}$ such that*[22] $\mathrm{Id} \otimes \theta(\mathrm{M}_n) = N_n$*, where M_n is the matricial representation of n.*

The global picture of the embedding we are going to define is as follows:

[21] In the sense of the Murray-von Neumann equivalence (remember Definition A.2.3).

[22] We denote by Id the identity matrix of $\mathrm{M}_6(\mathbb{C})$. We will allow ourselves the same abuse of notation in the following statements and proofs in order to simplify the formulae.

$$\begin{array}{ccc} \mathrm{M}_6 & (& \mathrm{M}_{k+1}(\mathbb{C})) \\ \downarrow & & \downarrow \\ \mathrm{M}_6 & (& \mathfrak{R} \end{array} \qquad \mathrm{Id} \otimes \theta$$

It uses the fact that $\mathrm{M}_{6(k+1)}(\mathbb{C})$ is isomorphic to $\mathrm{M}_6(\mathbb{C}) \otimes \mathrm{M}_{k+1}(\mathbb{C})$. We explained the main steps to embed a matrix algebra into the hyperfinite factor in Definition A.2.7, and perform this construction anew below.

Proof. Let $N_n \in \mathrm{M}_6(\mathfrak{R})$ be a binary representation of $n \in \mathbb{N}$, and $\pi_0, \ldots, \pi_k$ the associated projections. We now define an embedding $\theta : \mathrm{M}_{k+1}(\mathbb{C}) \to \mathfrak{R}$:

$$\theta : \mathrm{M} \mapsto \sum_{i=0}^{k} \sum_{j=0}^{k} \mathrm{M}[i,j] u_{i,j}$$

with:

$$u_{i,j} = \begin{cases} (l_{00} + l_{01} + l_{10} + l_{11})^{j-1}(l_{0E} + l_{1E}) & \text{if } i = 0 \quad (3.7a) \\ (l_{00} + l_{01} + l_{10} + l_{11})^{j-1} & \text{if } i < j \text{ and } i \neq 0 \quad (3.7b) \\ ((l_{00} + l_{01} + l_{10} + l_{11})^{i-1}(l_{0E} + l_{1E}))^* & \text{if } j = 0 \quad (3.7c) \\ ((l_{00} + l_{01} + l_{10} + l_{11})^{i-1})^* & \text{if } i > j \text{ and } j \neq 0 \quad (3.7d) \\ \pi_k & \text{if } i = j = k \quad (3.7e) \end{cases}$$

We can easily check that the image by $\mathrm{Id} \otimes \theta$ of the matrix M_n representing n is equal to N_n.

For instance, let $\Diamond \in \{00, 01, 10, 11\}$,

$$\begin{aligned} \theta(\mathrm{l}_\Diamond) &= \sum_{i=0}^{k} \sum_{j=0}^{k} \mathrm{l}_\Diamond[i,j] u_{i,j} && \text{(By definition of } \theta) \\ &= \sum_{p \in I_\Diamond} \mathrm{l}_\Diamond[p, p+1] u_{p,p+1} && \text{(By definition of } \mathrm{l}_\Diamond) \\ &= \sum_{p \in I_\Diamond} \mathrm{l}_\Diamond[p, p+1](l_{00} + l_{01} + l_{10} + l_{11})^{p-1} && \text{(By applying case 3.7b)} \end{aligned}$$

When we embed the transposed of $(\mathrm{l}_{00} + \mathrm{l}_{01} + \mathrm{l}_{10} + \mathrm{l}_{11})$, the computation is the same except that we use case 3.7d.

Regarding $\theta(l_{1E} + l_{0E})$, remember that $(\mathrm{l}_{1\mathrm{E}} + \mathrm{l}_{0\mathrm{E}})[i,j] = 0$ except if $i = 0$ and $j = 1$, so by applying case 3.7a, $\theta(\mathrm{l}_{1\mathrm{E}} + \mathrm{l}_{0\mathrm{E}}) = (l_{00} + l_{01} + l_{10} + l_{11})^0(l_{0E} + l_{1E}) = (l_{0E} + l_{1E})$. The embedding of $(\mathrm{l}_{\mathrm{S}1} + \mathrm{l}_{\mathrm{S}0})$ is quite similar, by applying case 3.7c.

We have of course that $\theta(\mathrm{M}^{\mathrm{t}}) = (\theta(\mathrm{M}))^*$, and $u_{i,j}$ is defined so that $u_{i,j}^* = u_{j,i}$. Remark also that θ is not always defined, for instance $\theta(u_{n,n})$ where $n \neq 0, k$ is not defined. **Q.E.D.**

Rather than having $(k+1) \times (k+1)$ matrices in a 6×6 matrix, we have partial isometries in a 6×6 matrix: partial isometries adapts themselves to the size of the word we represent. So having the same matrix of operators is a gain in terms of uniformity, as all the integers are represented by matrices of the same size. But at the same time, as any embedding $\theta : \mathrm{M}_{k+1}(\mathbb{C}) \to \mathfrak{R}$ defines a representation of the integers, we have to check that they all are equivalent.

Proposition 3.4.2 (Equivalence of binary representations)**.** *Given N_n and N'_n two binary representations of $n \in \mathbb{N}^*$, there exists a unitary $u \in \mathfrak{R}$ such that* $(\mathrm{Id}\otimes u)N_n(\mathrm{Id}\otimes u)^* = N'_n$.

Proof. Let $\pi_0,\dots,\pi_k$ (resp. $\nu_0,\dots,\nu_k$) be the projections and $l_\blacklozenge$ (resp. $l'_\blacklozenge$) the partial isometries associated to N_n (resp. N'_n). It is straightforward that π_0 and ν_0 are equivalent according to Murray-von Neumann definition, so there exists a partial isometry v such that $vv^* = \nu_0$ and $v^*v = \pi_0$. For all $0 \leqslant i \leqslant k$ we define the partial isometries:

$$v_i = \underbrace{\left((l'_{00}+l'_{01}+l'_{10}+l'_{11})^{i-1}(l'_{0E}+l'_{1E})\right)}_{\nu_0} v \underbrace{\left((l_{00}+l_{01}+l_{10}+l_{11})^{i-1}(l_{0E}+l_{1E})\right)^*}_{\pi_0^*}$$

We can easily check that:

$$v_i^* v_i = \pi_i \qquad\qquad v_i v_i^* = \nu_i$$

It follows that the sum $u = \sum_{i=0}^{n} v_i$ is a unitary and $(\mathrm{Id}\otimes u)N_n(\mathrm{Id}\otimes u)^* = N'_n$. **Q.E.D.**

All the representations of the same integer are equivalent, from a mathematical point of view, but if we want to compute with those integers, we have to be sure that the algorithm will make no difference between several representations. This will be the subject of the third subsection of this section, but before that we take some time to study *how* integers and programs interact, because that interaction experienced several definitions.

Where programs and integers interact

This small subsection is here to justify some choice, to dwell on the advances we proposed and to link this representation to an other choice made regarding GoI. The equivalence of all the representations has some philosophical motivations: Girard wanted all the representations to be "egal", none of them should be more "normal" than any other.[23] This notion of a norm as an unjustified constraint explains the title "Normativity in Logic".

But of course, what makes two representations equal or not is their behaviour from a computational point of view. We want to define "observations", programs acting on them, and we want them to be "objective", i.e. insensitive to the representation chosen. In this context, "abnormality" should become relative to the evaluating context.

Observations and integers will be the same kind of objects, i.e. operators in the II_1 hyperfinite factor, and they will interact through plain multiplication. The question is: "How can we represent the end of the computation?" We will define an accept / reject pattern, the observations do not produce any complex output. This first criteria proposed was "When the determinant of the product is 0", the answer we developed with Seiller is "When the product gets to 0", i.e. if it is nilpotent. To be fair, this second criteria was already in use in "Normativity in Logic" [59], but only in background.

Technically speaking, there was a first attempt to define a frame where programs and integers could nicely interact, using a commutation condition, but it was way too constraining. In this setting, an algorithm could express only trivial computation [59, Proposition 12.1].

[23] In fact, there is a detail that goes again the success of this plan: there is still a norm on the representations. If we look closely to Proposition 3.4.2, we should see that all the representations of an integer n has to have the same number of bits. This contradicts lightly the introductory remark of Section 3.3, but does not really alter the whole construction.

The interaction used by Girard was based on Fuglede-Kadison determinant, a generalization of the usual determinant of matrices that can be defined in a type II_1 factor. This requires to define "normalised" determinant and trace, for them to be independent on the dimension of the matrix under treatment. In Proposition 3.4.2, we showed that two representations were unitary equivalent, as the trace of a unitary is 1, that let the determinant of the representation of the integer unchanged. This used to be crucial when cancellation of determinant was used as a marker for the end of the computation. It was also interesting for the trace of the projections are independent of their size, and those "normalised" traces and determinants are invariant under the embedding of Proposition 3.4.1.

Those notions are nicely developed in Seiller's thesis [121, chap. 4.2] and it was proven, thanks notably to a technical lemma [120, Lemma 61], that this condition on determinant was equivalent to the condition on nilpotency in some cases. In fact, it even relax some constrains: most notably, it allows one to consider a broader class of groups. This comes from the fact that the use of the determinant forces to consider only amenable groups, so that the result of the crossed product in Proposition 3.4.3 yields the type II_1 hyperfinite factor. Defining acceptance with a clause on determinant but using nilpotency forced Girard to consider languages obtained from finite positive linear combinations of unitaries induced by the group elements. This positivity is needed so that the condition involving the determinant implies nilpotency. Taking directly nilpotency as a marker for acceptance allows us to get rid of this condition, and we could use negative coefficients.

The notion of *normative pair* is a joint restriction on the pair of two subalgebras $(\mathfrak{N}, \mathfrak{O})$ where we pick the representations and the programs. It was defined by Girard [59] in order to describe the situations in which an operator in $\mathfrak{O}$ acts uniformly on the set of all representations of a given integer in $\mathfrak{N}$.

Normative Pairs

We will introduce here at the same time the constraint we put on programs so they interact "indifferently" of the representation of the input, and *the* constraint we will use later on. The technical tools needed for the crossed product and in general to grasp the interactions between von Neumann algebras and groups are defined in Section A.2.

Definition 3.4.2 (Normative Pairs)**.** Let $\mathfrak{N}$ and $\mathfrak{O}$ be two subalgebras of a von Neumann algebra $\mathfrak{M}$. The pair $(\mathfrak{N}, \mathfrak{O})$ is a *normative pair (in $\mathfrak{M}$)* if:

- $\mathfrak{N}$ is isomorphic to $\mathfrak{R}$;
- For all $\phi \in \mathrm{M}_6(\mathfrak{O})$ and $N_n, N'_n \in \mathrm{M}_6(\mathfrak{N})$ two binary representations of n,

$$\phi N_n \text{ is nilpotent} \Leftrightarrow \phi N'_n \text{ is nilpotent} \tag{3.8}$$

Proposition 3.4.3. *Let S be a set and for all $s \in S$, $\mathfrak{N}_s = \mathfrak{R}$. For all group $\mathcal{G}$ and all action α of $\mathcal{G}$ on S, the algebra $\mathfrak{M} = (\bigotimes_{s\in S} \mathfrak{N}_s) \rtimes_{\hat{\alpha}} \mathcal{G}$ contains a subalgebra generated by $\mathcal{G}$ that we will denote $\mathfrak{G}$. Then for all $s \in S$, the pair $(\mathfrak{N}_s, \mathfrak{G})$ is a normative pair (in $\mathfrak{M}$).*

Proof. From the hypotheses, $\mathfrak{N}_s$ is isomorphic to $\mathfrak{R}$. Regarding the second condition (Equation 3.8), we will only show one implication, the other being obtained by symmetry. By Proposition 3.4.2, for

all N_n, N'_n two representation of n in $\mathfrak{N}_s$, there exists a unitary u such that $(\mathrm{Id}\otimes u)N_n(\mathrm{Id}\otimes u)^* = N'_n$. We define $v = \bigotimes_{s\in S} u$ and π_v the unitary in $\mathfrak{M}$ induced by v. Then π_v commutes with the elements of $\mathfrak{G}$,[24] so if there exists $d \in \mathbb{N}$ such that $(\phi N_n)^d = 0$, then $(\phi N'_n)^d = (\phi u N_n u^*)^d = (u\phi N_n u^*)^d = u(\phi N_n)^d u^* = 0$. **Q.E.D.**

Definition 3.4.3 (Observations)**.** Let $(\mathfrak{N}, \mathfrak{G})$ be a normative pair, an *observation* is an operator in $\mathrm{M}_6(\mathfrak{G}) \otimes \mathrm{Q}$, where Q is a matrix algebra, i.e. $\mathrm{Q} = \mathrm{M}_s(\mathbb{C})$ for an integer s, called the *algebra of states*.

For the time being, we may just think of Q as a technical trick to have some more space where the action of the group is not needed. Its role will be seen later on. Just remark that from now on, we have to take the tensor product of N_n and Id_Q as an input of an observation.

Definition 3.4.4 (Set "recognized" by an operator)**.** Let $(\mathfrak{N}_0, \mathfrak{G})$ be a normative pair and $\phi \in \mathrm{M}_6(\mathfrak{G})\otimes\mathrm{Q}$ be an observation. We define the set of natural numbers:

$$[\phi] = \{n \in \mathbb{N} \mid \phi(N_n \otimes \mathrm{Id}_\mathrm{Q}) \text{ is nilpotent, } N_n \in \mathrm{M}_6(\mathfrak{N}_0) \text{ any binary representation of } n\}$$

This could be seen as the language recognized by the operator ϕ, but for the time being it is only a mathematical definition.

Definition 3.4.5 (Language decided by a set of observations)**.** Let $(\mathfrak{N}_0, \mathfrak{G})$ be a normative pair and $X \subseteq \cup_{i=1}^{\infty} \mathrm{M}_6(\mathfrak{G}) \otimes \mathrm{M}_i(\mathbb{C})$ be a set of observations. We define the *language decided by* X as the set[25]:

$$\{X\} = \{[\phi] \mid \phi \in X\}$$

Now we will give the normative pair used as well by Girard [59] and by Aubert and Seiller [10, 11].

Corollary 3.4.4 (of Proposition 3.4.3)**.** *Let $\mathscr{S}$ be the group of finite permutations over $\mathbb{N}$, and for all $n \in \mathbb{N}$, $\mathfrak{N}_n = \mathfrak{R}$. Then $(\mathfrak{N}_0, \mathfrak{G})$ is a normative pair in $(\bigotimes_{n\in\mathbb{N}} \mathfrak{N}_n) \rtimes_{\hat{\alpha}} \mathscr{S}$.*

In this particular case, the algebra $(\bigotimes_{n\in\mathbb{N}} \mathfrak{N}_n) \rtimes_{\hat{\alpha}} \mathscr{S}$ is the type II_1 hyperfinite factor. This is one of the reason why Girard considered it, as it is then possible to use Fuglede-Kadison determinant. From now on, we will consider this normative pair fixed, and we will study three sets of observations. We first need to define the 1-norm:

Definition 3.4.6 (1-norm)**.** We define the family of norms $||\cdot||_1^k$ on $\mathrm{M}_k(\mathrm{M}_6(\mathfrak{R}))$ by:

$$||(a_{i,j})_{1\leqslant i,j\leqslant k}||_1^k = \max_{1\leqslant j\leqslant k}\left(\sum_{i=1}^{k} ||a_{i,j}||\right)$$

where $||a_{i,j}||$ is the usual norm, i.e. the C*-algebra norm (Definition A.1.17), on $\mathrm{M}_6(\mathfrak{R})$. We will simply write $||\cdot||_1$ when the superscript is clear from the context.

[24] A precise proof of this fact is Seiller's PhD Seiller [121, Proposition 11.2.5].

[25] To be consistent with our notation, we should have written them in bold, as $\{\mathbf{X}\}$. But we are going to introduce sets of observations denoted by P_+, and it could mislead the reader by reminding him/her of $\mathbf{P}$, the polynomial-time hierarchy. Morevoer, we wanted to keep the notation adopted in our previous works.

Definition 3.4.7 ($P_{\geqslant 0}$, P_+ and $P_{+,1}$)**.** An observation $\phi \in \mathrm{M}_6(\mathfrak{G}) \otimes \mathrm{M}_s(\mathbb{C})$ is said to be *positive* (resp. *boolean*) when for all $0 \leqslant i, j \leqslant 6s$, $\phi[i,j]$ is a positive finite linear combination (resp. a finite sum) of unitaries induced by elements of $\mathscr{S}$, i.e. $\phi[i,j] = \sum_{l=0}^{6s} \alpha_l \lambda(g_l)$ with $\alpha_l \geqslant 0$ (resp. $\alpha_l = 1$).

We then define the following sets of observations:

$$P_{\geqslant 0} = \{\phi \mid \phi \text{ is a positive observation}\}$$
$$P_+ = \{\phi \mid \phi \text{ is a boolean observation}\}$$
$$P_{+,1} = \{\phi \mid \phi \in P_+ \text{ and } \|\phi\|_1 \leqslant 1\}$$

We have trivially that $P_{+,1} \subsetneq P_+ \subsetneq P_{\geqslant 0}$ and $\{P_{+,1}\} \subseteq \{P_+\} \subseteq \{P_{\geqslant 0}\}$.

3.5 Nilpotency Can Be Decided With Logarithmic Space

We defined a way of representing computation with mathematical objects, but we should now wonder how complex it is to simulate this computation on another device. As usual, the Turing Machine will be the model of computation we chose to prove that the observations are a "reasonable" model of computation.

For the hyperfinite factor is a too complex object to be written on an input tape, we will need to prove that it is possible to work only with the significant part of the operator. This will be the aim of the following subsection, which states and proves a lemma of utmost importance. It is a hard proof that is due to Seiller and is taken from [10], it uses some mathematical notions that were not defined in this work: we refer to any classical textbook referenced in Appendix A or to Seiller [121].

Going back to finite-dimensional

We are going to perform here the reverse operation of the embeding of Definition 3.4.1: we will take the representation of an integer by an infinite operator and enclose it in a finite matrix. And we will do the same for the operator acting on it. Even though the hyperfinite factor $\mathfrak{R}$ is necessary in order to have a uniform representation of integers, it is an algebra of operators acting on an infinite-dimensional Hilbert space. It is therefore not obvious that one can check the nilpotency of such an operator with a Turing Machine, and it is of course not possible in general. However, the following lemma has as a direct consequence that checking the nilpotency of a product ϕN_n where ϕ is an observation in $P_{\geqslant 0}$ is equivalent to checking the nilpotency of a product of matrices.

Lemma 3.5.1. *We consider the normative pair* $(\mathfrak{N}_0, \mathfrak{G})$ *defined in Corollary 3.4.4 and denote by* $\mathfrak{M}$ *the algebra* $(\bigotimes_{n\geqslant 0} \mathfrak{R}) \rtimes \mathscr{S}$*. Let* N_n *be a binary representation of an integer* n *in* $\mathrm{M}_6(\mathfrak{N}_0)$ *and* $\phi \in \mathrm{M}_6(\mathfrak{G}) \otimes \mathrm{Q}$ *be an observation in* $P_{\geqslant 0}$*. Then there exist an integer* k*, an injective morphism* $\Psi : \mathrm{M}_k(\mathbb{C}) \to \mathfrak{M}$ *and two matrices* $M \in \mathrm{M}_6(\mathrm{M}_k(\mathbb{C}))$ *and* $\bar{\phi} \in \mathrm{M}_6(\mathrm{M}_k(\mathbb{C})) \otimes \mathrm{Q}$ *such that* $\mathrm{Id} \otimes \Psi(M) = (N_n \otimes 1_{\mathrm{Q}})$ *and* $\mathrm{Id} \otimes \Psi \otimes \mathrm{Id}_{\mathrm{Q}}(\bar{\phi}) = \phi$*.*

Proof. We denote by n the integer represented by N_n and $R \in \mathrm{M}_{6(n+1)}(\mathbb{C})$ its matricial representation. Then there exists a morphism $\theta : \mathrm{M}_{n+1}(\mathbb{C}) \to \mathfrak{R}$ such that $\mathrm{Id} \otimes \theta(R) = N_n$ by Proposition 3.4.1. Composing θ with the inclusion $\mu : \mathrm{M}_{n+1}(\mathbb{C}) \to \bigotimes_{n=0}^{N} \mathrm{M}_{n+1}(\mathbb{C})$, $x \mapsto x \otimes 1 \otimes \cdots \otimes 1$, we get:

$$\mathrm{Id} \otimes (\bigotimes_{n=0}^{N} \theta(\mu(R)) = \overline{N_n} \otimes \underbrace{1 \otimes \cdots \otimes 1}_{N \text{ copies}}$$

where $\overline{N_n}$ is the representation of n in $\mathrm{M}_6(\mathbb{C})\otimes\mathfrak{R}$ (recall the representation N_n in the statement of the lemma is an element of $\mathrm{M}_6(\mathbb{C})\otimes\mathfrak{M}$).

Moreover, since ϕ is an observation, it is contained in the subalgebra induced by the subgroup $\mathscr{S}_N$ where N is a fixed integer,[26] i.e. the subalgebra of $\mathscr{S}$ generated by $\{\lambda(\sigma) \mid \sigma\in\mathscr{S}_N\}$. We thus consider the algebra $(\bigotimes_{n=0}^{N}\mathrm{M}_{n+1}(\mathbb{C}))\rtimes\mathscr{S}_N$. It is isomorphic to a matrix algebra $\mathrm{M}_k(\mathbb{C})$: the algebra $\bigotimes_{n=0}^{N}\mathrm{M}_{n+1}(\mathbb{C})$ can be represented as an algebra of operators acting on the Hilbert space $\mathbb{C}^{N(n+1)}$, and the crossed product $(\bigotimes_{n=0}^{N}\mathrm{M}_{n+1}(\mathbb{C}))\rtimes\mathscr{S}_N$ is then defined as a subaglebra $\mathfrak{I}$ of the algebra $\mathcal{B}(L^2(\mathscr{S}_N,\mathbb{C}^{(n+1)^N}))\cong\mathrm{M}_{(n+1)^N N!}(\mathbb{C})$. We want to show that $(N_n\otimes 1_{\mathrm{Q}})$ and ϕ are the images of matrices in $\mathfrak{I}$ by an injective morphism Ψ which we still need to define.

Let us denote by α the action of $\mathscr{S}_N$ on $\bigotimes_{n=0}^{N}\mathrm{M}_{n+1}(\mathbb{C})$. We know by Definition A.2.11 that $\mathfrak{I}=(\bigotimes_{n=0}^{N}\mathrm{M}_{n+1}(\mathbb{C}))\rtimes\mathscr{S}_N$ is generated by two families of unitaries:

- $\lambda_\alpha(\sigma)$ where $\sigma\in\mathscr{S}_N$;
- $\pi_\alpha(x)$ where x is an element of $\bigotimes_{n=0}^{N}\mathrm{M}_{n+1}(\mathbb{C})$.

We will denote by γ the action of $\mathscr{S}$ on $\bigotimes_{n=0}^{\infty}\mathfrak{R}$. Then $\mathfrak{M}=(\bigotimes_{n\geqslant 0}\mathfrak{R})\rtimes\mathscr{S}$ is generated by the following families of unitaries:

- $\lambda_\gamma(\sigma)$ for $\sigma\in\mathscr{S}$;
- $\pi_\gamma(x)$ for $x\in\bigotimes_{n\geqslant 0}\mathfrak{R}$.

As we already recalled, ϕ is an observation in $P_{\geqslant 0}$ and is thus contained in the subalgebra induced by the subgroup $\mathscr{S}_N$. Moreover, N_n is the image through θ of an element of $\mathrm{M}_{n+1}(\mathbb{C})$. Denoting β the action of $\mathscr{S}_N$ on $\bigotimes_{n=0}^{N}\mathfrak{R}$, the two operators we are interested in are elements of the subalgebra $\mathfrak{J}$ of $\mathfrak{M}$ generated by:

- $\lambda_\beta(\sigma)$ for $\sigma\in\mathscr{S}_N$;
- $\pi_\beta(\bigotimes_{n=0}^{N}\theta(x))$ for $x\in\bigotimes_{n=0}^{N}\mathrm{M}_{n+1}(\mathbb{C})$.

We recall that ϕ is a matrix whose coefficients are finite positive linear combinations of elements $\lambda_\gamma(\sigma)$ where $\sigma\in\mathscr{S}_N$, i.e. (denoting by k the dimension of the algebra of states):

$$\phi=\Big(\sum_{i\in I_{a,b}}\alpha^i_{a,b}\lambda_\gamma(\sigma^i_{a,b})\Big)_{1\leqslant a,b\leqslant 6k}$$

We can therefore associate to ϕ the matrix $\bar{\phi}$ defined as

$$\bar{\phi}=\Big(\sum_{i\in I_{a,b}}\alpha^i_{a,b}\lambda_\alpha(\sigma^i_{a,b})\Big)_{1\leqslant a,b\leqslant 6k}$$

We will now use the theorem stating the crossed product algebra does not depend on the chosen representation (Theorem A.2.1). The algebra $\bigotimes_{n=0}^{N}\mathfrak{R}$ is represented (faithfully) by the morphism $\pi_\beta\circ\bigotimes_{n=0}^{\infty}\theta$. We deduce from this that there exists an isomorphism from $\mathfrak{I}$ to the algebra generated by

[26] That comes from the fact that ϕ "uses" only a finite number of permutations.

the unitaries $\lambda_\beta(\sigma)$ ($\sigma \in \mathcal{S}_N$) and $\pi_\beta \circ \bigotimes_{n=0}^{\infty} \theta(x)$ ($x \in \bigotimes_{n=0}^{N} \mathrm{M}_{n+1}(\mathbb{C})$). This isomorphism induces an injective morphism ω from $\mathfrak{I}$ into $\mathfrak{J}$ such that:

$$\omega(\pi_\alpha(x)) = \pi_\beta(\bigotimes_{n=0}^{N} \theta(x))$$
$$\omega(\lambda_\alpha(\sigma)) = \lambda_\beta(\sigma)$$

We will denote by ι the inclusion map $\bigotimes_{n=0}^{N} \mathfrak{R} \subseteq \bigotimes_{n=0}^{\infty} \mathfrak{R}$ and υ the inclusion map $\mathcal{S}_N \subseteq \mathcal{S}$. We will once again use the same theorem as before, but its application is not as immediate as it was. Let us denote by $\mathcal{S}_N \backslash \mathcal{S}$ the set of the orbits of $\mathcal{S}$ for the action of $\mathcal{S}_N$ by multiplication on the left, and let us chose a representant $\bar{\tau}$ in each of these orbits. Recall the set of orbits is a partition of $\mathcal{S}$ and that $\mathcal{S}_N \times \mathcal{S}_N \backslash \mathcal{S}$ is in bijection with $\mathcal{S}$. As a consequence, the Hilbert space $L^2(\mathcal{S}_N, L^2(\mathcal{S}_N \backslash \mathcal{S}, \bigotimes_{n=0}^{\infty} \mathbb{H}))$ is unitarily equivalent to $L^2(\mathcal{S}, \bigotimes_{n=0}^{\infty} \mathbb{H})$. We will therefore represent $\bigotimes_{n=0}^{N} \mathfrak{R}$ on this Hilbert space and show this representation corresponds to π_γ. For each $x \in \bigotimes_{n=0}^{N} \mathfrak{R}$, we define $\rho(x)$ by:

$$\rho(x)\xi(\bar{\tau}) = \gamma(\bar{\tau}^{-1})(\iota(x))\xi(\bar{\tau})$$

This representation is obviously faithful. We can then define the crossed product of this representation with the group $\mathcal{S}_N$ on the Hilbert space $L^2(\mathcal{S}_N, L^2(\mathcal{S}_N \backslash \mathcal{S}, \bigotimes_{n=0}^{\infty} \mathbb{H}))$. The resulting algebra is generated by the operators (in the following, $\xi \in L^2(\mathcal{S}_N, L^2(\mathcal{S}_N \backslash \mathcal{S}, \bigotimes_{n=0}^{\infty} \mathbb{H}))$):

$$\begin{aligned}
\lambda(\nu)\xi(\bar{\tau})(\sigma) &= \xi(\bar{\tau})(\nu^{-1}\sigma) \\
\pi(x)\xi(\bar{\tau})(\sigma) &= \rho(\beta(\sigma^{-1})(x))\xi(\bar{\tau})(\sigma) \\
&= \gamma(\bar{\tau}^{-1})(\gamma(\sigma^{-1})(\iota(x)))\xi(\bar{\tau})(\sigma) \\
&= \gamma(\bar{\tau}^{-1}\sigma^{-1})(\iota(x)))\xi(\bar{\tau})(\sigma) \\
&= \gamma((\sigma\bar{\tau})^{-1})(\iota(x)))\xi(\bar{\tau})(\sigma)
\end{aligned}$$

Through the identification of $L^2(\mathcal{S}_N, L^2(\mathcal{S}_N \backslash \mathcal{S}, \bigotimes_{n=0}^{\infty} \mathbb{H}))$ and $L^2(\mathcal{S}, \bigotimes_{n=0}^{\infty} \mathbb{H}))$, we therefore get:

$$\begin{aligned}
\lambda(\nu)\xi(\sigma\bar{\tau}) &= \xi(\nu^{-1}\sigma\bar{\tau}) \\
&= \lambda_\gamma(\nu)\xi(\sigma\bar{\tau}) \\
\pi(x)\xi(\sigma\bar{\tau}) &= \gamma((\sigma\bar{\tau})^{-1})(\iota(x)))\xi(\sigma\bar{\tau}) \\
&= \pi_\gamma(\iota(x))\xi(\sigma\bar{\tau})
\end{aligned}$$

Applying Theorem A.2.1 we finally get the existence of an injective morphism ζ from $\mathfrak{J}$ into $\mathfrak{M}$ such that:

$$\pi_\beta(x) \mapsto \pi_\gamma(\iota(x))$$
$$\lambda_\beta(\sigma) \mapsto \lambda_\gamma(\sigma)$$

Figure 3.7 illustrates the situation. We now define $\Psi : \mathfrak{I} \to \mathfrak{M}$ by $\Psi = \zeta \circ \omega$. Noticing that $N_n =$

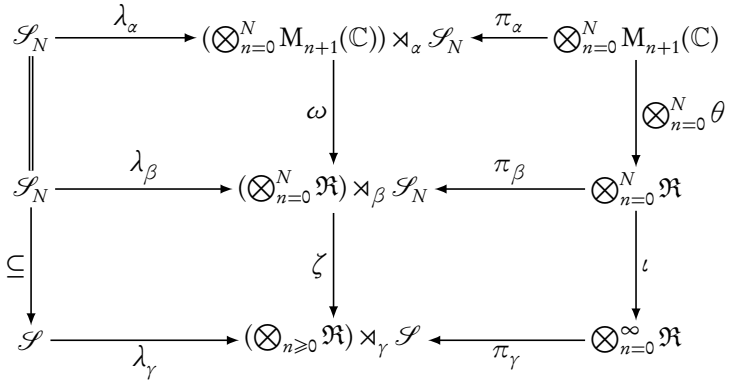


Figure 3.7: Representation of the main morphisms defined in the proof of Lemma 3.5.1

$\mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes (\pi_\gamma(\iota \circ \mu(\overline{N_n}))$, we get:

$$\begin{aligned}
\mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes \Psi(M) &= \mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes \Psi(\mathrm{Id} \otimes \pi_\alpha(\mathrm{Id} \otimes \mu)(R)) \\
&= \mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes \pi_\gamma(\iota \circ \bigotimes_{n=0}^{N} \theta(\mu(R))) \\
&= \mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes \pi_\gamma(\iota(\overline{N_n} \otimes 1 \otimes \cdots \otimes 1)) \\
&= \mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes \pi_\gamma(\iota \circ \mu(\overline{N_n})) \\
&= N_n
\end{aligned}$$

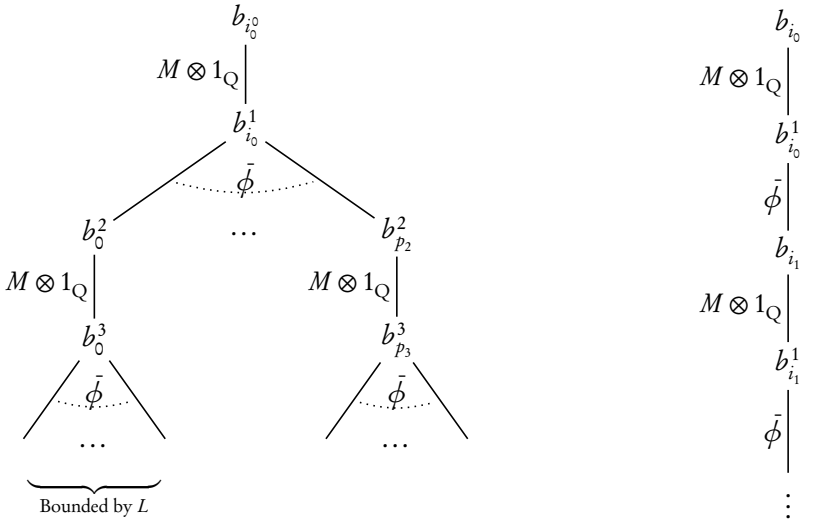
$$\begin{aligned}
\mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes \Psi \otimes \mathrm{Id}_{\mathrm{Q}}(\bar{\phi}) &= (\sum_{i \in I_{a,b}} \alpha_{a,b}^i \Psi(\lambda_\alpha(\sigma_{a,b}^i)))_{1 \leqslant a,b \leqslant 6k} \\
&= (\sum_{i \in I_{a,b}} \alpha_{a,b}^i \lambda_\gamma(\sigma_{a,b}^i))_{1 \leqslant a,b \leqslant 6k} \\
&= \phi
\end{aligned}$$

The (injective) morphism Ψ thus satisfies all the required properties. **Q.E.D.**

Stated less technically, the previous lemma tells us that no matter the operator ϕ, it always uses only a finite number of elements of $\mathscr{S}$, i.e. of permutations. Therefore, we can identify a finite subgroup $\mathscr{S}_N$ of $\mathscr{S}$, and $m \in \mathbb{N}$ such that $\forall \sigma \in \mathscr{S}, \forall i \geqslant m, \sigma(i) = i$. It follows that the computation of the iterates of $\phi(N_n \otimes 1_{\mathrm{Q}})$ takes place in the algebra $\mathrm{M}_6(\mathbb{C}) \otimes \mathrm{Q} \otimes (\bigotimes_{i=1}^m \mathrm{M}_{\mathrm{n}+1}(\mathbb{C}) \rtimes \mathscr{S}_N)$, which acts on a *finite* Hilbert space.

Recognizing operators with logarithmic space

This last part is devoted to the study of the complexity of deciding if the product of operators representing a program and an integer is nilpotent. It uses the previous lemma as an evidence that this product can be written as a finite object.



For $\bar{\phi}$ a matrix obtained from $\phi \in P_{\leqslant 0}$ and M a representation of an integer, the iteration of $\bar{\phi}(M \otimes 1_Q)$ may be represented as a tree, where b_i^j denotes the elements of the basis encountered.
On the right hand-side, if $\phi \in P_{+,1}$, the computation is simply a line graph.

Figure 3.8: A representation of the computation of an operator

Problem F: Nilpotency of $\phi(N_n \otimes 1_Q)$ for ϕ an operator in $P_{\geqslant 0}$

`Input:` $n \in \mathbb{N}$ and $\phi \in P_{\geqslant 0}$
`Output:` Accepts iff $\phi(N_n \otimes 1_Q)$ is nilpotent for $N_n \in \mathrm{M}_6(\mathfrak{N}_0)$ and $\phi \in \mathrm{M}_6(\mathfrak{G}) \otimes \mathrm{Q}$.

Theorem 3.5.2. *Problem F is in* **STA**$(\log, *, \forall)$.

Proof. The Turing Machine we are going to define cannot take as input ϕ, but we know there is a finite object that behave the same way. Indeed, we know thanks to Lemma 3.5.1 that $\phi(N_n \otimes 1_Q)$ is nilpotent iff $\bar{\phi}(M \otimes 1_Q)$ is nilpotent. Moreover, the degree of nilpotency of the these two products is the same.

Using the bound p given by Lemma 3.5.1, we know that our algebra is

$$\mathrm{M}_6(\mathbb{C}) \otimes ((\underbrace{\mathrm{M}_{n+1}(\mathbb{C}) \otimes \cdots \otimes \mathrm{M}_{n+1}(\mathbb{C})}_{p \text{ copies}}) \rtimes \mathscr{S}_N) \otimes \mathrm{Q}$$

We let K be the dimension of the Hilbert space this algebra is acting on, i.e. for q the dimension of Q, $K = 6(n+1)^p\, p!q$. The elements of its basis are of the form

$$(\pi, a_1, \ldots, a_p; \sigma; e)$$

where $\pi \in \{1, 6\}$ is an element of the basis $(0o, 0i, 1o, 1i, s, e)$ of $\mathrm{M}_6(\mathbb{C})$, $a_i \in \{1, \ldots, p\}$ are the elements of the basis chosen to represent the integer n, $\sigma \in \mathscr{S}_N$ and e is an element of a basis of Q. When we apply $M \otimes 1_Q$ representing the integer to an element of this basis, we obtain one and only one vector of

the basis $(\pi, a_1, \ldots, a_p; \sigma; e)$. The image of a basis element by the observation $\bar{\phi}$ is a finite linear positive combination of $L \in \mathbb{N}$ elements of the basis:

$$\bar{\phi}(\pi, a_1, \ldots, a_p; \sigma; e) = \sum_{i=0}^{L} \alpha_i (\rho, a_{\tau_i(1)}, \ldots, a_{\tau_i(p)}; \tau_i \sigma; e_i) \tag{3.9}$$

This combination is finite thanks to Lemma 3.5.1, and it is linear positive by definition of $P_{\geqslant 0}$ (Definition 3.4.7).

So we are going to define a universally non-deterministic Turing Machine that will follow the computation on each basis vector thus obtained. The computation can be represented by a tree, as shown in Figure 3.8.

We know that L and the nilpotency degree of $\bar{\phi}(M \otimes 1_{\mathrm{Q}})$ are both bounded by the dimensions of the underlying space, that is to say by K. Since every coefficient α_i is positive, the matrix is thus nilpotent if and only if every branch of this tree is of length at most K. What ends a branch being of course reaching 0.

This Turing Machine has a logarithmic amount of information to store: the current basis element (i.e. $b_{i_0^0}$), the last computed term of the sequence and its index. Every time a branch splits –that is, every time we apply $\bar{\phi}$–, a non-deterministic transition takes place in order to continue the computation on every branch. Two branches cannot cancel each other, because they have positive coefficients, and the overall product is nilpotent iff every element of its basis goes to 0. As our Turing Machine is "universally non-deterministic ", it can test for nilpotency every element in parallel.

If a branch ends, our Turing Machine halts accepting, if after K applications of $M \otimes 1_{\mathrm{Q}}$ and $\bar{\phi}$ the basis is not null, the Turing Machine starts anew the computation with a "fresh" element of the basis. Once all the elements of the basis has been "tested" and none of them led to nilpotency, the Turing Machine halts rejecting. So the Turing Machine accepts iff there exists an element of the basis such that all branches cancel, that is to say iff $\bar{\phi}(M \otimes 1_{\mathrm{Q}})$ is nilpotent, and by Lemma 3.5.1 this is equivalent to $\phi(N_n \otimes 1_{\mathrm{Q}})$ nilpotent. **Q.E.D.**

There is a variant of Problem F, when $\phi \in P_{+,1}$, and we define the following problem.

Problem G: Nilpotency of $\phi(N_n \otimes 1_{\mathrm{Q}})$ for ϕ an operator in $P_{+,1}$

`Input:` $n \in \mathbb{N}$ and $\phi \in P_{+,1}$
`Output:` Accepts iff $\phi(N_n \otimes 1_{\mathrm{Q}})$ is nilpotent for $N_n \in \mathrm{M}_6(\mathfrak{N}_0)$ and $\phi \in \mathrm{M}_6(\mathfrak{G}) \otimes \mathrm{Q}$.

Theorem 3.5.3. *Problem G is in* **STA**$(\log, *, \mathbf{d})$.

Proof. In the case when $\bar{\phi}$ is obtained from an observation $\phi \in P_{+,1}$. In this particular case, Equation 3.9 becomes:

$$\bar{\phi}(\pi, a_1, \ldots, a_p; \sigma; e) = (\rho, a_{\tau_{b_0}(1)}, \ldots, a_{\tau_{b_0}(p)}; \tau_{b_0} \sigma; e_{b_0}) \tag{3.10}$$

Indeed, writing $\bar{\phi}$ as a $\mathrm{Card}(Q) \times \mathrm{Card}(Q)$ matrix (here Q denotes a basis of Q, which is a matrix algebra from Definition 3.4.3 of observations), we use that $||\bar{\phi}||_1 \leqslant 1$ to deduce that for all $q \in \mathrm{Q}$, $\sum_{q' \in \mathrm{Q}} ||\bar{\phi}[q, q']|| \leqslant 1$.

From the equation:

$$\bar{\phi}(\pi, a_1, \ldots, a_p; \sigma; e) = \sum_{i=0}^{l} \alpha_i (\rho, a_{\tau_i(1)}, \ldots, a_{\tau_i(p)}; \tau_i \sigma; e_i)$$

we have $||\bar{\phi}[e,e_i]|| = 1$ $(i = 1,\dots,l)$, and thus $\sum_{e'\in Q}||\bar{\phi}[e,e']|| \geqslant l$. Therefore, $l \leqslant 1$, and we refer to Figure 3.8 for a graphical representation.

Notice that the nilpotency degree of $\bar{\phi}M$ is again bounded by K (the dimension of the Hilbert space the algebra is acting on). One can thus easily define a deterministic Turing Machine that takes the basis elements one after the other and compute the sequence b_{i_0}, b_{i_1}, ...: if the sequence stops before the Kth step, the machine starts with the next basis element as b_{i_0}, and if the sequence did not stop after K step it means the matrix was not nilpotent. **Q.E.D.**

Corollary 3.5.4.

$$\{P_{\geqslant 0}\} \subseteq \textbf{co-NL} \qquad\qquad \{P_{+,1}\} \subseteq \textbf{L}$$

Proof. From Lemma 3.5.1, given an observation and an integer, we know there is a finite product of matrix that is nilpotent iff the product of this observation and integer is nilpotent. Then, by Theorem 3.5.2 and Theorem 3.5.3, we know that a log-space Alternating Turing Machine can decide if this product is nilpotent, i.e. if the observation is going to accept or reject the input. If this observation is of 1-norm strictly greater than 1, the Alternating Turing Machine has to be universally non-deterministic, to test in parallel every element of the basis. If this observation is of 1-norm less or equal to 1, the Alternating Turing Machine only has to test the elements of the basis one after the other, therefore it can be deterministic. **Q.E.D.**

Results and perspectives

What have we done?

We showed how to represent data and programs as mathematical objects. We first used the Church encoding and the Curry-Howard isomorphism to represent data —in this case integers— as proofs. Then, by using the proof net representation, we showed how it was possible to represent those proofs as matrices. As this representation was not uniform, i.e. the size of the matrix depends on the size of the integer, we showed how the matrices could be embedded in the II_1 hyperfinite factor. The motivations to use this mathematical object was exposed in the first section.

There is infinitely many ways of representing integers as operators in a von Neumann algebra, but we managed to define a framework —normative pairs— where integers and programs interact nicely, i.e. where the programs are indifferent to the particular representation chosen. We used the most convenient normative pair, i.e. the one that internalise permutations over $\mathbb{N}$. We chose to take nilpotency as a criteria of acceptance rather than the cancellation of the determinant, and that allows us to consider broader classes of groups.

At last by using a lemma proven by Thomas Seiller, we showed that it was possible to come back to finite objects and to decide their nilpotency with logarithmic space resources. This algebra has for the time no precise computational meaning, but we introduced a norm that somehow represent deterministic computation, the "basic" computation being widely non-deterministic, and to be more precise universally non-deterministic.

What could be done?

The purely mathematical way of modelling computation we defined in this chapter is a brand new and exciting field of research. Our first task will be to understand its computational nature, and that will

be one of the purposes of the next chapter, by proving that a specific kind of "pointer machine" can be reasonably simulated by operators.

And yet, there are already numerous hints toward extensions or modification of this framework:

1. We could try to embed other kinds of data in the von Neumann algebras, to have several constructors. As long as a data can be represented by a proof, we can follow exactly the same construction and obtain a representation in the hyperfinite factor. For instance, any finite list of objects of type U can be typed with $\forall X\ X \to ((U \to (X \to X)) \to X)$ in system F [62, pp. 90 sqq.]. This could take place in a general movement to go back to logical considerations.

2. As we sketched in Section 3.4, we could try to take the crossed-product of a von Neumann algebra with other groups. A good candidate could be "l'algèbre des boîtes", introduced by Girard [56, pp. 506 sqq.]. That would be really nice if we could prove some property like "if $\mathscr{G}$ is a strict subgroup of $\mathscr{H}$, then the set of elements recognized by a crossed product with $\mathscr{G}$ is strictly included in the set of elements recognized by a crossed product with $\mathscr{H}$." That could lead to separation results, but that the construction maintains strict inclusion could be hard to prove.

3. We could also play on another level with our normative pair, by allowing negative coefficients. In this setting, there could be cancellation between different "branches" of a computation, thus giving to our decomposition of the basis the structure of a graph —instead of the tree we saw in Figure 3.8. Such sharing in the computation reminds of classes of complexity stronger than log-space computation.

4. Another perspective could be to try to get rid of von Neumann algebras.[27] We could define observations that act only on a fixed size of input and stay on a matricial level. We should be really cautious about duplication in such a setting, but as well as one could give a bound on time for a space-bounded Turing Machine, maybe we could guarantee that an operator would not need matrices bigger than a pre-computed bound. That would probably need a uniformity similar to the one defined for the Boolean circuits (recall Definition 2.1.4), i.e. to define how to build a family of observations.

5. A last perspective would be to consider some advanced works in von Neumann algebra, which is a very active field of research. For instance, Fulman [43] developed a construction similar to the crossed product of a von Neumann algebra by a group, the crossed product of an abelian von Neumann algebra by an equivalence relation. Surely a discussion with an expert on those topics could leads to numerous other ideas!

[27] Since the writing of this memoir, Girard developed a new approach on Geometry of Interaction, using unification techniques [60]. A large part of the work presented in Chapter 3 and Chapter 4 has been rephrased in terms of "unification algebra" in a joint work with Marc Bagnol [9].

CHAPTER 4

Nuances in Pointers Machineries

Contents

Our goal in this chapter will be to build a computational model that accounts for the computation of the operators (or observations) defined in the previous chapter. For reasons that should be clear after reading Section 4.4, the computation of operators may be seen as a wander in the structure of the binary word that cannot modify it, but that may "save" values in registers. It is quite natural to think of *pointers* in this setting, and to Girard [59] it went without saying that the stroll of some "fingers" was the log-space-computation.

However, we saw that speaking of "the" log-space computation could be misleading: recall (Definition 1.1.11) that depending on the bounds on time and alternation, an Alternating Turing Machine with

only a logarithmic amount of space can characterise from $\mathbf{NC}^1$ up to **P**! So another approach —more reasonable— is to start from this idea of fingers wandering in an input and to wonder to what model and what complexity class it could correspond. That makes us dive into another vivid field of research, closer to computer science, that deals with pointers, and we will see that depending on the kind of pointers we manipulate and the way we manipulate them, we can capture more, less, or exactly **L**.

We propose here a chapter mid-way between a brief summary of the "pointer machines" that were defined and an illustration of the flexibility of a model of computation —Finite Automaton—, but both will be necessary to achieve our goal, i.e. to define *Non-Deterministic Pointer Machines* as the model of computation that corresponds to operators. We will prove that claim thanks to a low-level simulation: the proof is mainly based on the definition of the encoding of instructions (read a value, move a pointer, change the state) as operators. This section will be quite long, for we have to recall the framework of Chapter 3 and introduce notations to ease the simulation. We moreover handle the case of deterministic computation.

The first two sections were greatly inspired by some discussions with Arnaud Durand, and Paulin Jacobé de Naurois gave me some precious hints during the redaction of this chapter. The first part of the following subsection witness a work led with Margherita Zorzi and Stefano Guerrini that unfortunately turned out to be a deadlock. Some elements of this chapter, especially the last two sections, is a joint work [10, 11] with Thomas Seiller and constitute our main contribution. Some elements, as Theorem 4.2.2, are presented in an innovative way.

It could be helpful to keep in mind some notions on descriptive complexity to fully grasp the first section of this chapter,[1] but apart from that, everything was defined in Chapter 1 and in Chapter 3. Some parts of the second section may be really redundant to a complexity expert, but we felt it could be advantageous to recall some fundamentals in this topic.

4.1 A First Round of Pointers Machines

We will not make the inventory of the pointer machines that were defined, for they are *really* numerous.[2] Ben-Amram [16] made it clear that the heading of "pointer machines" was really misleading, for it covers at the same time "pointer algorithms" and "abstract machine models", both atomistic and high-level. We will apply his advice and call the atomistic pointer machines model we are going to work with . . . pointers machines! We will nevertheless blur this distinction, for the subjects we are going to evoke are sometimes midway between models of computation, programming languages and descriptive complexity: it comes from the "fact that it is natural to implement a Pointer Algorithm using the capabilities of [SMM or LISP machines]." ([16, p. 94])

Instead, this section will go against the intuitive idea that "pointers are log-space", by showing that depending on the precise definition of the machine that manipulates "pointers" (whose definition also fluctuate), we can characterise less (with JAG and PURPLE) or more (with SMM and KUM) than **L**. The exact characterization of **L** by a pointer machine will be postponed to the next section.

This journey towards the understanding of the "pointer complexity" will help us to define a subtle variation of Finite Automata that will end up characterizing the kind of computation we mimic with the operators of Chapter 3.

[1] And the reader may with benefit take a look at two classical textbooks, Immerman [79] and Ebbinghaus and Flum [40].
[2] We could only mention Tarjan [129]'s reference machine as one of the first model refereed to as "pointer machine".

JAGs and PURPLE: on the limitations of pebbles and pure pointers

We begin with two models, strongly linked, that cannot decide a problem that was proven to belong to **L**.

Jumping Automaton for Graphs (JAG) were introduced by Cook and Rackoff [29] to try to demonstrate a lower-bound for *maze-threadability* problem, a variation on *STConn* (recall Problem B). This model takes a graph as input, and moves pebbles, or tokens, along its edges. It has a finite state control, the ability to know if two pebbles are at the same vertex and to make one pebble "jump" to another. This model can easily be considered as a strict restriction of Turing Machine, it illustrates once again the importance of graphs for the log-space computation. It manipulates pointers in the sense that those pebbles are only location, addresses, in the input graph, they cannot be used to store any other kind of information. This model was designed to solve accessibility problems, but numerous modifications were proposed to handle other problems.

And yet, Hofmann and Schöpp [73] thought that this approach could be led further, for "neither JAGs nor [Deterministic Transitive Closure (DTC)]-logic adequately formalise the intuitive concept of "using a constant number of graph variables only"". This work is also greatly inspired by Cai, Fürer, and Immerman [22], who proved that First-Order Logic (FO) with least fixed point operator and counting, but without ordering, failed to capture **P**. So they define PURPLE, in order to make formal the idea that "log-space is more than 'a constant number of pointers'".

PURPLE is a `while` programming language, without quantifiers loop, and it is "pure" in the sense that pointers are abstract values, without internal structure. PURPLE's input are *partially ordered* graphs, for a precise reason: if there was a total order on graphs, then one could encode information in it. Without going further into details, we can mention that on ordered structures, FO and DTC corresponds to **L**, but it is not true on partially ordered structures. They tried to prove lower bounds and this approach was thrilling, for they proved that

- PURPLE cannot solve "the number of nodes of a graph is a power of 2" nor *USTConn* [73] which are in **L**. This remains true even if we add to PURPLE iteration (the quantifier loop) (this was a new lower bound, the purpose of [72]).

- With counting, PURPLE can decide *USTconn*.

- By introducing some elements of non-determinism, Hofmann, Ramyaa, and Schöpp [71] tries to prove that $\mathbf{L} \subsetneq^{?} \mathbf{PTIME}$. By doing so, they managed to show that even with counting, a non-deterministic version of PURPLE could not solve a **L**-complete problem, tree isomorphism.

Those numerous variations on PURPLE help to provide new lower bounds, and it is inspired by techniques of descriptive complexity. In this perspective, one can easily add or remove a single "ingredient to our recipe", the logic under study, and try to establish new bounds. The interested reader may for instance refer to the work of Grädel and McColm [66], where it is proven that transitive closure logic is strictly more powerful that DTC.

We tried with Margherita Zorzi and Stefano Guerrini to link together PURPLE and Girard's approach. Unfortunately, we came to the conclusion that our functional setting could not match the imperative setting of PURPLE, for we could encode structured information in our finite control.

KUM and SMM: questioning the meaning of log-*space*

The Kolmogorov Uspenskiĭ Machine (KUM) was first presented in Russian [89], but the English translation[90] of this article is better known. This model of computation is a "pointer machine" in the sense that instead of operating on registers —displayed as a tape— it manipulates a graph, not only by reading it —as for PURPLE—, but by making it evolves.

The Storage Modification Machine (SMM), introduced by Schönhage [116], was defined independently, but can be conceived as a special kind of KUM where the graph manipulated is directed, whereas it is undirected for the KUM. We give below an informal definition of the SMM.

Definition 4.1.1 (SMM)**.** A SMM is composed from a finite control given as a program and a dynamic Δ-structure for Δ an alphabet. The machine reads an input string and writes on an output string, and uses as a storage a dynamic labelled graph (X, a, p), where

- X is a finite set of nodes,
- $a \in X$ is the center (the active node),
- $p = (p_\alpha \mid \alpha \in \Delta)$ is the family of pointer mappings $p_\alpha : X \to X$

To write $p_\alpha(x) = y$ means that the pointer with label α originating from x goes to y. We define $p^* : \Delta^* \to X$ recursively:

$$\begin{aligned} p^*(\epsilon) &= a, \text{where } \epsilon \text{ is the empty word} \\ p^*(W\alpha) &= p_\alpha(p^*(W)) \text{ for all } \alpha \in \Delta, W \in \Delta^* \end{aligned}$$

A SMM manipulates a graph by moving its "center" in it, by merging and creating nodes, by adding or removing pointers, i.e. labelled edges. It manipulates pointers in the sense that p expresses *addresses*, it encodes paths from the center to any node.

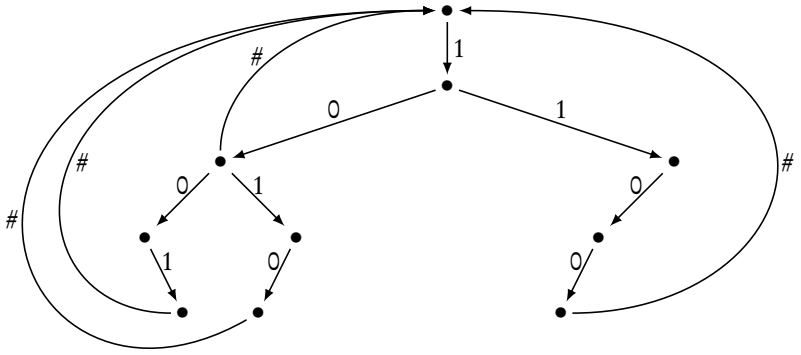
One simple example should show how to compute with this kind of machine.

Example 4.1.1 (Sort with SMM)**.** We want to build a simple program that sort integers. Suppose we want to sort 9, 12, 10 and 2, the input will be #1001#1100#1010#10#. The SMM will proceed as follows: every time the bit currently read is a 0 (resp. a 1), it tests for an edge labelled with 0 (resp. 1) leaving its actual center. If its exists, it follows it, elsewhere it creates a node, add an edge labelled with 0 (resp. 1) and follows it. When it reads a #, it goes back to its initial center. For the input we gave, when the SMM reaches the end of the input tape, the graph looks as in Figure 4.1

Then, the results is read simply by a depth first search, following in priority the edges labelled by #, 0, and then 1. This gives us for the current example : 10, 1001, 1010, 1100, i.e. the integers sorted.

The algorithm we gave is *linear* in the size of the input if we take the "uniform time measure", *a.k.a.* one transition count for one unit of time. It might look *too efficient*, for the best comparison-based sorting algorithms are $\mathcal{O}(n \log(n))$, and $\mathcal{O}(\log^2(n))$ in parallel.[3] All the same, it was proven [116, Theorem 6.1, p. 503] that a SMM could perform integer-multiplication in linear time, whereas the best known algorithm for other models needs $\mathcal{O}(n \log(n))$ steps.

[3] Of course, it is a different story if we consider *Counting Sort* or *Bucket Sort* algorithms.



Remember we are sorting #1001#1100#1010#10#.

Figure 4.1: How a KUM sorts integers

It is the same regarding space: it seems reasonable to consider that the number of nodes is the measure of space-complexity of a SMM. But on a Δ-structure with n nodes and if $\mathrm{Card}(\Delta) = k$, one can encode $\mathcal{O}(nk\log(n))$ bits of information, whereas a Turing Machine with n cells on alphabet Δ can encode $\mathcal{O}(n\log(k))$ bits. This results follows from a combinatorial analysis [116, p. 492] of the number of graphs of n nodes with edges labelled by elements of Δ.

In this setting, "one should be aware of the fact that for such a model log-**space** has become a different concept." (van Emde Boas [131, p. 106]) SMM and KUM does not belong to the class of "reasonable models of computation", for their time- and space-complexity —or at least the measure we defined on them— do not seem realistic.[4] They are a good way to express an algorithm, a sort of generalized or universal pointer machine that was rather used as a way of conceiving the problems and the algorithms than a model of computation actively used.

Yet, the ICC framework gave some new perspectives to that model with the recent work of Leivant and Marion [97]. They develop a program with a dynamic data-structure, that support the creation, deletion and update of vertices and edges. Thanks to ramification, they managed to enforce a polynomial time-complexity on that structure.

It seems really complicated to see the computational behaviour of observations as a rewriting of a graph. We were addressing the notion of pointer rather because our framework does not seem to have the ability to write, but only to parse. So no matter how interesting that model is, we will move to another, more suited. The reader who would like to learn more about this subject should look at the special issue of the Journal for Universal Computer Science [21], which is devoted to the celebration of the ten years of Gurevich's Abstract State Machines.

[4] This is a matter of taste: one could perfectly say that those models are the "reasonable" ones, and that the other models are too bureaucratic, inefficient, to handle smoothly complex algorithms based on pointers.

4.2 Finite Automata and L

We will recall the definition of a classical model of computation, Multi-Head Finite Automaton (FA), a read-only variant of the multi-head Turing Machines.[5] This model was introduced in the early seventies by Hartmanis [69] and provided many interesting results. Their nice characterization of **L** and **NL** (Theorem 4.2.4) and the similarities they share with Non-Deterministic Pointer Machines motivate their presence here in this small tutorial: even if they are called "(read-only) heads", they are exactly what we mean when we speak of "pointers". We will play a bit with this model of computation, take some times to understand how to compute with pointers and illustrate the flexibility of this model.

All the definitions and properties that follow are classical results, stated in a uniform way whereas the references sometimes mixes the notations. They are mostly borrowed to the recently published Holzer, Kutrib, and Malcher [75], which is a nice overview of the main results and open problems of the theory. Whereas the number of heads is an important criteria from a complexity point of view (in fact, it is the only relevant one), and even if separations results are known regarding that criteria (cf. Theorem 4.2.1), we won't count precisely the number of heads (the reason should be clear after looking at Theorem 4.2.4) in the modifications that follows.

Presentation and variations

Definition 4.2.1 (Non-Deterministic Multi-Head Finite Automaton (NFA))**.** For $k \in \mathbb{N}^*$, a *non-deterministic two-ways k-heads finite automaton* (NFA(k)) is a tuple $M = \{S, A, k, \triangleright, \triangleleft, s_0, F, \sigma\}$ where:

- S is the finite set of *states*;
- A is the *alphabet*;
- k is the number of heads;
- $\triangleright$ and $\triangleleft$ are the *left* and *right endmarkers*, $\triangleright, \triangleleft \notin A$;
- $\mathbf{s}_0 \in S$ is the *initial state*;
- $F \subseteq S$ is the set of *accepting states*;
- $\sigma \subseteq (S \times (A \cup \{\triangleright, \triangleleft\})^k) \times (S \times \{-1, 0, +1\}^k)$ is the *transition relation*, where -1 means to move the head one square to the left, 0 means to keep the head on the current square and 1 means to move it one square to the right.

  Moreover, for all $a_1, \dots a_k \in A^k$, $1 \leqslant i \leqslant k$ and $m_i \in \{-1, 0, +1\}$, whenever[6]

  $$(\mathbf{s}, (a_1, \dots, a_n)\sigma(\mathbf{s'}, (m_1, \dots, m_k)))$$

  then $a_i = \triangleright$ implies $m_i \in \{0, +1\}$, and $a_i = \triangleleft$ implies $m_i \in \{-1, 0\}$. That is to say that the heads cannot move beyond the endmarkers.

[5] Although this model was first defined as a variant of Finite Automaton, we thought it would be more convenient to introduce it as a variation of Turing Machine. We will nonetheless use some of their properties as a Finite Automaton, and will refer to the literature of this field. For an excellent overview of this theory, we refer for any classical textbook, like the one by Hopcroft, Motwani, and Ullman [76].

[6] Of course, one may see σ as a partial function from $S \times (A \cup \{\triangleright, \triangleleft\})^k$ to $\mathcal{P}_{\text{fin}}(S \times \{-1, 0, +1\}^k)$, and that justifies this notation that we will use from now on.

One defines configurations and transitions in a classical way. Let $M \in \mathrm{NFA}(k)$ and $w \in A^n$ an input of length n. We say that M *accepts* w if M starts in state $\mathbf{s}_0$, with $\triangleright\ w\ \triangleleft$ written on its input tape and all of its heads on $\triangleright$, and if after a finite number of transitions at least one branch of M reaches a state belonging to F and cannot perform any transition. We say that M always halts if for all input all branches of M reach after a finite number of transitions a configuration such that no transition may be applied. Remark that if no transition may be applied and the current state is not in F, this branch rejects the input. If all branches reject the input, that is, they all halt in a state not in F, M rejects the input.

If σ is functional, M is said to be *deterministic* and we write $\mathrm{DFA}(k)$ the set of deterministic two-way k heads automata (DFA). We will write Finite Automata (FA) in all generality, if we don't want to be specific about the (non-)determinacy of our Finite Automata. There are numerous other variation of the FA that we did not mention. Among them, the 1-way (Non-Deterministic) Finite Automata, which has exactly the same definition as the NFA, except that the heads can only go from left to right, i.e. -1 is not in the set of movements. This model is however interesting, for it provided some separation results and helped to be even more precise in the study of the computational power of the FA (we refer to the following subsection for that topic).

We will denote as usual by $\mathcal{L}(M)$ the language accepted by M. We will denote by **DFA** and **NFA** the sets of languages decided by respectively the sets $\cup_{k\in\mathbb{N}^*}\mathrm{DFA}(k)$ and $\cup_{k\in\mathbb{N}^*}\mathrm{NFA}(k)$. The weight of the font should help the reader to get the context: a machine belongs to $\mathrm{NFA}(k)$ and a problem to **NFA**.

Now we will assert several claims and give ideas of proofs. They are quite classical and some of their equivalent for Turing Machines were asserted without any proof, but we felt it should be made precise in this case.

Remember for instance that due to its definition, it is not trivial at all that for all $M \in \mathrm{DFA}(k)$ there exists $M' \in \mathrm{DFA}(k)$ such that M accepts iff M' rejects! It was an open problem for Hartmanis [69], denoting by C_D^k "the set of all languages accepted by deterministic k-head automata": "It would be also interesting to determine whether for every $A \in C_D^k$ also $\bar{A} \in C_D^k$. Again we do not know the answer to this problem. If C_D^k should not be closed under complementation then it would be interesting to find out how many additional heads are needed to obtain the complements of all sets in C_D^k. We know that $2k$ heads suffice but we suspect that a considerably smaller number of heads is really required to accept the complements." We won't bother answering this question here: all we need to know is that **DFA** $=$ **co-DFA**.

Before we develop some basic computation with our FA, we define some syntactical sugar:

- For every alphabet A, we define A' to be $A \cup \{\triangleright, \triangleleft\}$.
- We write $*$ for any symbol in A', [not a] for any symbol in $A' \setminus \{a\}$.
- We will sometimes do as if the heads could make several movements in one transition, i.e. take $\sigma \subseteq (S \times (A'^k) \times (S \times \{q\}^k)$ with $q \in \mathbb{Z}$. It is only a shorthand, for it is really easy to modify a FA that uses this feature to a one that does not.
- For all head H, we write $\sharp H$ the distance (i.e. the number of cells) between $\triangleright$ and H.

This last point will be used to show how some information can be encoded in *the position* of the heads: what will matter won't be any more the value pointed, but its position in the input string.

Every of our claims will apply as well to the deterministic and the non-deterministic cases, and the resources needed to process the translation and the overhead will be studied only for the first case, the proof being always the same.

Claim 1: *We can always take the alphabet A to be $\{0,1\}$.*

Proof. Formally, we prove that for all $k \in \mathbb{N}^*$, all $M \in \mathrm{FA}(k)$ with alphabet $A \neq \{0,1\}$ there exists $M' \in \mathrm{FA}(k)$ with $B = \{0,1\}$ and $f : A^* \to B^*$ such that $f(\mathcal{L}(M)) = \mathcal{L}(M')$. We will prove moreover that M' simulates M in real-time.

Let p be such that $2^{p-1} < \mathrm{Card}(A) \leqslant 2^p$. If $p = 1$, then f is simply a renaming of A. If $p > 1$, we encode any symbol of A with p bits in B. This is simply done by associating to every symbol of A a binary integer expressed with p bits.

Then, for all $a_1, \ldots, a_k \in A^k$ and and $(m^1, \ldots, m^k) \in \{-1, 0, +1\}^k$, every transition of M is of the shape

$$(\mathbf{s}, (a^1, \ldots, a^k))\sigma(\mathbf{s'}, (m^1, \ldots, m^k)) \tag{4.1}$$

For all such transitions, we introduce p states $\mathbf{s}_{a_1^i \ldots a_1^k}, \ldots, \mathbf{s}_{a_1^i a_2^i \ldots a_p^i \ldots a_1^k a_2^k \ldots a_p^k}$ where a_w^j denotes the wth bit of the binary encoding of a^j, i.e. $a_i \equiv a_1^i \ldots a_p^i$.

We then translate the transition of Equation 4.1 by

$$\begin{gathered}
(\mathbf{s}, (a_1^i, \ldots, a_1^k))\sigma'(\mathbf{s}_{a_1^i \ldots a_1^k}, (+1, \ldots, +1)) \\
(\mathbf{s}_{a_1^i \ldots a_1^k}, (a_2^i, \ldots, a_2^k))\sigma'(\mathbf{s}_{a_1^i a_2^i \ldots a_1^k a_2^k}, (+1, \ldots, +1)) \\
\vdots \\
(\mathbf{s}_{a_1^i a_2^i \ldots a_p^i \ldots a_1^k a_2^k \ldots a_p^k}, (a_m^i, \ldots, a_m^k))\sigma'(\mathbf{s'}, (m'^1, \ldots, m'^k))
\end{gathered}$$

where $m'^j = \begin{cases} -(2 \times p) & \text{if } m^j = -1 \\ -p & \text{if } m^j = 0 \\ +1 & \text{if } m^j = +1 \end{cases}$

We can easily check that M accepts an entry $w \in A^n$ iff M' accepts $f(w) \in \{0,1\}^{n \times \log(p)}$. The number of states of M' grows quite violently compared to M, but M' does not need more heads and simulates M in real time, i.e. with a linear overhead. **Q.E.D.**

It is worth mentioning that two letters- and three letters-alphabets are not always equivalent, as proven by Kozen [91], for instance when we study their properties as context-free languages. We won't study such properties here, but we have to pay attention to the complexity of f and the resources needed to describe M' given n and the description of M. The encoding f is really classic, and can be performed by a constant-depth circuit. The description of M' is mainly obtained by inserting routines in the description of M. With a log-time bounded ATM, one can guess in parallel the description of M, label every state with the k^p different values that may be read, and modify the transition function accordingly. This translation can then be performed with $\mathbf{NC}^1$ resources, using the power of alternation of the ATM. The following claims admit the same bounds on the complexity of the translation.

Claim 2: *It is possible to move only one head at every transition.*

Proof. Formally, we prove that for all $k \in \mathbb{N}^*$, for all $M \in \mathrm{FA}(k)$, there exists $M' \in \mathrm{FA}(k)$ such that $\mathcal{L}(M) = \mathcal{L}(M')$ and such that for all transition in M',

$$(\mathbf{s}_1, (a_1, \dots, a_k))\sigma'(\mathbf{s}_2, (m_1, \dots, m_k))$$

for all $1 \leqslant i \leqslant k$, only one among m_i is different from 0.

Every translation in M

$$(\mathbf{s}_1, (a_1, \dots, a_k))\sigma(\mathbf{s}_2, (m_1, \dots, m_k))$$

can be translated, for $\mathbf{s}_1^i$ k "fresh" states, i.e. not in S, by

$$\begin{aligned}
&(\mathbf{s}_1, (a_1, \dots, a_k))\sigma'(\mathbf{s}_1^1, (m_1, 0, \dots, 0)) \\
&\quad (\mathbf{s}_1^j, (*, \dots, *))\sigma'(\mathbf{s}_1^{j+1}, (0, \dots, 0, m_i, 0, \dots, 0)) && \text{(For } 1 \leqslant j < k\text{)} \\
&\quad (\mathbf{s}_1^k, (*, \dots, *))\sigma'(\mathbf{s}_2, (0, \dots, 0, m_k))
\end{aligned}$$

Q.E.D.

Definition 4.2.2 (Sensing heads)**.** We say that a FA has *sensing heads* if it has states $\mathbf{s}_{i=j?}$ such that

$$(\mathbf{s}_{i=j?}, (a_1, \dots, a_k))\sigma \begin{cases} (\mathbf{s}_1, (m_1, \dots, m_k)) & \text{if } \sharp H_i = \sharp H_j \\ \\ (\mathbf{s}_2, (m'_1, \dots, m'_k)) & \text{elsewhere} \end{cases} \tag{4.2}$$

In other words, our FA can "sense" if two heads are at the same cell and act accordingly. Note that it has nothing to do with non-determinacy: only one among the two transition of Equation 4.2 is applied.

Claim 3: *One additional head is sufficient to mimic the sensing heads ability.*

Proof. We prove that for all $M \in \mathrm{FA}(k)$ with sensing heads, there exists $M' \in \mathrm{FA}(k+1)$ without sensing heads such that $\mathcal{L}(M) = \mathcal{L}(M')$.

The simulation needs an additional head H_{k+1} that is on the beginning of the tape. Every translation of the form of Equation 4.2 is translated thanks to two fresh states $\mathbf{s}_{i=j}$ and $\mathbf{s}_{i\neq j}$, with the following routine[7]:

$$\begin{aligned}
(\mathbf{s}_{i=j?}, (\dots, a_i, \dots, a_j, \dots, \triangleright))\sigma'(\mathbf{s}_{i=j?}, (\dots, -1, \dots, -1, \dots, +1)) \\
(\mathbf{s}_{i=j?}, (\dots, [\text{not } \triangleright], \dots, [\text{not } \triangleright], \dots, *))\sigma'(\mathbf{s}_{i=j?}, (\dots, -1, \dots, -1, \dots, +1)) \\
\left.\begin{array}{l} (\mathbf{s}_{i=j?}, (\dots, \triangleright, \dots, [\text{not } \triangleright], \dots, *)) \\ (\mathbf{s}_{i=j?}, (\dots, [\text{not } \triangleright], \dots, \triangleright, \dots, *)) \end{array}\right\}\sigma'(\mathbf{s}_{i\neq j}, (\dots, +1, \dots, +1, \dots, -1)) \\
(\mathbf{s}_{i=j?}, (\dots, \triangleright, \dots, \triangleright, \dots, *))\sigma'(\mathbf{s}_{i=j}, (\dots, +1, \dots, +1, \dots, -1)) \\
(\mathbf{s}_{i=j}, (\dots, *, \dots, *, \dots, [\text{not } \triangleright]))\sigma'(\mathbf{s}_{i=j}, (\dots, +1, \dots, +1, \dots, -1)) \\
(\mathbf{s}_{i\neq j}, (\dots, *, \dots, *, \dots, [\text{not } \triangleright]))\sigma'(\mathbf{s}_{i\neq j}, (\dots, +1, \dots, +1, \dots, -1)) \\
(\mathbf{s}_{i=j}, (\dots, *, \dots, *, \dots, \triangleright))\sigma'(\mathbf{s}_1, (m_1, \dots, m_k, 0)) \\
(\mathbf{s}_{i\neq j}, (\dots, *, \dots, h_j, \dots, \triangleright))\sigma'(\mathbf{s}_2, (m'_1, \dots, m'_k, 0))
\end{aligned}$$

[7] Where the … are to be read as $*$ on the left hand side, and as 0 on the right hand side of σ'.

The symbol $\triangleright$ has the role of a buttress, the configurations store if we had $\sharp H_i = \sharp H_j$ and we can check that at the end of the test H_i, H_j and H_{k+1} are back to their position. We also need to modify all the other transition to add "whatever H_{k+1} reads" and "H_{k+1} does not move".

Every time M performs a comparison, M' have to perform at most $2 \times n$ transitions. **Q.E.D.**

We considered in Section 1.1 that it was trivial to define a Turing Machine that always halts from a Turing Machine that may loop, provided a bound, and we will prove the same for FA. We felt it necessary to give a proof, for it is amusing (the heads acts like hands of a clock) and not evident that without a writing ability such mechanism could be developed.

Claim 4: *We can consider only FA that always stops.*

Proof. We will prove that for all $M \in \mathrm{FA}(k)$, there exists $M' \in \mathrm{FA}(k')$ such that M' always halt and $\mathcal{L}(M) = \mathcal{L}(M')$.

Given an input $w \in A^n$, we know that the number of configurations of M is bounded by $\mathrm{Card}(S) \times (\mathrm{Card}(A'))^k \times (n+2)^k$, lets $d \in \mathbb{N}$ be such that the number of configurations is inferior to n^d. We will construct M' so that it halts rejecting after performing more than this number of transitions.

We set $k' = k + d$, and we construct M' so that its k first heads act as the heads of M, and the d heads act as the hands of a clock, going back and forth between the two endmarkers. We set

$$S' = \{s \times \{\rightarrow, \leftarrow\}^d \mid s \in S\}$$

that is to say that we copy every state of S and encode in them the current d directions of the d heads.[8] At every transition the dth head moves according to the direction encoded in the state. For $k < i < k'$, when the ith head reaches an endmarker, it changes its direction, something that is encoded in the state. If this endmarker was $\triangleright$, the $i+1$th head moves of one square according to its direction. If the k'th head has made a roundtrip on n —that is, is back on $\triangleright$— M' halts rejecting.[9]

If M did not stop after $\mathrm{Card}(S) \times 2^k \times (n+2)^k$ transitions, it means that it was stuck in a loop and therefore will never accept. By construction, M' always halts after n^d transitions and accepts exactly as M does, so we proved that $\mathcal{L}(M) = \mathcal{L}(M')$ and that M' always halts. **Q.E.D.**

Claim 5: *Finite Automata can compute addition, subtraction, power, multiplication, division*[10] *and modulo over integers.*

Proof. We prove that given two heads H_p and H_q such that $\sharp H_p = p$ and $\sharp H_q = q$, $M \in \mathrm{FA}(k)$ with $k \geqslant 4$ can put a head at distance $q+p$, $q-p$, q^p, $q \times p$ or $\lfloor q/p \rfloor$.

Suppose H_1 and H_2 are at $\triangleright$, they will be used to compute the value we are looking for. We define the following routines:

q+p H_1 goes right until $\sharp H_1 = \sharp H_q$, and every time H_1 goes right, H_p moves one square right. When H_1 reaches H_q, H_p is at distance $q+p$ from $\triangleright$.

q−p H_p goes left until it reaches $\triangleright$, and every time H_p goes left, H_q moves one square left. When H_p reaches $\triangleright$, H_q is at distance $q-p$ from $\triangleright$. If H_q reaches $\triangleright$ before H_p, p was greater than q.

[8] So if we had $\mathrm{Card}(S) = s$, $\mathrm{Card}(S') = s \times 2^d$, and we should even double the number of states to handle the fact that the $d+1$th head has already read $\triangleright$ or not.

[9] That can be simply done by halting the computation in a state not belonging to F.

[10] Rounded by the floor operation $\lfloor \cdot \rfloor$, with $\lfloor x \rfloor = \max\{m \in \mathbb{Z} \mid m \leqslant x\}$.

$\mathbf{q^p}$ H_1 goes right until $\sharp H_1 = \sharp H_q$

H_1 goes back to $\triangleright$ and every time H_1 moves left, H_q moves one square right.

When H_1 is at $\triangleright$, H_p goes one square left.

We iterate this routine until H_p read $\triangleright$, when it does, $\sharp H_q = q^p$.

$\mathbf{q \times p}$ H_1 goes right until $\sharp H_1 = \sharp H_q$

H_2 goes right until $\sharp H_2 = \sharp H_q$, and every time H_2 moves, H_p goes one square right.

When H_2 reaches H_q, H_1 goes one square left.

H_2 goes left until it reaches $\triangleright$, every time H_2 moves, H_p goes one square right.

When H_2 reaches $\triangleright$, H_1 goes one square left.

When H_1 reaches $\triangleright$, H_p is at distance $q \times p$ from $\triangleright$.

$\lfloor \mathbf{q/p} \rfloor$ H_1 goes right until $\sharp H_1 = \sharp H_p$, and every time H_1 moves, H_q goes one square left.

When $\sharp H_1 = \sharp H_p$, H_2 moves one square right.

Then H_1 goes left until it reaches $\triangleright$, and every time H_1 moves, H_q goes one square left.

When H_1 reaches $\triangleright$, H_2 moves one square right.

We iterate this routine until H_q reaches $\triangleright$. When it does, H_2 is at distance $\lfloor q/p \rfloor$ from $\triangleright$.

Regarding $q \mod p$, simply remark that $q \mod p = q - (\lfloor q/p \rfloor \times q)$, and we saw we can perform any of those operations, provided we have three extra heads to perform some intermediate computation.

About k and time: in the computations as developed above, up to three additional heads are required and the FA is supposed to have sensing heads.[11] The routines are linear in q, and they sometimes "destroy" the value q or p, but we may simply add an additional head that will be used as a marker to store the value. **Q.E.D.**

Classical results

To motivate the use of FA, we begin by giving some separation results, and then we study the links between **L**, **NL** and FA.

Theorem 4.2.1 (Separation results)**.** *For all $k \geqslant 1$,*

$$\mathcal{L}(\text{1DFA}(k)) \subsetneq \mathcal{L}(\text{1DFA}(k+1))) \tag{4.3}$$
$$\mathcal{L}(\text{1NFA}(k)) \subsetneq \mathcal{L}(\text{1NFA}(k+1)) \tag{4.4}$$
$$\mathcal{L}(\text{1DFA}(k)) \subsetneq \mathcal{L}(\text{1NFA}(2)) \tag{4.5}$$
$$\mathcal{L}(\text{DFA}(k)) \subsetneq \mathcal{L}(\text{DFA}(k+1)) \tag{4.6}$$
$$\mathcal{L}(\text{NFA}(k)) \subsetneq \mathcal{L}(\text{NFA}(k+1)) \tag{4.7}$$
$$\mathbf{NL} = \mathbf{L} \textit{ iff } \mathcal{L}(\text{1NFA}(2)) \subseteq \cup_{k \geqslant 1} \mathcal{L}(\text{DFA}(k)) \tag{4.8}$$

[11] But we saw how to get rid of that feature with an additional head previously.

Yao and Rivest [137] proved eqs. (4.3) to (4.5), eqs. (4.6) and (4.7) are due to Monien [108], and the proof of eq. (4.8) may be found in Sudborough [125, p. 75].

It is often taken for granted that the computation of FA is equivalent to log-space computation on a Turing Machine. For instance, for Monien [108, p. 67], "[i]t is wellknown that SPACE(log n), the class of languages acceptable within space bound log n, is identical with the class of languages acceptable by two-way multihead automata". Holzer, Kutrib, and Malcher [75] gives as a reference for this theorem Hartmanis [69], who introduced those "simple computing devices". It is true that this paper proves that "log n-tape bounded T.M.'s and k-head automata" are equivalent, but it is only a part of a more complex proof. So we begin with a reformulation of the proof that a FA can simulate a log-space bounded Turing Machine.

Theorem 4.2.2. *For all Turing Machine M using a logarithmic amount of space, there exists $k' \in \mathbb{N}^*$ and $M' \in \mathrm{FA}(k')$ such that $\mathcal{L}(M) = \mathcal{L}(M')$.*

Proof. We can assume without loss of generality that M has k working tapes, that its alphabet is $\{0,1\}$, that during the course of the computation, M won't use more than $\log(n)$ cells on each tape, and that all of its cells contains at the beginning of the computation the symbol 0. We will prove that M' accepts the same language as M, with $(k \times 2) + 5$ heads, $H_b^1, H_a^1, \ldots, H_b^k, H_a^k, H_r$ and four additional heads to perform some intermediate computation without destroying the values, following Claim 5.[12]

Informally, H_r will act as the reading head of M, H_b^i will encode the bits at the left ("before") of the head on the ith tape, and H_a^i will encode the bits at the right ("after") of this head. The bit currently read is encoded, by convention, in H_b^i.

For instance, suppose the head on the ith tape is on the jth cell, and the first j bits on this tape corresponds to the binary writing of the integer e. Then H_b^i will be at distance e from $\triangleright$ (i.e. $\sharp H_b^i = e$). The remaining $j - \log(n)$ bits are encoded *backward* by an integer e', and H_a^i will be at distance e' from $\triangleright$ (i.e. $\sharp H_a^i = e'$). The trick is that the least significant bit is in both case near the head of M'.

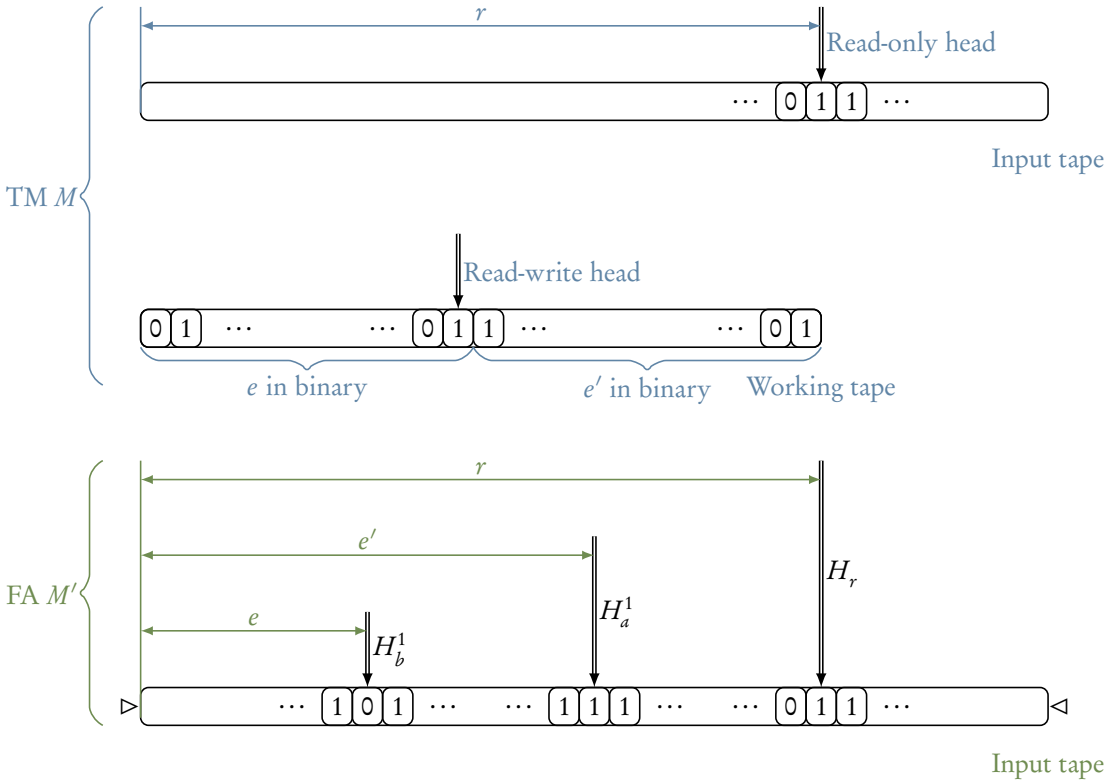
The following table explains how to mimic the basic operation of M with M':

Turing Machine	Finite Automata
Move reading head	Moves H_r accordingly
Read the value under the head on the ith tape	Compute $\sharp H_b^i \mod 2$
Write a 0 instead of a 1 (resp. a 1 instead of a 0) under the head on the ith tape	Move H_b^i one square right (resp. move H_b^i one square left)
Move the head on the ith tape right	Let $\sharp H_b^i$ be $(2 \times \sharp H_b^i) + (\sharp H_a^i \mod 2)$ Let $\sharp H_a^i$ be $\sharp H_a^i / 2$
Move the head on the ith tape left	Let $\sharp H_a^i$ be $(2 \times \sharp H_a^i) + (\sharp H_b^i \mod 2)$ Let $\sharp H_b^i$ be $\sharp H_b^i / 2$

All those intermediate computation may be written as routines, they all take a time linear in $|n|$, even without sensing heads. Those are not states, just the basic of the Turing Machine computation, how it actually runs. By just replacing the reading / writing pattern by movements of heads, we can encode the content of the tapes in the position of the heads.

Since M cannot write more than $\log(n)$ bits on each tape, it is obvious that the heads of M' does not need to be at a distance greater than $|n|$. Moreover, it is quite obvious that $\mathcal{L}(M) = \mathcal{L}(M')$, and that if M is (non-)deterministic, so is M'.

[12] A lot of optimization could be performed on this algorithm, of course, but this is not what is at stake here.



For simplicity, we considered a log-space bounded Turing Machine M with only one working tape. Remember that e' corresponds to the string at the right of the read-write head *encoded backward*, to lighten the simulation of the writing. Of course, the working-tape should be of length logarithmic in the length of the input tape: the scale is not correct!
The Finite Automata M' encodes all the informations within the distance between the beginning of the input tape and the heads. We refer to Theorem 4.2.2 for a precise statement.

Figure 4.2: How to simulate Turing Machines with Finite Automata

Once we know the number of tapes and if the Turing Machine is deterministic or not, it is easy to build a Boolean circuit family that takes as input $(Q, \Sigma, \delta, q_0, g)$ the description of M and that output $\{S, A, k, \triangleright, \triangleleft, s_0, F, \sigma\}$ the description of M'.[13] This can be done with constant-depth circuit. **Q.E.D.**

The Figure 4.2 should help the reader to get how this simulation works.

Corollary 4.2.3.

$$\mathbf{L} \subseteq \mathbf{DFA} \qquad\qquad \mathbf{NL} \subseteq \mathbf{NFA}$$

To obtain the converse inclusion is quite trivial: a Turing Machine simulates a FA(k) with $k \times \log(n)$ space by writing and updating the addresses of the k pointers, and that obviously take a logarithmic

[13] That would require to complete the proof to take care of the "special states" of M **accept** and **reject**, and to encode Q and δ in binary, but this is just a matter of plain encoding.

amount of space in n to write an address on a string of size n. If the FA was deterministic, so is the Turing Machine, otherwise they are both non-deterministic.

Theorem 4.2.4.

$$\mathbf{L} = \mathbf{DFA} \qquad\qquad \mathbf{NL} = \mathbf{NFA}$$

What is the exact meaning of this theorem? It is of interest to us because it gives an implicit characterization of both **L** and **NL**: any program that can be written with this device will use a logarithmic amount of space to be simulated on a Turing Machine, any language that can be recognized belong to **L** or **NL**. It is also the reason why 2-ways FA are more studied that their 1-way variant.

It might not be evident at first glance, but Finite Automata in general and this model in particular shares a lot with descriptional complexity, as illustrated by an (other) excellent survey recently published by Holzer and Kutrib [74].

4.3 Non-Deterministic Pointer Machines

All those previous modifications and theorems are classic, but now we will introduce some *ad-hoc* modifications to build a machine model called Pointer Machines (PM). This device is suited to be simulated by operators, it will help us to grasp the computational behaviour of observations. It was introduced and refined in two previous works [10, 11] led with Thomas Seiller, but the proofs and explanations below are far more advanced that what was proposed.

The main feature of this model is that it is universally non-deterministic, something that has been foreseen by Girard:

> "*[One of the goal of the GoI program is] to define an abstract computer that should not be too far from concrete ones. As we explained in [49], the geometrical analysis should essentially free us from* ad hoc *time, i.e. artificial causalities, overcontrol, and this means that we are clearly seeking a* parallel *computer.*"
>
> (Girard [48, p. 222])

The universally non-deterministic setting is clearly what we are looking for[14]: there is strictly no sharing, no communication, in this setting, every computation goes wildly in its own direction, even if the acceptance of the overall process (the nilpotency) is obtained by the cancellation of all the branches.

Pointers Machines (PMs) are designed to mimic the computational behaviour of operators: they do not have the ability to write, their input tape is cyclic, and rejection is meaningful whereas acceptance is the default behaviour. In a sense, the PMs are somehow the dual of FA, concerning acceptance, for a PM rejects if one of its branch reaches **reject**. We will present and motivates the other peculiarities of this model before defining them, and then we will prove that the similarities it shares with FA are "stronger" than their differences. An alternate proof of the characterization of **co-NL** by Non-Deterministic Pointer Machines may be found in Appendix B. This proof is somehow more direct for it does not need FA, but it cannot be extended to deterministic computation and does not provide the elements for a better understanding that are exposed below.

[14] And yet we did not used the "Alternating Multihead Finite Automata" proposed by King [88], for we would have used not enough of their features to justify their introduction.

Presentation

First, we list some peculiarities of any PM:

1. Its alphabet is $\{0,1,\star\}$ and $\star$ occurs only once.
2. It moves at most one pointer at every transition.
3. It does not have sensing heads.
4. It always stops.[15]
5. Its input is written on a circular input tape.
6. Acceptance and rejection are not states nor "labeled states", but in the co-domain of the transition function. They are, so to say, configurations.
7. Its transition relation is "total", i.e. for every configuration that is not **accept** or **reject**, there is a transition that can be applied.
8. The initialisation is quite different, because the value is read only when the pointer moves.
9. It suffices that one branch reaches **reject** to make the computation rejects.

We already showed that items 1 to 4 are reasonable modifications of FA, i.e. that provided we are not concerned by the number of heads, we can add or remove those features. We prove in the following (Claim 6, Claim 7) that item 5 and item 7 are not really important,[16] i.e. that we can for free add or remove this feature to FA.

The introduction of another name is justified[17] by item 7 and item 8: they would have needed to strongly modify the definition of FA to be implemented. If we took into account that **co-NL** = **NL**, item 9 could be implemented in the FA. We do not admit this theorem here, for it would mask one of the specificity of our machine, that justifies to see it as a complementary of FA and ease the gateways with operators.

We immediately prove that item 5 is not a big change:

Claim 6: *The input tape of a FA can be circular.*

Proof. Let $M \in \mathrm{FA}(k)$ whose input w is written as $\star w$ on a circular input tape i.e. that M can with two transitions goes from the last (resp. first) to the first (resp. last) bit of w. Suppose moreover that M moves only one head at a time.

To simulate M, we define $M' \in \mathrm{FA}(k)$, with $\triangleright\gg \star w \star \ll\triangleleft$ written on its (flat) input tape. The transition function σ' of M' is the same as the transition function σ of M, except that for all state $\mathbf{s} \in S$, for all $1 \leqslant i \leqslant k$, we add:

$$(\mathbf{s},(*,\dots,*,\gg,*,\dots,*))\sigma'(\mathbf{s}_i^{\mathrm{end}},(0,\dots,0,+1,0,\dots,0)$$
$$(\mathbf{s}_i^{\mathrm{end}},(*,\dots,*,[\text{not } \triangleleft],*,\dots,*))\sigma'(\mathbf{s}_i^{\mathrm{end}},(0,\dots,0,+1,0,\dots,0))$$
$$(\mathbf{s}_i^{\mathrm{end}},(*,\dots,*,\triangleleft,*,\dots,*))\sigma'(\mathbf{s},(0,\dots,0,-1,0,\dots,0)$$

[15] A property we called *acyclicity* when applied to NDPMs in our works with Thomas Seiller.
[16] The item 7 may be seen as the reverse operation of *minimization* [76, p. 159, section 4.4.3].
[17] Beside, we wanted to keep the original naming we developed with Tomas Seiller.

$$(\mathbf{s},(*,\dots,*,\ll,*,\dots,*))\sigma'(\mathbf{s}_i^{\text{beg}},0,\dots,0,-1,0,\dots,0)$$
$$(\mathbf{s}_i^{\text{beg}},(*,\dots,*,[\text{not } \rhd],*,\dots,*))\sigma'(\mathbf{s}_i^{\text{beg}},0,\dots,0,-1,0,\dots,0)$$
$$(\mathbf{s}_i^{\text{beg}},(*,\dots,*,\rhd,*,\dots,*))\sigma'(\mathbf{s},0,\dots,0,1,0,\dots,0)$$

where the instructions depends on, and give instructions to the ith head of M'. Those two routines permits to "freeze" the computation when a head is told to "use the ciclicity of the input tape". We move it to the other side of the tape and then resume the computation. **Q.E.D.**

Claim 7: *We can take F the set of accepting states to be a singleton, and make that every branch of the computation halt in that state or in another predefined state.*

Proof. Let M be a FA that always stop, we defined M' with two additional states **accept** and **reject**. For every state $\mathbf{s}\in S$ and every configuration $(\mathbf{s},(a_1,\dots,a_k))$ such that $\sigma(\mathbf{s},(a_1,\dots,a_k))=\emptyset$, we have in σ'

$$(\mathbf{s},(a_1,\dots,a_k))\sigma'\begin{cases}(\textbf{accept},(0,\dots,0)) & \text{if } \mathbf{s}\in F\\ \\ (\textbf{reject},(0,\dots,0)) & \text{elsewhere}\end{cases} \tag{4.9}$$

Q.E.D.

This is an important claim, for it shows how to get closer to a model of computation similar to the Turing Machine, where the computation ends when we reach **accept** or **reject**. By the way, we can notice that the FA M' we defined respects item 7, i.e. its transition relation is "total".

A Pointer Machine is given by a set of pointers that can move back and forth on the input tape and read (but not write) the values it contains, together with a set of states. For $1\leqslant i\leqslant p$, given a pointer p_i, only one of three different *instructions* can be performed at each step: p_i+, i.e. "move one step forward", p_i-, i.e. "move one step backward" and ϵ_i, i.e. "do not move". In the following, we let $I_{\{1,\dots,p\}}$ be the *set of instructions* $\{p_i+,p_i-,\epsilon_i \mid i\in\{1,\dots,p\}\}$. We will denote by $\sharp p_i$ the distance (clockwise) between $\star$ and the current position of p_i.

Definition 4.3.1. A non-deterministic pointer machine (NPM) with $p\in\mathbb{N}^*$ pointers is a couple $M=\{Q,\to\}$ where Q is the set of *states*; we always take the alphabet Σ to be $\{0,1,\star\}$ and the set of instructions to be $I_{\{1,\dots,p\}}$; and $\to\subseteq(\Sigma^p\times Q)\times((I^p_{\{1,\dots,p\}}\times Q)\cup\{\textbf{accept, reject}\})$ is the *transition relation* and it is total. We write $\mathrm{NPM}(p)$ the set of non-deterministic pointer machines with p pointers.

We define a *pseudo-configuration* c of $M\in\mathrm{NPM}(p)$ as a "partial snapshot": $c\in\Sigma^p\times Q$ contains the last values read by the p pointers and the current state, *but does not contain the addresses of the p pointers*. The set of pseudo-configurations of a machine M is written C_M and we should remark that it corresponds to the domain of the transition relation. If $\to$ is functional, M is a *deterministic* pointer machine, and we write $\mathrm{DPM}(p)$ the set of deterministic pointer machines with p pointers. We let $\mathrm{PM}(p)=\mathrm{DPM}(p)\cup\mathrm{NPM}(p)$.

Let $M\in\mathrm{PM}(p)$, $s\in C_M$ and $n\in\mathbb{N}$ an input. We define $M_s(n)$ as M with n encoded as a string on its circular input tape (as $\star a_1\dots a_k$ for $a_1\dots a_k$ the binary encoding of n and $a_{k+1}=a_0=\star$) starting in the *initial pseudo-configuration* s with $\sharp p_i=0$ for all $1\leqslant i\leqslant p$ (that is, the pointers starts on $\star$). Each of them is associated to a *memory slot* (or *register*) that store the values it reads, but (recall

item 8) as the pointers did not moved yet, no value has been read, and the slots are empty. The initial pseudo-configuration s gives the p values for the p registers and the initial state $\mathbf{s}$.

Remark that the *initial pseudo-configuration* s does not initializes those p registers necessarily in a faithful way (it may not reflect the values contained at $\sharp p_i$). That is to say that the first transition will rely on the values enclosed in the pseudo-configuration, and not on the values the pointer actually points at (that are always, by definition, $\star$). This complex definition will be justified in the following section.

The entry n is *accepted* (resp. *rejected*) by M with *initial pseudo-configuration* $s \in C_M$ if after a finite number of transitions every branch of $M_s(n)$ reaches **accept** (resp. at least a branch of M reaches **reject**). We say that $M_s(n)$ halts if it accepts or rejects n and that M decides a set S if there exists a pseudo-configuration $s \in C_M$ such that $M_s(n)$ accepts if and only if $n \in S$.

We write as usual $\mathcal{L}(M)$ the language of a PM M, and **DPM** (resp. **NPM**) the set of languages decided by $\cup_{p\in\mathbb{N}^*}\mathrm{DPM}(p)$ (resp. $\cup_{p\in\mathbb{N}^*}\mathrm{NPM}(p)$).

A modification of Finite Automata

One could quite simply prove that PM are just reasonable re-arranging of the presentation of FA "of a special kind". The minor modifications (among which the initial pseudo-configuration, the fact that **accept** and **reject** are not in the set of the states) are just red tape and does not enhance nor lower the power of computation of FA. What is really at stake, here, is to take into account that **co-NL** = **NL** or not. If we do, then it is obvious that PM can recognise any language in **NL**, if we don't, we should rather see them as some kind of complementary of FA. We prefer not to take as granted this result for the computation of operator algebra we are going to mimic should really be understood as universally non-deterministic.

Proposition 4.3.1 (Simulation of a FA by a PM)**.** *For all $k \in \mathbb{N}^*$, for all $M \in \mathrm{NFA}(k)$, there exists $M' \in PM(k+2)$ such that co-$\mathcal{L}(M) = \mathcal{L}(M')$.*

Proof. We can assume that M always halts, does not have sensing heads and has for alphabet $\{0,1\}$. The construction of M' is really simple: given $w \in \{0,1\}^*$ an input, M' takes as input the description of M as a list $S \rhd w \lhd \sigma \mathbf{s}_0 F$. Then M' is initialized by placing k pointers on $\rhd$, a pointer on $\mathbf{s}_0$ and the last one on the first symbol of σ. This last pointer scans σ to find a transition that can be applied. To simulate a transition, M' moves the k first pointers according to σ and the $k+1$th pointer is moved to a new state according to σ.

When a transition that may be applied is found, a non-deterministic transition takes place: on one way M' simulates this first transition, on the other way M' keeps scanning σ to look for another transition to apply. When no transition may be applied —and it always happens, as M always halts— , M' checks if the current state belongs to S: if it does, M' rejects, elsewhere M' accepts.

It can be easily verified that M' accepts iff M rejects, that $k+2$ pointers are enough to perform this simulation, and that if M is deterministic, so is M', because there is no need to look for another transition to apply. Note moreover that M' always halts, as M does. **Q.E.D.**

Corollary 4.3.2.

L $\subseteq$ **DPM** **co-NL** $\subseteq$ **NPM**

We do not prove the reverse inclusion by proving that FA can simulate PM, even if that would not be difficult. Instead, we will prove that the model we defined in the previous chapter, operators, can simulate PM: together with the results of the previous chapter, that gives us the reverse inclusion.

4.4 Simulation of Pointer Machines by Operator Algebra

This section recalls the results previously obtained [10] and the enhancements that were proposed later [11]. We tried to use the space at our disposal to ease the comprehension of this simulation: we will recall the framework we are working with, give a general idea of the simulation, introduce some notations and then encode the basic operations of a PM. Our efforts to define properly the basic mechanisms will give us the main result of this section almost for free, but a last subsection will be devoted to the careful analysis of the deterministic case.

Our aim in this section is to prove (Lemma 4.4.1) that for any PM M and initial pseudo-configuration $s \in C_M$, there exists an observation $M_s^\bullet \in \mathrm{M}_6(\mathfrak{G}) \otimes Q_M$ such that for all $N_n \in \mathrm{M}_6(\mathfrak{N}_0)$ a binary representation of n, $M_s(n)$ accepts if and only if $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ is nilpotent. We will prove moreover that $M_s^\bullet \in P_+$, and that if M is deterministic, then $M_s^\bullet \in P_{+,1}$, i.e. that the encoding of DPMs satisfies a particular property: their 1-norm is less or equal to 1.

We will proceed carefully, for the encoding is a bit unintuitive, mainly because the computation as simulated by operators is *widely parallel*.

> "*[O]ur modelisation is without* global external time, *i.e. without any kind of synchronisation. Due to this lack of synchronisation, some 'actions at distance' familiar to the syntax cannot be performed [...]. We have therefore gained a new freedom w.r.t. time, and this should be essential in view of parallel execution.*"
>
> (Girard [48, p. 223])

This freedom regarding time is for the moment hard to capture, and that is one of the reasons we are going to simulate non-deterministic log-space computation, in a highly parallel way.

Recalling the framework

The setting is as we left it in Chapter 3: we take the binary representation of integers N_n as operators in $\mathrm{M}_6(\mathfrak{N}_0)$ (Definition 3.4.1), we consider only the normative pair $(\mathfrak{N}_0, \mathfrak{G})$ as defined in Corollary 3.4.4 and we take the sets of observations $P_{\geqslant 0}$, P_+ and $P_{+,1}$ defined in Definition 3.4.7.

Intuitively, the encoding of $M = \{Q, \to\}$ a PM with p pointers and an initial pseudo-configuration s will be an observation $M_s^\bullet$, which is an operator of $\mathrm{M}_6(\mathfrak{G}) \otimes Q_M$, where the algebra of states[18] Q_M is

$$Q_M = \underbrace{\mathrm{M}_6(\mathbb{C}) \otimes \mathrm{M}_6(\mathbb{C}) \otimes \cdots \otimes \mathrm{M}_6(\mathbb{C})}_{p \text{ times}} \otimes \mathrm{M}_{q'}(\mathbb{C})$$

This algebra Q_M represents a set of *pseudo-states*, i.e. it will encode the memory slots that store the last p values read by the pointers, and the extended set of states $Q^\uparrow$. We extend the set of states Q of M with

[18] That should rather be called the *algebra of pseudo-states*, but we wanted to stay consistent with the notation of Chapter 3, and, besides, the state will also be encoded in it.

some intermediate states needed to encode the basic operations of M properly. We let $\mathrm{Card}(Q^{\uparrow}) = q'$ and will call *an extended pseudo-configuration* an element of $\{0,1,\star\}^p \times Q^{\uparrow}$.

The intuition is that the jth copy of $\mathrm{M}_6(\mathbb{C})$ represents the memory slot that contains the last value read by the jth pointer. We will therefore distinguish for each copy of $\mathrm{M}_6(\mathbb{C})$ a basis $(0o, 0i, 1o, 1i, s, e)$ corresponding to the different values that a pointer can read.[19] To sum up, the distinguished basis of Q_M considered will be denoted by tuples $(a_1, \ldots, a_p, \mathbf{q})$. Notice that such a tuple naturally corresponds to an extended pseudo-configuration.

A crucial remark should be made to explain properly how the computation will be performed. Recall that the input of $M_s^{\bullet}$ will be $N_n \otimes \mathrm{Id}_{Q_M}$, i.e. the tensoring of N_n with the unit of the algebra of states. For the product $M_s^{\bullet}(N_n \otimes \mathrm{Id}_{Q_M})$ to be nilpotent means to be nilpotent *for any basis of* Q_M, that is, for any possible initial pseudo-configuration.

Stated differently, the integer is considered at the same time in every possible pseudo-configuration. As a result, the computation represented by the sequence

$$M_s^{\bullet}(N_n \otimes \mathrm{Id}_{Q_M}), (M_s^{\bullet}(N_n \otimes \mathrm{Id}_{Q_M}))^2, \ldots$$

somehow simulates for all $c \in C_M$ the computations of $M_c(n)$ simultaneously. Of course, this applies only for the initialisation: once the first pseudo-configuration is fixed, the computation is led according to it. But among all the computations started in parallel, only one effectively has for initial pseudo-configuration s.

The representation of reject cannot be done without considering an initial pseudo-configuration, because if the computation starts with the "wrong" pseudo-configuration, we will under certain circumstance start it again with the "good" initial pseudo-configuration, i.e. s. This will be explained when we will encode rejection.

Encoding a machine

In fact, our operator could be conceived as having a single pointer, and p registers. This explains the "one movement at a time" motto, our single pointer will travel a lot, but this is algorithmically equivalent to having p pointers. Girard [59, p. 262] refers to this oddity as if we had at our disposal a "thumb" and several fingers. Only the thumb can move, but it can exchange its position with any other finger.

We know how to access the values of N_n thanks to projections, but we have to handle two difficulties: that every other bit is redundant, and that the instruction will depend on the previous projection, "go deeper" or "lift back to the surface". What we mean is that changing the direction of the movement will not be trivial: we will handle differently a movement from left to right on the string (a "forward move") and a movement from right to left (a "backward move"), and furthermore we will handle them differently depending on the previous orientation of the movement. Decomposing this movement and handling the different case will require some intermediate steps and *extended states*.

We will encode the transition function[20] of M as if it were a function between C_M and $C_M \cup \{\mathbf{accept}, \mathbf{reject}\}$. This could seem really surprising, for establishing the pseudo-configuration c' reached

[19] As Thomas Seiller points it, we should rather consider those values dynamically: our observation will always be on the edge of reading the next value, so reading an occurrence of $0o$ rather corresponds to reading the input of the value connected to this output of a 0.

[20] We continue for the sake of clarity to speak of $\to$ as if it was a function, but it is of course eventually a relation, in the non-deterministic case.

after applying the transition $\to$ from a pseudo-configuration c needs to take care of the actual input and of the positions of the pointers.[21] Here the actual position of the pointers and the reading of the input will be handled differently: the actual operation of reading the input and moving the pointers will be handled directly within the transition, i.e. c' will be reached iff the values read correspond to a "legal move" from c to c'. So to say, we will skip the "instruction part" to go directly to the pseudo-configuration resulting from the movement of *the* pointer and the reading of *the* new value.[22]

So we will encode each couple (c, c') of pseudo-configuration such that "nothing in $\to$ prevents c' to be reached from c" by an operator $\phi_{c,c'}$. How can we know that $c \to c'$ is possible without knowing nor the input nor the position of the heads? Well, every time there is $c \to (i_1, \dots, i_p, \mathbf{q'})$ for i instructions $i_1, \dots, i_p \in I_{\{1,\dots,p\}}$, we consider that every pseudo-configuration c' whose state is $\mathbf{q'}$ can be reached from c. The following will explain how those pseudo-configurations are sorted between the one that are actually reachable considered an input and the one that are not.

All the possibles transitions from the pseudo-configuration c are then encoded by

$$\sum_{c \to c'} \phi_{c,c'}$$

and the encoding of the transition relation will then correspond to the sum:

$$M_s^\bullet = \sum_{c \in C_M} \sum_{c' \text{ s.t. } c \to c'} \phi_{c,c'} + \text{reject}_s \tag{4.10}$$

As we may see, $M_s^\bullet$ is entirely defined by the encoding of the transition function of M, decomposed in all its possible applications. Every transition from every pseudo-configuration will be encoded in the sum, and a built-in way of assuring that it corresponds to the actual position of the pointers and the input will discriminate between the transitions that are actually possible and the one that aren't.

In fact the instructions to move a pointer will be performed by the selection of the pointer, a projection onto the values that may have been read, the storing of the value read, the deactivation of the pointer and the changing of state.

Notations

Before describing the encoding in details, we need to introduce several notations that will be used for the definition of the $\phi_{c,c'}$ operators. The von Neumann algebra $\mathrm{M}_6(\mathfrak{G}) \otimes Q_M$ will be considered as $\mathrm{M}_6(\mathbb{C}) \otimes \mathfrak{G} \otimes Q_M$ and we will define the operators needed to encode the basic operation of M as tensor products $u \otimes v \otimes w$, where $u \in \mathrm{M}_6(\mathbb{C})$, $v \in \mathfrak{G} \subseteq \mathfrak{R}$ and $w \in Q_M$. We will explain the main patterns of those three operators.

Some of these definitions are tiresome, because the values of N_n are encoded in the pattern we defined in Chapter 3, and so every bit comes in a "input" and "output" version. That is to say, if we remember Figure 3.4 and more generally Section 3.3, that $\star a_k \dots a_1 \star$ is in fact encoded as $e a_{i1} a_{o1} \dots a_{ik} a_{ok} s$. Moreover, we need to cope[23] with the "last direction of the pointer" to adapt cleverly the projections.

The operator $u \in \mathrm{M}_6(\mathbb{C})$ will alternatively read a value, and enforce the direction. We define the projections $\pi_\blacklozenge$ of $\mathrm{M}_6(\mathbb{C})$ for $\blacklozenge \in \{0i, 0o, 1i, 1o, e, s\}$ as the projections on the subspace induced by the

[21] Or, equivalently, of the position of our "traveler" and of the values of the registers.

[22] For M move only one pointer at a time.

[23] This comes from a concern regarding the norm of the operators we are going to define, something we did not bother with in our first work [10, p. 18]. In this work, [in] and [out] directly encodes the movement, regardless of the orientation of the previous movement.

basis element ⧫. We will sometimes denote e (resp. s) by $\star i$ (resp. $\star o$). The reading part will be essentially handled by those projections.

The direction is managed by

1. $\pi_{\text{in}} = \pi_{0i} + \pi_{1i} + \pi_e$ and $\pi_{\text{out}} = \pi_{0o} + \pi_{1o} + \pi_s$. They are such that $\pi_{\text{in}}\pi_{\text{out}} = \pi_{\text{out}}\pi_{\text{in}} = 0$.

2. $[\text{in} \to \text{out}]$ and $[\text{out} \to \text{in}]$ are such that

$$\begin{array}{ll} [\text{in} \to \text{out}][\text{out} \to \text{in}] = \pi_{\text{out}} & [\text{out} \to \text{in}][\text{in} \to \text{out}] = \pi_{\text{in}} \\ [\text{out} \to \text{in}]\pi_{\text{out}} = [\text{out} \to \text{in}] & [\text{in} \to \text{out}]\pi_{\text{in}} = [\text{in} \to \text{out}] \\ [\text{out} \to \text{in}]\pi_{\text{out}} = 0 & [\text{in} \to \text{out}]\pi_{\text{in}} = 0 \end{array}$$

They are "the memory of the direction of the previous movement", but one should notice that they are not projections, i.e. $[\text{out} \to \text{in}]^2 = [\text{in} \to \text{out}]^2 = 0$.

We can express those operators as matrices, to ease the checking of the previous equalities:

$$\pi_{\text{in}} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad [\text{in} \to \text{out}] \equiv \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\pi_{\text{out}} \equiv \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad [\text{out} \to \text{in}] \equiv \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The operator $v \in \mathfrak{G} \subset \mathfrak{R}$ will in fact always be a permutation. We will only need in the following $\tau_{i,j}$, the transposition exchanging the integers i and j.[24]

The last operator $w \in Q_M$ will encode an updating in the registers and a changing of state, i.e. the evolution from an extended pseudo-configuration $c = (a_1, \ldots, a_p, \mathbf{q})$ to an extended pseudo-configuration $c' = (a'_1, \ldots, a'_p, \mathbf{q}')$. We define the partial isometry:

$$(c \to c') = (a_1 \to a'_1) \otimes \cdots \otimes (a_p \to a'_p) \otimes (\mathbf{q} \to \mathbf{q}')$$

where $(v \to v')$, for $v, v' \in \{0i, 0o, 1i, 1o, e, s\}$ or $v, v' \in Q^{\uparrow}$, is simply defined as:

$$(v \to v') \coloneqq \begin{array}{c} \\ \\ \\ \\ \\ \\ v' \end{array} \overset{\begin{array}{c} \\ \\ \\ \\ \\ \\ v \end{array}}{\begin{pmatrix} 0 & \ldots & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ 0 & \ldots & 1 & \ldots & 0 \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & \ldots & 0 \end{pmatrix}}$$

[24] To be more precise, $\tau_{i,j}$ is the unitary induced by the transposition $\begin{pmatrix} i & j \end{pmatrix}$.

We should remark that for all $c, c', c" \in C_M$,

$$(c \to c')c" = \begin{cases} c' & \text{if } c' = c" \\ \\ 0 & \text{elsewhere} \end{cases} \tag{4.11}$$

A transition will always impact on at most one value stored in the registers and on the state $\mathbf{q}$. Suppose it acts on the the jth register, we define

$$([a_j \to a'_j]_j, \mathbf{q} \to \mathbf{q'})$$

as

$$(\mathrm{Id}, \ldots, (a_j \to a'_j), \ldots, \mathrm{Id}, \mathbf{q} \to \mathbf{q'})$$

It is of course possible to encode a transition that let the state unchanged: just let $\mathbf{q} \to \mathbf{q}'$ be Id if $\mathbf{q} = \mathbf{q}'$.

In practise, we won't need to be specific about a_j *and* a'_j at the same time. All that will matter will be the value stored *or* the value we are going to store in the jth register, for we will decompose the action of reading the value and the action of storing it in two steps.[25]

We will use two shorthands to encode the actions on the jth register. For $* \in \{0, 1, \star\}$ and $d \in \{o, i\}$, we define

$$[\to *d]_j = \sum_{\blacklozenge \in \{0i, 0o, 1i, 1o, e, s\}} [\blacklozenge \to *d]_j$$

$$[*d \to]_j = \sum_{\blacklozenge \in \{0i, 0o, 1i, 1o, e, s\}} [*d \to \blacklozenge]_j$$

At last, we will sometimes let the $*$ of the previous equation be *a subset* of $\{0, 1, \star\}$.

This somehow complicated way of writing the operators could be explained with two examples:

$[\{0i, 1i, \star i\} \to]_j$ should be read as "no matter what the previous value stored in the jth slot was, we just make sure that it came from a forward move, i.e. it is endowed with a i".

$[\to 0i]_j$ will simply be "store a $0i$ in the jth slot, no matter what the previous stored value was".

The left hand side of the arrow tells us something about the value a_j of c, the right hand side of the arrow tells us something about the value a'_j of c'.

We are now ready to define the operators needed to encode the basic operations of the PM. acceptance will be especially easy, but basic movement and rejections are a more complex sum of operators defined in Figure 4.3 and Figure 4.4.

Basic operations

From now on, we consider given M a PM with p pointers and $c = (a_1, \ldots, a_p, \mathbf{q}) \in C_M$ a pseudo-configuration. An initial pseudo-configuration $s \in C_M$ will be needed to encode rejection. There are only three cases in M: either $c \to c'$ for c' a pseudo-configuration (but remark that c' is not unique if M is non-deterministic), $c \to$ **accept**, or $c \to$ **reject**. We will define the encoding relatively to those three cases.

$$
\begin{aligned}
\mathrm{bf}^{c}_{j,\mathbf{q}'} &= \pi_{\mathrm{out}} \otimes \tau_{0,j} \otimes ([\{0o, 1o, \star o\} \to]_j, \mathbf{q} \to \mathbf{mov}^{\mathbf{q},\mathbf{q}'}_j)\\
\mathrm{ff}^{c}_{j,\mathbf{q}'} &= [\mathrm{in} \to \mathrm{out}] \otimes \tau_{0,j} \otimes ([\{0i, 1i, \star i\} \to]_j, \mathbf{q} \to \mathbf{mov}^{\mathbf{q},\mathbf{q}'}_j)\\
\mathrm{fb}^{c}_{j,\mathbf{q}'} &= \pi_{\mathrm{in}} \otimes \tau_{0,j} \otimes ([\{0i, 1i, \star i\} \to]_j, \mathbf{q} \to \mathbf{mov}^{\mathbf{q},\mathbf{q}'}_j)\\
\mathrm{bb}^{c}_{j,\mathbf{q}'} &= [\mathrm{out} \to \mathrm{in}] \otimes \tau_{0,j} \otimes ([\{0o, 1o, \star o\} \to]_j, \mathbf{q} \to \mathbf{mov}^{\mathbf{q},\mathbf{q}'}_j)\\
\mathrm{frec}^{c}_{j,\mathbf{q}'} &= \textstyle\sum_{*\in\{0,1,\star\}} \big(\pi_{*i} \otimes \tau_{0,j} \otimes ([\to *i]_j, \mathbf{mov}^{\mathbf{q},\mathbf{q}'}_j \to \mathbf{q'})\big)\\
\mathrm{brec}^{c}_{j,\mathbf{q}'} &= \textstyle\sum_{*\in\{0,1,\star\}} \big(\pi_{*o} \otimes \tau_{0,j} \otimes ([\to *o]_j, \mathbf{mov}^{\mathbf{q},\mathbf{q}'}_j \to \mathbf{q'})\big)
\end{aligned}
$$

$\mathrm{bf}^{c}_{j,\mathbf{q}'}$ encodes "if the previous movement was a backward move, change the direction, activate the jth pointer, erase the jth register".
$\mathrm{ff}^{c}_{j,\mathbf{q}'}$ encodes the same operation in the case where the previous movement was already a forward move.
$\mathrm{fb}^{c}_{j,\mathbf{q}'}$ and $\mathrm{bb}^{c}_{j,\mathbf{q}'}$ encodes the same couple of operations for a backward movement.
$\mathrm{frec}^{c}_{j,\mathbf{q}'}$ is made of a sum over $* \in \{0, 1, \star\}$, and every of its member encode "put the jth pointer back to its position, store $*$ in the jth register and adopt state $\mathbf{q'}$." $\mathrm{brec}^{c}_{j,\mathbf{q}'}$ encodes the same operation for the other direction.

Figure 4.3: The operators encoding the forward and backward move instructions

Moving a pointer, reading a value and changing the state Suppose that M, in pseudo-configuration $c = (a_1, \ldots, a_p, \mathbf{q})$, moves the jth pointer right, reads the value a'_j stored at $\sharp p_j$, updates the jth memory slot in consequence and changes the state to $\mathbf{q}'$. The new pseudo-configuration is then $c' = (a_1, \ldots, a_{j-1}, a'_j, a_{j+1}, \ldots, a_p, \mathbf{q}')$.

There are two cases: either the last movement of the jth pointer was a forward move or it was a backward move. The case we are dealing with is obtained from the value stored in the jth memory slot: if the value is $0i$, $1i$ or $\star i$, then the last move was a forward move, if the value is $0o$, $1o$ or $\star o$, the last move was a backward move. In the first case, the operator $\mathrm{ff}^{c}_{j,\mathbf{q}'}$ will be applied, and in the second the $\mathrm{bf}^{c}_{j,\mathbf{q}'}$ operator will be applied. Remark that we have to make the sum of those two operators, for we cannot foresee what was the direction of the last movement. The fact that the "wrong case" gets to 0 should be evident from the definitions of $[\mathrm{in} \to \mathrm{out}]$, $[\mathrm{out} \to \mathrm{in}]$, π_{in} and π_{out}.

Both these operators somehow *activate* the jth pointer by using the transposition $\tau_{0,j}$ and prepare the reading of the representation of the integer. This representation will then give the value of the next (when moving forward) digit of the input. The $\mathrm{frec}^{c}_{j,\mathbf{q}'}$ or the $\mathrm{brec}^{c}_{j,\mathbf{q}'}$ operator is then applied in order to simultaneously update the value of the jth memory slot and *deactivate* the jth pointer. Remark that this careful decomposition requires the introduction of a new (i.e. extended) state $\mathbf{mov}^{\mathbf{q},\mathbf{q}'}_j$.

To sum up, we let

$$\mathrm{forward}^{c}_{j,\mathbf{q}'} = \mathrm{bf}^{c}_{j,\mathbf{q}'} + \mathrm{ff}^{c}_{j,\mathbf{q}'} + \mathrm{frec}^{c}_{j,\mathbf{q}'} \qquad \mathrm{backward}^{c}_{j,\mathbf{q}'} = \mathrm{fb}^{c}_{j,\mathbf{q}'} + \mathrm{bb}^{c}_{j,\mathbf{q}'} + \mathrm{brec}^{c}_{j,\mathbf{q}'}$$

We insist one last time: this sum is here to cope with all the possible situations regarding the orientation of the previous movement of the jth pointer and the value actually read. All the "incorrect situations" will get cancelled and only the "legal transition between two pseudo-configurations " will be applied.

In the following, we will forget about the subscripts and superscripts of these operators and write ff, fb, bf, bb, and rec for bref or frec.

[25] That will explain the introduction of additional states to handle this intermediate step.

This —by the way— justifies once again the "one movement at every transition" (item 2), for we apply only one permutation at every transition.

Accept The case of acceptance is especially easy: we want to stop the computation, so every transition $(a_1,\dots,a_n,\mathbf{q}) \to \mathbf{accept}$ will be encoded by 0.

This justifies the item 7 of the previous section: we need the transition function to be always defined. If there is a pseudo-configuration h such that $h \to \emptyset$, this will be encoded by the operators as $h \to 0$, hence the default behaviour is to accept when the function is not defined.

So if the computation of M does not lead to acceptance, it is because M rejects (thanks to the "always stops" item 4). On the other hand, if the computation of $M_s^\bullet$ halts, it is in any case because it accepts, so we want to mimic the rejection by a loop. An attempt to define a naive "loop projection" fails: to illustrate it, we define an observation $\mathrm{reject}_{\mathrm{naive}} \in \mathrm{M}_6(\mathbb{C}) \otimes \mathfrak{G} \otimes Q_M$ that does nothing but reject, thanks to a yet-to-be-defined "loop projection" π_{reject}:

$$\mathrm{reject}_{\mathrm{naive}} = \mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes \mathrm{Id}_{\mathfrak{R}} \otimes \pi_{\mathrm{reject}}$$

It succeeds to make the computation loops, as for all $d \in \mathbb{N}$ and N_n:

$$\begin{aligned}((N_n \otimes \mathrm{Id}_{Q_M})\mathrm{reject}_{\mathrm{naive}})^d &= ((N_n \otimes \mathrm{Id}_{\mathfrak{R}} \otimes \mathrm{Id}_{Q_M})\mathrm{Id}_{\mathrm{M}_6(\mathbb{C})} \otimes \mathrm{Id}_{\mathfrak{R}} \otimes \pi_{\mathrm{reject}})^d \\ &= (N_n \otimes \pi_{\mathrm{reject}})^d \\ &= N_n^d \otimes \pi_{\mathrm{reject}}\end{aligned}$$

And we know that N_n is not nilpotent, so neither is $(N_n \otimes \mathrm{Id}_{\mathfrak{R}})\mathrm{reject}_{\mathrm{naive}}$: this attempt succeeds in creating a loop.

On the other hand, as the encoding of $\to$ is built as a sum of the basic operations, π_{reject} appears in it, and $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ cannot be nilpotent. So this solution is excessive and we have to find another way to be sure that the operator will loop if *and only if* the operator that simulates the reject is reached. To do so, we build an operator that simulates the reinitialisation of M in pseudo-configuration s with its p pointers on $\star$ when reached.

It comes with another advantage: remember that the computation of $M_s^\bullet$ started in all possible configurations, not only in s, so in all the other cases, if reject was reached after starting the computation with a pseudo-configuration $c' \neq s$, we enforce the computation of the machine on s. As a consequence, if $M_s(n)$ accepts, the rejection of $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ that corresponds to a computation on c' will be temporary: once rejection attained, the computation restarts with pseudo-configuration s and will therefore halts accepting. The fact that $c' \neq s$ could lead to acceptance is not relevant: as we said (item 9), "acceptance is the default behaviour", what really matters is rejection.

Reject Rejection should lead to the creation of a loop, in fact it is encoded by "put the p pointers back to $\star$ and starts the computation again with initial pseudo-configuration s". This is why we needed the initial pseudo-configuration s to be a parameter, for the reject_s operator will depend of it. We also need two extended states, $\mathbf{back}_i$ and $\mathbf{mov\text{-}back}_i$ for each $i = 1,\dots,p$.

The transitions of the form $(a_1,\dots,a_n,\mathbf{q}) \to \mathbf{reject}$ are encoded by the operator that represents the "transition" $(a_1,\dots,a_n,\mathbf{q}) \to (a_1,\dots,a_n,\mathbf{back}_1)$.

We can then define

$$\mathrm{reject}_s = \sum_{i=1}^{p} (\mathrm{rm}_i + \mathrm{rr}_i^0 + \mathrm{rr}_i^1 + \mathrm{rc}_i)$$

Remember we are given an initial pseudo-configuration $s = (a_1, \dots, a_n, \mathbf{q}_0)$.

$$
\begin{array}{rcl}
\mathrm{rm}_i = & \mathrm{Id} \otimes \tau_{0,i} \otimes & (\mathbf{back}_i \to \mathbf{mov\text{-}back}_i) \\
\mathrm{rr}_i^0 = & \pi_{0_o} \otimes \tau_{0,i} \otimes & ([\to 0_o]_i, \mathbf{mov\text{-}back}_i \to \mathbf{back}_i) \\
\mathrm{rr}_i^1 = & \pi_{1_o} \otimes \tau_{0,i} \otimes & ([\to 1_o]_i, \mathbf{mov\text{-}back}_i \to \mathbf{back}_i) \\
\mathrm{rc}_i = & \begin{cases} \pi_\star \otimes \tau_{0,i} \otimes ([\to a_i]_i, \mathbf{mov\text{-}back}_i \to \mathbf{back}_{i+1}) & (1 \leqslant i < p) \\ \pi_\star \otimes \tau_{0,p} \otimes ([\to a_p]_p, \mathbf{mov\text{-}back}_p \to \mathbf{q}_0) & (i = p) \end{cases}
\end{array}
$$

rm_i encodes "if you are in state $\mathbf{back}_i$, no matter what you read, activate the ith pointer and go to state $\mathbf{mov\text{-}back}_i$".
rr_i^0 (resp. rr_i^1) encodes "if you read a 0 (resp. a 1), deactivates the ith pointer, store the value you read and go to state $\mathbf{back}_i$".
rc_i encodes "If you read $\star$, deactivates the ith pointer, put the ith value of the initial pseudo-configuration s in the ith register, and change state to $\mathbf{mov\text{-}back}_{i+1}$. If you were dealing with the last pointer, go to the state $\mathbf{q}_0$ provided by the initial pseudo-configuration."

Figure 4.4: The operators encoding rejection

How does this operator encodes a reinitialization of M? The operator rm_i initializes the process, rr_i^0 and rr_i^1 encodes "as long as the ith pointer is reading a 0 or a 1, keep moving it backward". We are forced to alternate between the extended states $\mathbf{back}_i$ and $\mathbf{mov\text{-}back}_i$ to handle the redundancy of the input. When $\star$ is read by the ith pointer, the actual value of the initial pseudo-configuration s is stored in the corresponding register and rc_i starts this process for the $i+1$th pointer.

Lemma 4.4.1. *For all $k \in \mathbb{N}$, for all $M \in \mathrm{NPM}(k)$, for all $s \in C_M$ an initial pseudo-configuration, there exists an operator $M_s^\bullet$ such that for all $n \in \mathbb{N}$ and every binary representation $N_n \in \mathrm{M}_6(\mathfrak{N}_0)$ of n, $M_s(n)$ accepts iff $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ is nilpotent.*

Proof. We obtain $M_s^\bullet$ thanks to the encoding we just defined. We can remark that $M_s^\bullet \in P_+$, and as we are acting in a normative pair, it does not matter which representation N_n of n we pick. So let us fix $n \in \mathbb{N}$ and N_n one of its binary representation.

Considering the representation of the rejection it is clear that if a branch of $M_s(n)$ rejects, the operator $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ will not be nilpotent, so we just have to prove that if $M_s(n)$ accepts then $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ is nilpotent.

We prove its reciprocal: suppose $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ is not nilpotent. In this product N_n is given to the operator $M_s^\bullet$ that starts the simulation of the computation of M with input n in every possible initial pseudo-configuration at the same time. Since the encoding of M is built with respect to s, we know that there exists a j such that $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})\pi_j$ is the simulation of $M_s(n)$, but the computation takes place in the other projections too: for $i \neq j$ it is possible that $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})\pi_i$ loops where for a d $(M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M}))^d \pi_j = 0$.

We can handle this behaviour because for all $c \in C_M$, M_s always halts. This is due to our "built-in clock", that insures us that M will always stops, no matter what the initial configuration considered is. So if $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ is not nilpotent it is because at some point the **reject** state has been reached. After this state is reached (let's say after $r \in \mathbb{N}$ iterations) we know that $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})^r \pi_i$ is exactly the simulation of $M_s(n)$. If it loops again, it truly means that $M_s(n)$ rejects.

So we just proved that $M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M})$ is not nilpotent iff $(M_s^\bullet(N_n \otimes \mathrm{Id}_{Q_M}))^d \pi_j \neq 0$ for all $d \in \mathbb{N}$. But it is clear that in this case $M_s(n)$ rejects. **Q.E.D.**

Proposition 4.4.2.

$$\textbf{co-NL} \subseteq \{P_+\} \subseteq \{P_{\geqslant 0}\}$$

Proof. Even if the previous simulation may be hard to grasp, to output the description of $M_s^\bullet$ form $M = \{\to, Q\} \in \mathrm{NPM}(p)$ is $\mathbf{AC}^0$: all we have to do is to list all the $3^p \times \mathrm{Card}(Q)$ possible pseudo-configurations of M, and to output $\phi_{c,c'}$ for every of them. The pseudo-configuration c' is simply obtained by looking in $\to$ if there is a transition from c to a set of instructions (no matter what they are) endowed with the state of c'. Outputting the members of the sum can obviously be done in parallel, for it relies on simple routines. The reject_s operator is simply a constant factor in the processing.

Thus, the previous proposition tells us that $\mathbf{NPM} \subseteq \{P_+\}$ since the representation $M_s^\bullet$ of a couple (M, s), where M is an NPM and $s \in C_M$, is obviously in P_+. Moreover, since $P_+ \subset P_{\geqslant 0}$, we have $\{P_+\} \subseteq \{P_{\geqslant 0}\}$. The Corollary 4.3.2 gives us the last element to conclude, i.e. that $\textbf{co-NL} \subseteq \mathbf{NPM}$. **Q.E.D.**

The deterministic case

This subsection is dedicated to the somehow tedious verification that if M is a DPM, then $M_s^\bullet$ will be of 1-norm (Definition 3.4.6) less than 1. This will conclude this chapter by proving that operators can also mimic deterministic computation.

Proposition 4.4.3. *Let A, B be operators such that $AB^* = B^*A = 0$. Then:*

$$||A + B|| = \max\{||A||, ||B||\}$$

Proof. Actually, this proposition is proved by Maher [99, Theorem 1.7 (b)], but stated differently, with conditions on ranges.[26] It is proven that if $\mathrm{Ran}(A) \perp \mathrm{Ran}(B)$ and $\mathrm{Ran}(A^*) \perp \mathrm{Ran}(B^*)$, then $||A + B|| = \max\{||A||, ||B||\}$.

To obtain the result as we stated it, simply notice that $AB^* = 0$ implies $\mathrm{Ran}(B^*) \subseteq \ker(A) = (\mathrm{Ran}(A^*))^\perp$ so $\mathrm{Ran}(B^*) \perp \mathrm{Ran}(A^*)$. Similarly, $B^*A = 0$ implies that $\mathrm{Ran}(A) \subseteq \ker(B^*) = (\mathrm{Ran}(B))^\perp$ so $\mathrm{Ran}(A) \perp \mathrm{Ran}(B)$. Thus, if $AB^* = B^*A = 0$, we have $\mathrm{Ran}(A) \perp \mathrm{Ran}(B)$ and $\mathrm{Ran}(A^*) \perp \mathrm{Ran}(B^*)$ and one can then apply the theorem cited. **Q.E.D.**

Corollary 4.4.4. *The operators ff + bf, rec, bb + fb, and $\mathrm{rr}_i^0 + \mathrm{rr}_i^1 + \mathrm{rc}_i$ (i fixed) are of norm* 1.

Proof. The proof is in fact reduced to a simple verification, by looking at the projections in the last algebra of the operators we defined, recalling that $\pi_a \pi_b = 0$ when $a \neq b$ and using the previous proposition.

Since $\pi_{\mathrm{out}} \pi_{\mathrm{in}} = \pi_{\mathrm{in}} \pi_{\mathrm{out}} = 0$, we have that $\mathrm{bf} \times \mathrm{ff}^*$ and $\mathrm{ff}^* \times \mathrm{bf}$ are equal to 0. Thus, using the preceding proposition, we have $||\mathrm{ff} + \mathrm{bf}|| = \max\{||\mathrm{bf}||, ||\mathrm{ff}||\} = 1$. A similar argument shows that $||\mathrm{fb} + \mathrm{bb}|| = 1$.

Clearly, as $\pi_{ai} \pi_{bi} = \pi_{ao} \pi_{bo} = 0$ for $a \neq b$, we deduce by Proposition 4.4.3 that $||\mathrm{rec}|| = 1$.

Concerning $\mathrm{rr}_i^0 + \mathrm{rr}_i^1 + \mathrm{rc}_i$, we fix $1 \leqslant i \leqslant p$ and notice that $\mathrm{rc}_i \times (\mathrm{rr}_i^0)^* = 0$ as a consequence of $\pi_{\textbf{mov-back}_i} \times \pi_{\textbf{back}_i} = 0$. Conversely, $(\mathrm{rr}_i^0)^* \times \mathrm{rc}_i = 0$ as a consequence of[27] $\pi_{\textbf{back}_{i+1}} \times \pi_{\textbf{mov-back}_i} = 0$. A similar argument shows that $\mathrm{rc}_i \times (\mathrm{rr}_i^1)^* = 0$ and $(\mathrm{rr}_i^1)^* \times \mathrm{rc}_i = 0$. Moreover, we know that $(\mathrm{rr}_i^0)^* \times \mathrm{rr}_i^1 = 0$ and $\mathrm{rr}_i^1 \times (\mathrm{rr}_i^0)^* = 0$ by just looking at the first term of the tensor in their definition. By an iterated use of Proposition 4.4.3, we get that $||\mathrm{rr}_i^0 + \mathrm{rr}_i^1 + \mathrm{rc}_i|| = \max\{||\mathrm{rr}_i^0||, ||\mathrm{rr}_i^1||, ||\mathrm{rc}_i||\} = 1$. **Q.E.D.**

[26] The range being the image or co-domain, i.e. $\mathrm{Ran}(B) = \{x \mid \exists y By = x\}$. The kernel is defined as $\ker(A) = \{x \mid Ax = 0\}$.

[27] When $i = p$, we consider that $\mathbf{q}_0 = \textbf{back}_{i+1}$ to ease the argument.

We are now in possession of all the material needed to study the 1-norm of the operator obtained by encoding of a deterministic pointer machine.

Proposition 4.4.5. *Let M be a DPM, s an initial configuration and $M_s^\bullet$ its encoding, then $||M_s^\bullet||_1 \leqslant 1$.*

Proof. Since we are working with deterministic machines, the transition relation is functional: for each $c \in C_M$ there is exactly one $t \in C_M$ —as the transition function is "total"— such that $c \to t$. Thus, Equation 4.10 becomes in this setting:

$$M_s^\bullet = \sum_{c \in C_M} \phi_{c,t} + \text{reject}_s$$

Since $M_s^\bullet$ is isomorphic to an element of $\mathrm{M}_N(\mathrm{M}_6(\mathfrak{R}))$, we will now compute $||M_s^\bullet||_1^N$. We first show that this matrix has at most one non-null coefficient in each column. Then we will use the fact that these coefficients are of norm at most 1.

To prove this we introduce the set $P = \{1,\dots,6\}^p \times Q^\uparrow$ of *extended pseudo-configurations*, where $1,\dots,6$ are a shorthand for $\mathtt{0}i,\dots,\star o$. This set is a basis of the algebra of pseudo-states, and we write $M_s^\bullet = (\mathrm{a}[c,c'])_{c,c' \in P}$. This way, the 1-norm of $M_c^\bullet$ is defined as

$$||M_s^\bullet||_1^N = \max_{c' \in P} \Big\{ \sum_{c \in P} ||\mathrm{a}[c,c']|| \Big\}$$

Let $c = (a_1,\dots,a_n,\mathbf{q})$ be a (non-extended) pseudo-configuration. If c is a pseudo-configuration of the machine (i.e. $\mathbf{q}$ is an element of Q), then there is at most one pseudo-configuration t such that $c \to t$. This *atomic transition* can be:

Accept The operator $\phi_{c,t}$ is equal to $\mathtt{0}$ and the column is empty.

Reject The column corresponding to c contains only the operator $(\mathbf{q} \to \mathbf{back}_1)$ that encodes the transition $c \to (a_1,\dots,a_n,\mathbf{back}_1)$. And the norm of this operator is equal to 1.

Move forward the jth pointer and change state to q' The only extended pseudo-configurations introduced by the encoding is $\tilde{c} = (a_1,\dots,a_n,\mathbf{mov}_j^{\mathbf{q},\mathbf{q}'})$. The column corresponding to c (resp. $\tilde{c}$) contains the operator $\mathrm{ff} + \mathrm{bf}$ (resp. rec), which is of norm 1 by Corollary 4.4.4.

Move backwards... This case is similar to the previous one.

Now we consider the extended pseudo-configurations, they are all introduced by the operator reject_s, and for $i = 1,\dots,p$ they are all of one of the following form:

$$\bar{c}_i^m = (a_1,\dots,a_n,\mathbf{mov\text{-}back}_i) \qquad \bar{c}_i^b = (a_1,\dots,a_n,\mathbf{back}_i)$$

The column corresponding to c (resp. $\bar{c}_i^b$, $\bar{c}_i^m$) contains only the operator encoding the transition from c to $\bar{c}_1^b$ (resp. the operator rm_i, the operator $\mathrm{rr}_i + \mathrm{rc}_i$), which is a norm 1 operator by Corollary 4.4.4.

We just showed that each column of the matrix contains at most one operator different from $\mathtt{0}$, and that this operator is always of norm 1. Hence, for any extended pseudo-configuration $c \in P$:

$$\sum_c \mathrm{a}[c,c'] \leqslant 1$$

As a consequence of the definition of the 1-norm (Definition 3.4.6), we get

$$||M_s^\bullet||_1^N = \max_{c' \in P} \Big\{ \sum_{c \in P} \mathrm{a}[c, c'] \Big\} \leqslant 1$$

Q.E.D.

Proposition 4.4.6.

$$\mathbf{L} \subseteq \{P_{+,1}\}$$

Proof. By Corollary 4.3.2 we know that $\mathbf{L} \subseteq \mathbf{DPM}$ and the previous proposition shows that the encoding of a DPM is of 1-norm inferior to 1, i.e. belongs to $P_{+,1}$. **Q.E.D.**

Theorem 4.4.7.

$$\textbf{co-NL} = \{P_+\} = \{P_{\geqslant 0}\} \textit{ and } \mathbf{L} = \{P_{+,1}\}$$

Proof. By combining Proposition 4.4.2, Proposition 4.4.6 and Corollary 3.5.4 from previous chapter. **Q.E.D.**

Results and perspectives

What have we done?

We presented several "pointers machines" to break the myth that "Pointers are log-space": without any further assumption on the nature of the pointers and the way they are manipulated, this statement is not correct. JAG, PURPLE, SMM and KUM illustrate the way pointers may be handled as pebbles or data structure, without characterizing **L**.

We then focused on a special kind of Finite Automata, that is considered as the better way to capture "pointer computation" as a Turing Machine would do it. We did not considered its alternating variant, but proved several classical claims about them. This was done to ease the introduction of Pointer Machines, which are similar on many points, but should rather be seen as universally non-deterministic. This machine model is introduced as a suitable model to be embedded in the operators.

The last section describes in its finest details this simulation, and encodes the basic operations of a PM so that they fulfil a constrain on their norm. The encoding of the rejection is somehow the more complicated part, for the only way to mimic rejection is to make the computation loop. Due to some oddity of the operator presentation, the simulation starts in parallel in all the possible initial states. This made the encoding of rejection even more pertinent: instead of a plain loop, the computation is started again from scratch, taking this time the actual initial pseudo-configuration.

What could we do?

This work is still very young and could be simplified in numerous ways.[28] We made some choices regarding the presentation, but the cursor between mathematics and computational complexity could be elsewhere. For instance, we could have defined the rejection of the PM as much closer to the

[28] A first attempt using "unification algebra" can be found in a joint work with Marc Bagnol [9].

way it is encoded in operators, that would have saved us the introduction of the "extended pseudo-configurations", which is not very handy. Yet, we made this choice for it could look very surprising to define PM this way without having some insight on the way we are going to simulate their computation.

Regarding the future of this framework, we can mention —among numerous other, like the development of a higher-order programming language— the following ideas:

1. We gave the definition of 1-way FA not only to expose our culture, but because it should be clear that they can be encoded in operators all the same, with even a simpler presentation, for we would only need to define ff and frec to encode the movement of a head. But if we recall Theorem 4.2.1, there are separations results both for non-deterministic and deterministic FA, 1-way and 2-ways. We know how to simulate every variation of FA: could we provide a new proof of those separation results? How could we reformulate the $\mathbf{L} = \mathbf{NL}$ conjecture? Is there any way to prove in a new setting $\mathbf{co\text{-}NL} = \mathbf{NL}$? In short, what is in our mathematical framework the meaning of those theorems and conjectures?

2. All the same, if we get rid of the tensor product and allow only one copy of $\mathrm{M}_6(\mathbb{C})$, we describe the computation of a (one-head) Finite Automaton. The languages recognized by Finite Automata (**2DFA**(1) in our notation) corresponds to regular languages, which admits many other definitions, among them "being expressed using a regular expression", or taking tools from the Chomsky hierarchy. But on the contrary, could a regular expression captures the computation of our observations?

3. What does the mathematical properties of operators teach us from a computational point of view? It is not very likely that $M_s^\bullet$ for M a 1-way FA has some mathematical properties that the encoding of a 2-way FA has not. All the same, there is a lot of isometries that go trough our object, identifying a lot of them. Does that always corresponds to "reasonable" transformations between PM, or is it possible to determine a discrepancy between what is acceptable from a mathematical point of view, and what is acceptable from a computational point of view?

4. We took the dimension of the algebra of the pseudo-states to be bounded, but nothing restrain us from taking an infinite-dimensional algebra of state. What could that mean? Well, one way to grasp it would be to allow an infinite number of states in a PM. To what complexity class could correspond such a setting? It is natural to look in the direction of non-uniforms complexity classes, and to ask ourselves what could be the difficulty to enumerate all the states of such a machine.

Conclusion and Perspectives

In some sense, this work lacks imagination: we spent a lot of time carefully checking that all the components of our constructions were sound and well defined. The understanding of those objects requires some work, and we have the feeling that the work of simplification we led for proof circuits could be pushed further, and applied to operators. We hope this work will help to get a better understanding of those wonderful objects.

Proof circuits do not seem to offer that much perspectives: it is a practical way to define properly the parallel computation of Proof Nets, as it is perfectly suited to mimic Boolean circuits. But it seems at the same time that this framework is too restrictive: there is no clear way to handle types other than boolean, and it is really pertinent to study them only for constant-depth resources. It could nevertheless be interesting to build proof circuits that actually solves some problems: maybe the type constraints could tell us something about the necessity of increasing the depth of the formulae?

On the other hand, the opportunities to push the study of operators forward seem endless: for instance, how could we build a concrete algorithm as operator? The framework may not look very handy, but in fact the properties of the projections does all the hard work. This could open new perspectives, by bounding more precisely the resources: Jones [84] showed using a pointer-like programming language that a constant factor was a significant in the case of *actual* computation, when we come back from theoretical models to computation with computers. This work was pushed further by Ben-Amram and Jones [17] and proposes another way of handling the complexity, by paying more attention to what is hidden inside this $\mathcal{O}$-notation. We believe our operators could offer a framework to handle such concerns.

The study of the logical part of this work could also be developed: what could be the proof of an observation? Could we adapt this construction to other logics? The tensor logic of Melliès and Tabareau [104] could allow us to bridge the gap with categorical semantics, in a framework where we already know precisely how to handle resources. This logic moreover allows to express sequentiality without being forced to renounce to proof nets, among other tools provided by Linear Logic. This could also offer a nice alternative to λ-calcul to study Boolean proof nets from a sequential perspective. One could also benefit from the work led by Seiller [121], to approach GoI with combinatorial tools.

There are other perspectives we could mention, but we prefer to conclude with an intriguing remark: we saw that graphs were heavily linked to log-space computation. On the other hand, matrix are usually related to Boolean circuits, for they are the ideal input to process in parallel (you may think of the multiplication of two matrices, a problem that is in $\mathbf{NC}^1$). And yet we modelised non-deterministic log-space computation with matrices, and parallel computation with graphs. Could we try to apply operators to proof circuits, and vice-versa?

APPENDIX A

Operator Algebras Preliminaries

Operator algebra can be seen as an "harmonious blend of algebra and analysis" (Curien [30]). It is also a complex subject, where spectral theory, functional analysis, linear algebra and topology meet. Each one of those perspectives came with its own notation, interpretation and nomenclature.[1]

So we felt it could be useful to synthesize in this first appendix the material needed to fully understand the construction of Chapter 3 and the encoding of Section 4.4. We tried to use a clear and neutral notation —summed up in the table of notations, p. xiii.

Operators are taken here to be continuous linear functions on a separable Hilbert space, and operators algebras are taken to be self-adjoint. It makes us focus on C^*-algebras and von Neumann algebras, and we will go straightforward to the definitions of those notions, in order to rapidly define projections, factors, hyperfinitness and type. Those notions are the one in use in our construction, and even if it is not mandatory to be an expert on this subject to understand it, we felt it necessary to ease their introduction by some reminders.

The first section defines von Neumann algebras following two paths: the first one will set up all the material needed to understand them as an algebraic object whereas the second offers a topological approach. The two definitions we propose —Definition A.1.18 and Definition A.1.22— are of course completely equivalent, thanks to the famous "double commutant theorem". We left aside the so-called "abstract" definition of von Neumann algebras, *a.k.a.* W^*-algebras, as C^*-algebras that are the dual space of a Banach algebra.

The second section exposes some basis regarding von Neumann algebras, explain their links with matrices, and develop the quite complex operation of crossed product. This last part involves massively groups and is useful to understand the normative pair we use in our construction (cf. Corollary 3.4.4).

Almost everything in this appendix is called a definition, but in fact it mixes definitions, lemmas and theorems. This appendix is in debt to some notes written by Curien [30] and by Girard [53], which prepared and eased the lecture of more complex handbooks. Thomas Seiller patiently explained me

[1] Even the definition of operators is subject to variations! A textbook on Operator Theory and Operator Algebra [109, p. 3] defines operators as "the bounded linear maps from a normed vector space to itself", a Linear Algebra handbook [12, p. 57] defines them as "linear map[s] from a vector space to itself" whereas a handbook of Theory of Operator Algebra [86, p. 2] takes them to be mappings between two vector spaces. Here, we will constraint operators to be bounded linear maps from a normed vector space to itself.

a lot regarding those notions, and the reader may find a quick overview of the needed material in the appendix of one of his article [120]. His thesis [121] presents extensively this subject and constitutes an excellent bridge toward those objects. Some motivation and a different way of exposing the subject can also be found in GIRARD [50, Appendix on C*-algebra, pp. 48–60]. All those references could help the reader more used to Linear Logic to adapt to operator algebra.

The reader who would like to go further could refer to the textbooks of CONWAY [26] for the bases of the theory (starting from Hilbert and Banach spaces, C*-algebra, topologies on them), of MURPHY [109] for an excellent introduction to the theory of operator algebras. The series of TAKESAKI [126, 127, 128] remains the most complete reference.

A.1 Towards von Neumann Algebras

There is not much to say about prerequisites needed to read this section, for we tried to define everything in a simple manner. Just agree that we use δ for the *Kronecker delta*, defined by

$$\delta_{ij} = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases}$$

The algebraic way

The reader used to vector spaces, operators and C*-algebra, may directly go to Definition A.1.18 of von Neumann algebras. Elsewhere, (s)he can have a look at Figure A.1 to see the summary of the algebraic structures we need to properly defines von Neumann algebras.

Definition A.1.1 (Distance, Metric space)**.** A *metric space* is a pair (S,d) with $S \neq \emptyset$ a set and $d : S^2 \mapsto \mathbb{R}$ a *distance over* S such that $\forall x,y,z \in S$

$$\begin{aligned} d(x,y) &= d(y,x) && \text{(Symmetry)} \\ d(x,y) &= 0 \text{ iff } x = y && \text{(Separation)} \\ d(x,z) &\leqslant d(x,y) + d(y,z) && \text{(Subadditivity)} \end{aligned}$$

Definition A.1.2 (Algebraic Structures)**.** Let E be a set, $+$ and $\times$ two binary operations, we define the following set of rules:

$$\begin{aligned} \forall a,b \in E,\ & (a+b) \in E && \text{(Closure)} \\ \forall a,b,c \in E,\ & (a+b)+c = a+(b+c) && \text{(Associativity)} \\ \exists e_+ \in E, \forall a \in E,\ & e_+ + a = a + e_+ = a && \text{(Identity element)} \\ \forall a \in E, \exists -a \in E,\ & a+(-a) = (-a)+a = e_+ && \text{(Inverse element)} \\ \forall a,b, \in E,\ & a+b = b+a && \text{(Commutativity)} \\ \forall a_1,a_2,a_3 \in E,\ & (a_1+a_2)\times a_3 = (a_1 \times a_3)+(a_2 \times a_3) && (\times \text{ is right-distributive over } +) \\ \forall a_1,a_2,a_3 \in E,\ & a_1 \times (a_2+a_3) = (a_1 \times a_2)+(a_1 \times a_3) && (\times \text{ is left-distributive over } +) \end{aligned}$$

The rules are instantiated to $+$, but a simple substitution makes clear what are the rules for $\times$.

This set of rules allow us to define several key-concepts at the same time:

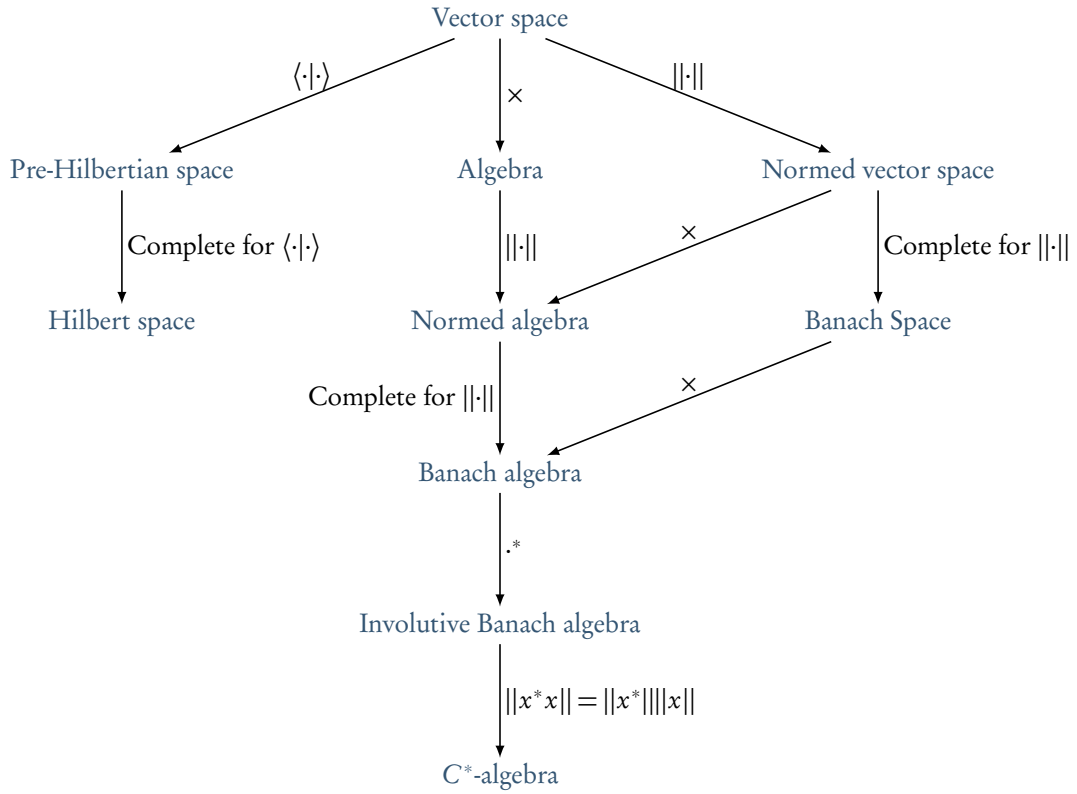


Figure A.1: Dependency of the algebraic structures

- If $\mathcal{G} = (E, +)$ respects eqs. (Closure) to (Inverse element), then $\mathcal{G}$ is a *group*. If $\mathcal{G}$ moreover respects eq. (Commutativity), then $\mathcal{G}$ is said to be *commutative* (or *abelian*).

- If $R = (E, +, \times)$ is such that $(E, +)$ is a commutative group, $\times$ respects eq. (Closure), eq. (Identity element) and is distributive over $+$,[2] then R is a *pseudo-ring*. If moreover $\times$ is associative, R is a *ring* and if moreover $\times$ is commutative then R is a *commutative ring*.

- If $\mathcal{F} = (E, +, \times)$ is such that $(E, +)$ and $(E \backslash \{e_+\}, \times)$ are two commutative groups and $\times$ distributes over $+$, then $\mathcal{F}$ is a *field* (also called a *corpus*).

In all of those case, we call E the *underlying set*. For the sake of simplicity, we will denote the underlying set of a group $\mathcal{G}$(resp. a field $\mathcal{F}$) with $\underline{\mathcal{G}}$(resp. $\underline{\mathcal{F}}$). We will also use later on this notation for vector spaces.

The operations $+$ and $\times$ are mostly refereed to as *addition* and *multiplication*, and the neutral elements e_+ and $e_\times$ are most of the time denoted 0 and 1. The inverse of $a \in E$ under $+$ is usually written $-a$ whereas the notation a^{-1} is kept for the inverse under $\times$.

Of particular interest to us will be the group of permutations over a set.

[2] Meaning that $\times$ is left-distributive over $+$ and $\times$ is right-distributive over $+$.

Definition A.1.3 (Permutation of a set, Symmetric group)**.** Let S be a set, a *permutation of* S is a bijection from S to itself. We set $\mathscr{S}(S)$ to be the group of all permutations of S, we write $\mathscr{S}_n$ if $S=\{1,\dots,n\}$ and $\mathscr{S}$ if $S=\mathbb{N}$.

For $<$ a total ordering over S and for all $\sigma\in\mathscr{S}(S)$, we define its *signature* $\mathrm{sgn}(\sigma)$ to be the parity of the number of inversions for σ

$$\mathrm{sgn}(\sigma)\coloneqq \mathrm{Card}(\{(x,y)\mid x<y \text{ and } \sigma(x)>\sigma(y)\}) \pmod 2$$

Definition A.1.4 (Subgroup)**.** Let $\mathscr{G}=(E,+)$ be a group, $\mathscr{H}=(F,+_{\restriction_F})$ is a *subgroup of* $\mathscr{G}$ if $F\subseteq E$ and $+_{\restriction_F}$ is a group operation on F. We write $\mathscr{H}\leq\mathscr{G}$. If $F\subsetneq E$, $\mathscr{H}$ is said to be a *proper subgroup of* $\mathscr{G}$.

If $\forall h\in F,\forall x\in E, x+h+x^{-1}\in F$, $\mathscr{H}$ is a *normal subgroup of* $\mathscr{G}$ and we write $\mathscr{H}\triangleleft\mathscr{G}$.

We introduce in the following two operations, the vector addition and the vector multiplication, and denote also them with $+$ and $\times$. This is quite classical and harmless, since the type and the context should always make clear which one is in use. Moreover, the algebras axioms tells us basically that those operations are the same over different sets.

Definition A.1.5 (Vector space, Algebra over a field)**.** A *vector space* V over a field $\mathscr{F}=(\underline{\mathscr{F}},+,\times)$ is a set $\underline{\mathsf{V}}$ endowed with two binary operations, $+:\underline{\mathsf{V}}^2\mapsto\underline{\mathsf{V}}$ the vector addition and $\cdot:\underline{\mathscr{F}}\times\underline{\mathsf{V}}\mapsto\underline{\mathsf{V}}$ the scalar multiplication, such that $(\underline{\mathsf{V}},+)$ is a commutative group and such that for all *vectors* $u,v,w\in\underline{\mathsf{V}}$, for all *scalars* $\lambda,\mu\in\underline{\mathscr{F}}$, the following holds:

$$\begin{aligned}
\lambda\cdot(u+v)&=(\lambda\cdot u)+(\lambda.v) && (\cdot \text{ is left-distributive over } +)\\
(\lambda+\mu)\cdot u&=(\lambda\cdot u)+(\mu\cdot u) && (\cdot \text{ is right-distributive over } +)\\
(\lambda\times\mu)\cdot u&=\lambda\cdot(\mu\cdot u) && (\text{Distributivity of } \times \text{ over } \cdot)\\
e_\times\cdot u&=u && (\text{Identity element of } \times)
\end{aligned}$$

We say that W is a *linear subspace of* V *over* $\mathscr{F}$ if $\underline{\mathsf{W}}\subseteq\underline{\mathsf{V}}$, $\underline{\mathsf{W}}\neq\emptyset$ and $\forall x,y\in\underline{\mathsf{W}}$, $\forall\lambda\in\underline{\mathscr{F}}$, $x+y\in\underline{\mathsf{W}}$ and $\lambda x\in\underline{\mathsf{W}}$.

If moreover V is endowed with a binary operation $\times$ such that

$$\begin{aligned}
(u+v)\times w&=(u\times v)+(u\times w) && (\times \text{ is right-distributive over } +)\\
u\times(v+w)&=(u\times v)+(v\times w) && (\times \text{ is right-distributive over } +)\\
(\lambda\cdot u)\times(\mu\cdot v)&=(\lambda\times\mu)\cdot(u\times v) && (\text{Compatibility with scalars})
\end{aligned}$$

Then $\mathcal{A}=(\underline{\mathsf{V}},+,\cdot,\times)$ is an *algebra over the field* $\mathscr{F}$. If moreover $\times$ is associative, i.e. if $u\times(v\times w)=(u\times v)\times w$, then $\mathcal{A}$ is said to be an associative algebra over $\mathscr{F}$.

We say that $\mathcal{U}$ is a *subalgebra of* $\mathcal{A}$ *over* $\mathscr{F}$ if $\underline{\mathcal{U}}\subseteq\underline{\mathcal{A}}$, $\mathcal{U}$ is a linear subspace of $\mathcal{A}$ over $\mathscr{F}$ and $\forall u,v\in\underline{\mathcal{U}}$, $u\times v\in\underline{\mathcal{U}}$.

Definition A.1.6 (Basis of a vector space)**.** A basis $\mathbb{B}$ of a $\mathscr{F}$-vector space V is a linearly independent subset of $\underline{\mathsf{V}}$ that spans V.

$\mathbb{B}\subseteq\underline{\mathsf{V}}$ is *linearly independent* if for every $\mathbb{B}_0=\{f_1,\dots,f_n\}\subset\mathbb{B}$, $\forall(\lambda_1,\dots,\lambda_n)\in\underline{\mathscr{F}}^n$, $(\lambda_1.f_1)+\dots+(\lambda_n.f_n)=e_+$ implies $\lambda_1=\dots=\lambda_n=e_+$.

$\mathbb{B} = (f_i)_{i\in I}$ *spans* V if $\forall u \in \underline{\mathsf{V}}$ there exists $(\lambda_i)_{i\in I} \in \underline{\mathscr{F}^I}$ with $\{i \in I \mid \lambda_i \neq 0\}$ finite such that $u = \sum_{i\in I} \lambda_i f_i$.

All bases of a vector space V have the same cardinality, called the *dimension* of V and denoted by $\dim(\mathsf{V})$.

Definition A.1.7 (Inner product, orthogonality, pre-Hilbertian space)**.** Let V be a $\mathbb{C}$-vector space, an *inner product* is a map $\langle\cdot|\cdot\rangle : \underline{\mathsf{V}}^2 \mapsto \mathbb{C}$, such that $\forall x, y, z \in \underline{\mathsf{V}}$, $\lambda \in \mathbb{C}$

$$\begin{aligned}
\langle x|y\rangle &= \overline{\langle y|x\rangle} && \text{(Conjugate symmetry)}\\
\langle \lambda x|y\rangle &= \lambda\langle x|y\rangle && \text{(Linearity in the first argument)}\\
\langle x+y|z\rangle &= \langle x|z\rangle + \langle y|z\rangle &&\\
\langle x|x\rangle \geq 0 \text{ and } \langle x|x\rangle = 0 &\text{ implies } x = 0 && \text{(Positive-definiteness)}
\end{aligned}$$

Where $\overline{(\cdot)}$ is the conjugate defined by $\overline{a + i\,b} = a - i\,b$. We call this product *sesquilinear*.[3]

If $\langle x|y\rangle = 0$, we say that x and y are orthogonal and we define $\mathsf{W}^\perp$ as the *orthogonal complement* of a subspace W of V by $\mathsf{W}^\perp = \{x \in \mathsf{V} \mid \forall y \in \mathsf{W}, \langle x|y\rangle = 0\}$.

A *pre-hilbertian space* $(E, \langle\cdot|\cdot\rangle)$ is a vector space E with an inner product $\langle\cdot|\cdot\rangle$.

Definition A.1.8 (Norm, normed vector space)**.** Let V be a $\mathscr{F}$-vector space, a *norm over* V is a map $||\cdot|| : \underline{\mathsf{V}} \mapsto \mathbb{R}$ such that $\forall u, v \in \underline{\mathsf{V}}$, $\forall \lambda \in \underline{\mathscr{F}}$, the following holds:

$$\begin{aligned}
||u|| = 0_{\mathbb{R}} &\text{ iff } u = e_+ && \text{(Separation)}\\
||u+v|| &\leqslant ||u|| + ||v|| && \text{(Subadditivity)}\\
||\lambda.u|| &= |\lambda|.||u|| && \text{(Positive homogeneity)}
\end{aligned}$$

A *normed vector space* is a pair $(\mathsf{V}, ||\cdot||)$ where V is a vector space and $||\cdot||$ is a norm on V. The norm induces a distance by letting $d(x, y) = ||x - y||$.

Definition A.1.9 (Convergent sequence, Cauchy sequence)**.** A sequence $x_1, x_2, \ldots$ of a metric space (S, d)*converge (in S)* if $\exists l \in S$—called *the limit*— such that $\forall \epsilon \in \mathbb{R}^{>0}$, $\exists N \in \mathbb{N}$, $\forall n \in \mathbb{N}$,

$$n \geq N \Rightarrow d(x_n - l) < \epsilon$$

We write $x_1, x_2, \ldots \to l$. We say that $x_1, x_2, \ldots$ is a *Cauchy sequence* if $\forall \epsilon \in \mathbb{R}^{>0}$, $\exists n \in \mathbb{N}$, $\forall p > n, \forall q \in \mathbb{N}$,

$$d(x_{p+q} - x_p) < \epsilon$$

Every convergent sequence is a Cauchy sequence, and (S, d) is called *complete* (or a *Cauchy space*) if every Cauchy sequence in S converges.

This definition can easily be adapted to normed vector space: we only have to replace the distance by the norm.

Definition A.1.10 (Pointwise convergent)**.** Let $\{f_n\}$ be a sequence of functions with the same domain and co-domain, $\{f_n\}$ *converges pointwise to* f iff for all x in the domain, $\lim_{n\to\infty} f_n(x) = f(x)$.

[3] We take the mathematical convention, our inner product is linear in the first argument and anti-linear in its second. The physicists use the opposite convention.

Definition A.1.11 (Banach space)**.** A *Banach space* is a normed vector space which is complete for the induced distance.

Definition A.1.12 (Normed algebra, Banach algebra)**.** A *normed algebra* $(S,+,\cdot,\times,||\cdot||)$ is

$(S,+,\cdot,\times)$ an associative algebra over $\mathbb{R}$ or $\mathbb{C}$,

$(S,||\cdot||)$ a normed vector space

such that the algebra multiplication and the norm respects: $\forall u,v\in S, ||u\times v||\leqslant||u||\times||v||$.

In particular, if $(S,||\cdot||)$ is a Banach space, $(S,+,\cdot,\times,||\cdot||)$ is a *Banach algebra*. If $\times$ admit an identity element $e_\times$, and $||e_\times||=1_\mathbb{R}$, $(S,+,\cdot,\times,||\cdot||)$ is *unital*.

Definition A.1.13 (Involutive Banach algebra, C*-algebra)**.** If a Banach algebra $\mathcal{B}=(S,+,\cdot,\times,||\cdot||)$ admits a map $(\cdot)^*:S\to S$ such that $\forall u,v\in S,\lambda\in\mathbb{C}$:

$$(\lambda\cdot u)^*=\overline{\lambda}\cdot u^* \qquad (u+v)^*=u^*+v^* \qquad (u\times v)^*=v^*\times u^* \qquad u^{**}=u \qquad ||u^*||=||u||$$

then V is called an *involutive Banach algebra* and the map $(\cdot)^*$ is called the *involution*. If the involution satisfies the following additional condition:

$$||u^*u||=||u^*||||u||=||u||^2$$

then $\mathcal{B}$ is called a C*-*algebra* (or *stellar algebra*).

A $*$-*homomorphism* between two C*-algebras $\mathcal{B}$ and $\mathcal{C}$ is a bounded linear map $f:\underline{\mathcal{B}}\to\underline{\mathcal{C}}$ that "preserve $(\cdot)^*$", i.e. $\forall x,y\in\underline{\mathcal{B}}$, $f(x^*)=f(x)^*$ and $f(xy)=f(x)f(y)$.

If f is injective, it is an isometry, and if f is bijective, it is a C*-isomorphism.

Definition A.1.14 (Hilbert space)**.** A *Hilbert space* $\mathbb{H}$ is a $\mathbb{C}$-vector space with a norm $\langle\cdot|\cdot\rangle$ such that $\mathbb{H}$ is a complete metric space with respect to the distance d defined by

$$d(x,y)=||x-y||=\sqrt{\langle x-y|x-y\rangle}$$

It is a Banach space where $\forall x\in\mathbb{H}$, $\langle x|x\rangle=||x||^2$.

We always take our Hilbert spaces to be infinite-dimensional and separable. That induces the existence of an orthonormal separable basis, but the usual concept of basis (defined in Definition A.1.6) cannot be applied "as it" to infinite Hilbert spaces. In this setting, we use *Hilbert basis* that allow us to approximate every element with an infinite sequence of finite elements.

Definition A.1.15 (Hilbert basis)**.** A family $\mathbb{F}=(x_i)_{i\in I}\subseteq\mathbb{H}$ is a *Hilbert basis* of $\mathbb{H}$ if

$$\forall i,j\in I^2,\ \langle x_i|x_j\rangle=\delta_{ij} \qquad (\mathbb{F}\text{ is orthonormal})$$

$$\forall x\in\mathbb{H},\exists\{\lambda_i\}_{i\in I}\in\mathbb{C}^I,\ x=\sum_{i\in I}\lambda_i x_i \qquad (\mathbb{F}\text{ is complete})$$

We will speak of "homomorphism" between two algebraic structure as long as this function preserve the algebraic structure under treatment. For instance, given $(\mathcal{G},+)$ and $(\mathcal{G}',+')$ two groups, $f:\mathcal{G}\to\mathcal{G}'$ is an homomorphism if $\forall g_1,g_2\in\mathcal{G}$, $f(g_1+g_2)=f(g_1)+'f(g_2)$.

Among the homomorphisms, of particular interest to us are the homomorphisms between vector spaces, called *linear maps*.

Definition A.1.16 (Linear map)**.** Let V and W be two $\mathscr{F}$-vector spaces, f a map from V to W is said to be a *linear map* (or is $\mathscr{F}$*-linear*) iff

$$\forall x, y \in \underline{\mathscr{F}}, \forall \mu \in \underline{\mathsf{V}}, f(x + (\mu \cdot y)) = f(x) + (\mu \cdot f(y))$$

Let $\mathcal{L}(V, W)$ be the set of linear applications from V to W, we define *the dual space* V^* to be $\mathcal{L}(\mathsf{V}, \mathscr{F})$.

If $\mathsf{V} = \mathsf{W}$, f is said to be an *endomorphism* and we write $\mathcal{L}(\mathsf{V})$ the set of linear maps from V to V. For all V a $\mathscr{F}$-vector spaces, $\mathcal{L}(\mathsf{V})$ endowed with the pointwise addition and the composition of functions as a multiplication operation is an algebra over the field $\mathscr{F}$.

In fact, we instantiate the previous definition to the special case of the Hilbert space, and restrict ourselves to the case we will use to define *operators* as the bounded mappings between two Hilbert spaces.

Definition A.1.17 (Operator, $\mathcal{B}(\mathbb{H})$)**.** A linear map u from $\mathbb{H}$ to itself is *bounded* if there exists $c \in \mathbb{N}$ such that $\forall x \in \mathbb{H}, ||u(x)||_{\mathbb{H}} \leqslant c||x||_{\mathbb{H}}$. We define $\mathcal{B}(\mathbb{H})$ to be the set of bounded linear maps from $\mathbb{H}$ to itself, and we define a norm on $\mathcal{B}(\mathbb{H})$:

$$||u||_{\mathcal{B}(\mathbb{H})} \coloneqq \sup_{x \in \mathbb{H}} \left(\frac{||ux||_{\mathbb{H}}}{||x||_{\mathbb{H}}} \right)$$

For all $u \in \mathcal{B}(\mathbb{H})$, there exists a unique $u^* \in \mathcal{B}(\mathbb{H})$, the *adjoint of* u, such that $(\cdot)^*$ is an involution satisfying $||u^* u|| = ||u||^2$.

If we take the addition to be the addition of vectorial spaces, the multiplication to be the composition and $+$ to be the multiplication by a scalar, then $(\mathcal{B}(\mathbb{H}), +, \times, +, ||\cdot||, (\cdot)^*)$ is a C*-algebra.

It is equivalent for an operator to be bounded or to be continuous, see Definition A.1.20.

Definition A.1.18 (Commutant, von Neumann algebra)**.** Let $\mathfrak{M} \subset \mathcal{B}(\mathbb{H})$ be a C*-subalgebra, we define the *commutant* of $\mathfrak{M}$(relatively to $\mathcal{B}(\mathbb{H})$, but we will omit this precision) to be the set

$$\mathfrak{M}' \coloneqq \{u \in \mathcal{B}(\mathbb{H}) \mid \forall m \in \mathfrak{M}, mu = um\}$$

If $\mathfrak{M} = \mathfrak{M}''$, $\mathfrak{M}$ is a von Neumann algebra.

The topological way

This path will be way shorter, mainly because we only need weak operator topology in the following. Some of the notions previously defined (as norm, Hilbert space, or convergence) are required for this section. We will spend some time with a classical example midway between topology and algebra.

Definition A.1.19 (Topology, Topological space)**.** Let X be a set and $\mathcal{T} \subseteq \mathcal{P}_{\text{fin}}(X)$ a family of subset of X. We say that $\mathcal{T}$ is a *topology* on X if:

$$X \in \mathcal{T}, \emptyset \in \mathcal{T} \qquad \forall x_1, x_2 \in \mathcal{T}, x_1 \cap x_2 \in \mathcal{T} \qquad \forall I, \forall (x_i)_{i \in I} \in \mathcal{T}^I, \cup_{i \in I} x_i \in \mathcal{T}$$

If $\mathcal{T}$ is a topology on X then the pair $(X, \mathcal{T})$ is a *topological space*. The elements of $\mathcal{T}$ are called *open sets* in X. The subset $Y \subseteq X$ is said to be *closed* if its complement is in $\mathcal{T}$.

For all $x \in X$, *the neighbourhood of* x is a set $N \subseteq X$ such that there exist an open set O with $x \in O \subseteq N$. We write $N(x)$ the set of the neighbourhoods of x. We call X*compact* if for all $\{U_\alpha\}_{\alpha \in A}$ of open subsets of X such that $X = \bigcup_{\alpha \in A} U_\alpha$, there is a finite $J \subseteq A$ such that $X = \bigcup_{i \in J} U_i$. We say that X is *locally compact*, if $\forall x \in X$, $N(x)$ is compact.

Definition A.1.20 (Continuous map)**.** Let $(X,\mathcal{T})$ and $(X',\mathcal{T}')$ be two topological spaces, a map $f : X \to X'$ is *continuous* if and only if for every $x \in X$ and every $V' \in N(f(x))$ there exists $V \in N(x))$ such that $y \in V$ implies $f(y) \in V'$.

Said differently, f is continuous if for every open set $O \subseteq X'$, $f^{-1}(V) = \{x \in X \mid f(x) \in V\}$ is an open subset of X.

Definition A.1.21 (Weak topology on $\mathbb{H}$ and $\mathcal{B}(\mathbb{H})$)**.** Let $\mathbb{H}$ be a Hilbert space, we say that a sequence $\{x_i\}_{i\in\mathbb{N}}$ converges weakly to 0 when $\langle x_i|y\rangle \to 0$ for all $y \in \mathbb{H}$. Weak convergence is thus a point-wise or direction-wise convergence.

On $\mathcal{B}(\mathbb{H})$, we says that $\{u_i\}_{i\in\mathbb{N}}$ converges weakly to 0 when, for any $x \in \mathbb{H}$, $u_i x$ converges weakly to 0. This defines the *weak operator topology*.

We can show that $\mathcal{B}(\mathbb{H})$ is the dual of a space denoted by $\mathcal{B}(\mathbb{H})_*$ containing the *trace-class operators*. For further details, the reader may refer to Murphy [109, section 2.4, pp. 53 sqq.] or Takesaki [126, p. 63]. We recall this result only to define the σ-weak topology: if A is a topological space and A^* is its dual, the *weak* topology* on A^* is defined as the point-wise topology.

Definition A.1.22 (von Neumann algebra (2))**.** A von Neumann algebra $\mathfrak{M}$ is a C*-subaglebra of $\mathcal{B}(\mathbb{H})$ which is closed in the weak operator topology and contains the identity operator.

We are beginning to mix topology and linear algebra with the following example of one of the most famous Hilbert space. The following construction could be generalized with the Lesbegue spaces L^2 over measurable spaces, i.e. the set of measurable functions whose power of 2 has finite integral.

Example A.1.1 ($\ell^2(S)$)**.** Let S be a set, we define $\ell^2(S)$ to be the set of square-summable functions

$$\ell^2(S) = \{f : S \mapsto \mathbb{C} \mid \sum_{s\in S} |f(s)|^2 < \infty\}$$

where $|\cdot|$ is the absolute value of the complex number, i.e. if $f(s) = a + ib$, $|f(s)| = \sqrt{a^2+b^2}$.

This set $\ell^2(S)$ is a $\mathbb{C}$-vectorial space if we take for the addition the pointwise addition of the functions, for the multiplication the multiplication of $\mathbb{C}$. It is a Hilbert space if we define an inner product $\forall f, g \in \ell^2(S)$

$$\langle f|g\rangle = \sum_{s\in S} f(s)\overline{g}(s)$$

where $\overline{g}(s) = \overline{g(s)}$ is the conjugate of $g(s)$. The norm is as usual defined as $||f|| = \sqrt{\langle f|f\rangle}$ and we can easily check that $\ell^2(S)$ is complete with respect to that norm.

This example is interesting because every Hilbert space with an Hilbert basis $\mathbb{F}$ is isomorphic to $\ell^2(\mathbb{F})$. In particular, if $\mathbb{F}$ is a discrete group $\mathcal{G}$, $\ell^2(\mathcal{G})$ is a von Neumann algebra.

And if $\mathcal{G}$ is the set bijections from $\mathbb{N}$ to $\mathbb{N}$ that permute a finite number of elements, the von Neumann algebra obtained is a II_1-factor, whose definition follows.

A.2 Some Elements on von Neumann Algebras

In this section we go one step further by introducing some more complex notions. We recall some of the terminology of the von Neumann algebras, explains the links between operator algebra and matrices algebra, and then study the crossed-product of a von Neumann algebra with a group.

Basics of von Neumann algebras

First, we should remark that von Neumann algebras also admit a *tensor product*, which comes basically from the tensor product for Hilbert spaces.

Definition A.2.1 (Tensor product for the von Neumann algebras)**.** If for $i = 1, 2$, $\mathfrak{M}_i$ is a von Neumann algebra on the Hilbert space $\mathbb{H}_i$, then $\mathfrak{M}_1 \otimes \mathfrak{M}_2$*is* the von Neumann algebra on $\mathbb{H}_1 \otimes \mathbb{H}_2$.

Much of the terminology of C*-algebras is the same as for the operator theory on Hilbert space.

Remark (Operators zoo). Let $u \in \mathcal{B}(\mathbb{H})$, we say that u is

Normal if $uu^* = u^*u$

Self-adjoint or Hermitian if $u^* = u$

A projection if $u = u^* = u^2$

A partial isometry if $uu^* = p$ for p a projection

If $\mathcal{B}(\mathbb{H})$ is *unital* (it admits an identity operator $\mathrm{Id}_{\mathcal{B}(\mathbb{H})}$), we say that u is

Unitary if $uu^* = u^*u = \mathrm{Id}_{\mathcal{B}(\mathbb{H})}$

An isometry if $uu^* = \mathrm{Id}_{\mathcal{B}(\mathbb{H})}$

Definition A.2.2 (Center of a von Neumann algebra, factor)**.** The *center* of a von Neumann algebra $\mathfrak{M}$ is defined as $\{x \in \mathfrak{M} \mid \forall y \in \mathfrak{M}, xy = yx\}$. This is the same as the set $\mathfrak{M} \cap \mathfrak{M}'$ for $\mathfrak{M}'$ the commutant of $\mathfrak{M}$.

A von Neumann algebra $\mathfrak{N}$ is a *factor* if its center is trivial, i.e. if its consists only of multiples (scalar operators) of the identity operator.

Every von Neumann algebra on a separable Hilbert space is isomorphic to a direct integral (a continuous direct sum) of factors, and this decomposition is essentially unique.

So the study of von Neumann algebras is partially subsumed by the study of classes of factors. The factors themselves can be classified by their sets of projections, whose definition follows. Informally, a projection sends the Hilbert space $\mathbb{H}$ onto a closed subspace of $\mathbb{H}$. Depending on the dimensions and the properties of this closed subspace, we may define an equivalence relation on them.

Definition A.2.3 (Finite and infinite projections)**.** A projection is an operators $p \in \mathcal{B}(\mathbb{H})$ such that $p = p^* = p^2$. We write $\Pi(\mathfrak{M})$ the set of projections in $\mathfrak{M}$. Let p, q be two projections of a von Neumann algebra, if $qp = pq = p$ or equivalently if $\dim(p) \subseteq \dim(q)$, then we write $p \leqslant q$.

In a von Neumann algebra, we can define an equivalence relation on $\Pi(\mathfrak{M})$ by $p \sim_{\mathfrak{M}} q$ if there exists a partial isometry $u \in \mathfrak{M}$ such that $uu^* = p$ and $u^*u = q$. This equivalence relation is sometimes called the *Murray and von Neumann equivalence*.

If there exists q such that $q < p$ and $q \sim p$, p is said to be *infinite*. Conversely, a projection p is *finite* if $\forall q$ a projection, $q \sim p$ and $q < p$ implies $q = p$. A projection p is said to be *minimal* if $q \leqslant p$ implies $q = 0$ or $q = p$.

A von Neumann algebra can be "recreated" from its projections, it is sufficient to know $\Pi(\mathfrak{M})$ to describe totally $\mathfrak{M}$. In fact, there is a bijection between the projections of $\mathfrak{M}$ and the subspaces of $\mathbb{H}$ that belong to it. Projections also allow to classify them, whenever they have minimal projections or not, finite projections or not. We will only use the II_1 factor, so we define only this one, and we define at the same time hyperfinitness.

Definition A.2.4 (Hyperfinite type II_1 factor)**.** A von Neumann algebra $\mathfrak{N}$ where $\leqslant$ is total is *a factor*, and it is said to be of type II if $\mathfrak{N}$ contains finite projections but has no minimal projections. If the identity of $\mathfrak{N}$ is a finite projection, $\mathfrak{N}$ is of type II_1.

A von Neumann algebra $\mathfrak{M}$ is *hyperfinite* if there exists an ascending sequence $(\mathfrak{M}_i)_{i\in\mathbb{N}}$ of finite-dimensional subalgebras of $\mathfrak{M}$ such that their union $\cup\mathfrak{M}_i$ is dense in $\mathfrak{M}$.[4] Equivalently,

$$\dim(\mathfrak{M}_i) < \infty \text{ for all } i \qquad \mathfrak{M}_1 \subset \mathfrak{M}_2 \subset \mathfrak{M}_3 \dots \qquad \mathfrak{M} = \cup_{i\in\mathbb{N}}\mathfrak{M}_i$$

The type II_1 hyperfinite factor is unique up to isomorphism and we will write it $\mathfrak{R}$.

The hyperfinitness of a factor $\mathfrak{M}$ implies that the operators of $\mathfrak{M}$ can be approximated[4] by matrices, i.e. by operators acting on a finite-dimensional Hilbert space.

On matrices

Matrices shares a lot with hyperfinite von Neumann algebras, partly thanks to the possibility they have to represent linear functions. We will also see how to represent matrices algebra with operator algebra. But first, let's agree on some notations.

Definition A.2.5 (Matrices)**.** Let $\mathrm{M}_{n,m}(\mathscr{F})$ be the set of matrices with coefficients in $\mathscr{F}$, with n rows and m columns. If $n = m$, the matrices are said to be *square* and we write $\mathrm{M}_n(\mathscr{F})$ the set of square matrices. Most of the time we won't specify the field where the matrix takes it coefficients, and we define as usual:

- I_n to be the *identity matrix* of size n, 0_n to be the *null matrix*.
- $\mathrm{M}[i,j]$ to be the the entry in the ith row and jth column of M.
- If $\forall i,j \leqslant n, \mathrm{M}[i,j] = 0$ if $i \neq j$, M_n is a *diagonal matrix*.
- If $\forall i,j \leqslant n, \mathrm{M}[i,j] = \mathrm{M}[j,i]$, M_n is a *symmetric*.
- The *trace* of a matrix M_n is defined by $\mathrm{tr}(\mathrm{M}_n) = \sum_{i=1}^{n} \mathrm{M}[i,i]$.
- A matrix M_n is a *projection* if $\mathrm{M}_n^2 = \mathrm{M}_n$.
- The *determinant* of a matrix M_n is defined as usual by the Leibniz formula:

$$\det(\mathrm{M}_n) = \sum_{\sigma\in\mathscr{S}_n} \mathrm{sgn}(\sigma) \prod_{i=1}^{n} \mathrm{m}_{i,\sigma(i)}$$

- If $\mathrm{M}_1[i,j] = \mathrm{M}_2[j,i]$, we say that M_2 is the *transpose* of M_1 and we write $\mathrm{M}_1{}^{\mathrm{t}} = \mathrm{M}_2$.

[4] For the σ-weak topology that we did not defined.

- If $\overline{\mathrm{M}[i,j]} = \mathrm{M}[i,j]$, we say that $\overline{\mathrm{M}}$ is the *conjugate* of the matrix M.
- If $\mathrm{M}^* = \overline{\mathrm{M}^{\mathrm{t}}}$, we say that M^* is the *conjugate-transpose* of M.
- If M_n is such that $\exists d \in \mathbb{N}^*$ and ${\mathrm{M}_{\mathrm{n}}}^d = 0_{\mathrm{n}}$, M_n is said to be *nilpotent* and the smallest such d is the *degree (of nilpotency) of* M_n.

We should remark that $\forall n \in \mathbb{N}$, the set of matrices $\mathrm{M}_n(\mathbb{C})$ is a non-commutative algebra over $\mathbb{C}$. We simply takes the classical matrix addition and multiplication, the scalar multiplication, and $\mathrm{M}_n(\mathbb{C})$ inherits all the properties of $\mathbb{C}$ as an algebra.

Definition A.2.6 (Matrix representation of linear maps)**.** Let V be a $\mathscr{F}$-vector space of dimension n and let $\mathbb{B} = \{b_1, \ldots, b_n\}$ be a basis of V and $u \in \mathcal{L}(\mathsf{V})$.

$$\forall 1 \leqslant k \leqslant n, \exists \lambda_1^k, \ldots, \lambda_n^k \in \mathscr{F} \text{ such that } u(b_k) = \lambda_1^k(b_1) + \ldots + \lambda_n^k(b_n)$$

And we define

$$[u]_{\mathbb{B}} \coloneqq \begin{pmatrix} \lambda_1^1 & \ldots & \lambda_1^n \\ \vdots & \ddots & \vdots \\ \lambda_n^1 & \ldots & \lambda_n^n \end{pmatrix}$$

Remark that

- u is nilpotent ($u \circ \ldots \circ u = 0$) iff $[u]_{\mathbb{B}}$ is nilpotent.
- If u is nilpotent, $\det([u]_{\mathbb{B}}) = \mathrm{tr}([u]_{\mathbb{B}}) = 0$.
- For all $\mathbb{B}_1$ and $\mathbb{B}_2$ two basis of V, $\mathrm{tr}([u]_{\mathbb{B}_1}) = \mathrm{tr}([u]_{\mathbb{B}_2})$. So we can speak of *the* trace of a linear map.

In the case of the II_1 hyperfinite factor, any element can be approximated by finite matrices, and we can always define a "normalised trace" by $\mathrm{Tr}(\mathrm{M}_n) = \frac{\mathrm{tr}(\mathrm{M}_n)}{n}$. This trace allows us to define a generalization of the determinant, the *Fuglede-Kadison determinant*.

But for now we consider the reverse operation, i.e. representing any matrix algebra in the II_1 hyperfinite factor.

Definition A.2.7 (Embedding matrices algebra in the II_1 hyperfinite factor)**.** Let $\mathfrak{R}$ be the II_1 hyperfinite factor and $\Pi(\mathfrak{R})$ be the set of projections of $\mathfrak{R}$. As $\mathfrak{R}$ is of type II, it has no minimal projection, so we can always decompose a projection, i.e. for all $p \in \Pi(\mathfrak{R})$, there exists $q_1, q_2 \in \Pi(\mathfrak{R})$ such that $q_1 q_2 = 0$ and $q_1 + q_2 = p$. Moreover, we can chose q_1 and q_2 such that $q_1 \sim q_2$ i.e. such that they act on subspaces with the same dimensions.

If we generalize, $\forall p \in \Pi(\mathfrak{R})$, for all $n \in \mathbb{N}^*$, there exists $q_1, \ldots q_n \in \Pi(\mathfrak{R})$ such that the following holds:

$$q_i \sim q_j \text{ for all } 1 \leqslant i \leqslant j \leqslant n \qquad \sum_{i=1}^{n} q_i = p \qquad q_i q_j = \delta_{ij} q_i$$

Now we take $p_1, \ldots, p_n \in \Pi(\mathfrak{R})$ such that $\sum_{i=1}^n p_i = 1$. As $p_i \sim p_j$, we know that there exists $u_{i,j}$ a partial isometry such that

$$u_{i,j} u_{i,j}^* = p_j \qquad u_{i,j}^* u_{i,j} = p_i$$

Such a partial isometry is not unique, and we may chose the $u_{i,i+1}$ such that

$$u_{i,j} = \begin{cases} u_{i,i+1} \dots u_{j-1,j} & \text{if } i < j \\ u^*_{j,j+1} \dots u^*_{i-1,i} & \text{if } i > j \\ p_i & \text{if } i = j \end{cases}$$

And we have that

$$u_{i+1,i} = u^*_{i,i+1} \tag{A.1}$$

$$u_{i,j} u_{k,m} = \delta_{jk} u_{i,m} \tag{A.2}$$

Now we define the embedding Θ from the set of $n \times n$ matrices over $\mathbb{C}$ to the II_1 hyperfinite factor $\mathfrak{R}$. Let M be a $n \times n$ matrix, we define $\Theta(\mathrm{M}[i,j]) = \mathrm{M}[i,j]u_{i,j}$, i.e. $\Theta(\mathrm{M}) = \sum_{i,j}^{n} \mathrm{M}[i,j]u_{i,j}$.

Let M_n and M'_n be two matrices over $\mathbb{C}$, we check that Θ commutes with the operations of the algebra. There is not much to check, because Θ basically inherits of all the algebraic properties of $\mathbb{C}$, so we just illustrate with $+$, $\times$ and the transposition.

$$\begin{aligned}
(\Theta \mathrm{M}_n) + (\Theta \mathrm{M}'_n) &= (\sum_{i,j}^{n} \mathrm{M}[i,j]u_{i,j}) + (\sum_{i,j}^{n} \mathrm{M}'[i,j]u_{i,j}) \\
&= \sum_{i,j}^{n} ((\mathrm{M}[i,j]u_{i,j}) + (\mathrm{M}'[i,j]u_{i,j})) \\
&= \sum_{i,j}^{n} (\mathrm{M}[i,j] + \mathrm{M}'[i,j])u_{i,j} \\
&= \Theta(\mathrm{M}_n + \mathrm{M}'_n)
\end{aligned}$$

$$\begin{aligned}
(\Theta \mathrm{M}_n) \times (\Theta \mathrm{M}'_n) &= (\sum_{i,j}^{n} \mathrm{M}[i,j]u_{i,j}) \times (\sum_{i',j'}^{n} \mathrm{M}'[i',j']u_{i',j'}) \\
&= \sum_{i,j}^{n} \sum_{i',j'}^{n} (\mathrm{M}[i,j]u_{i,j} \times \mathrm{M}'[i',j']u_{i',j'}) \\
&= \sum_{i,j}^{n} \sum_{k}^{n} (\mathrm{M}[i,k] \times \mathrm{M}'[k,j])u_{i,j} && \text{(By Equation A.2)} \\
&= \Theta(\sum_{i,j}^{n} \sum_{k}^{n} (\mathrm{M}[i,k] \times \mathrm{M}'[k,j]) \\
&= \Theta(\mathrm{M}_n \times \mathrm{M}'_n)
\end{aligned}$$

$$\begin{aligned}
\Theta(\mathrm{M}^{\mathrm{t}}_n) &= \sum_{i,j}^{k} (\mathrm{M}[j,i]u_{i,j}) \\
&= \sum_{i,j}^{k} (\mathrm{M}[j,i]u^*_{j,i}) && \text{(By Equation A.1)} \\
&= \Theta(\mathrm{M}_n)^*
\end{aligned}$$

So we just checked that Θ is an isomorphism, and we can check that the properties of the normalised trace is preserved by this isomorphism.

In fact Θ depends on the size of n, so we should write $\Theta_{n+1} : \mathrm{M}_n(\mathbb{C}) \to \mathfrak{R}$, and we should also remark that such an embedding is not unique.

Groups and von Neumann algebras

We will in the following show how to internalize the action of a group into a von Neumann algebra. We need first some preliminaries definitions.

Definition A.2.8 (Topological group)**.** Let $\mathcal{G} = (E, +)$ be a group endowed with a topology $\mathcal{T}$, $\mathcal{G}$ is a *topological group* if $+$ and $(\cdot)^{-1}$ are continuous maps.

A topological group is a *compact group* if it is compact, any compact group is locally compact.

Definition A.2.9 (Action of a group)**.** Let $\mathcal{G} = (E, +)$ be a group and X be a set, an action of $\mathcal{G}$ on X is a function $\alpha : E \times X \to X$ such that for every $a, b \in E$ and $x \in X$, $\alpha(e_+, x) = x$ and $\alpha(a, \alpha(b, x)) = \alpha(a + b, x)$.

Equivalently, if we set $\mathcal{S}(X)$ to be the group of all permutations of X, an action α of $\mathcal{G}$ on X is a homomorphism $\alpha : E \to \mathcal{S}(X)$.

An action of a topological group $\mathcal{G}$ on a von Neumann algebra $\mathfrak{M}$ is a continuous homomorphism of $\mathcal{G}$ into $\mathrm{Aut}(\mathfrak{M})$, the set of automorphisms of $\mathfrak{M}$.

Definition A.2.10 (Representations)**.** Let $\mathfrak{M}$ be a von Neumann algebra. A couple $(\mathbb{H}, \rho)$ where $\mathbb{H}$ is a Hilbert space and ρ is a $*$-homomorphism from $\mathfrak{M}$ to $\mathcal{B}(\mathbb{H})$ is a *representation* of $\mathfrak{M}$. If ρ is injective, we say the representation is *faithful*.

Haagerup [68] proved that among the numerous representations of a von Neumann algebra, there was a *standard representation*, a representation satisfying several important properties.

The operation that will be of interest to us will be that of taking the *crossed product* of an algebra and a group. This operation is closely related to that of semi-direct product of groups and is a way of internalizing automorphisms. Given an algebra $\mathfrak{M}$ and a group $\mathcal{G}$ of automorphisms of $\mathfrak{M}$, we construct the algebra $\mathfrak{M} \rtimes \mathcal{G}$ generated by the elements of $\mathfrak{M}$ and the elements of $\mathcal{G}$.

Definition A.2.11 (Crossed product (representations))**.** Let $(\mathbb{H}, \rho)$ be a representation of a von Neumann algebra $\mathfrak{M}$, $\mathcal{G}$ a locally compact group, and α an action of $\mathcal{G}$ on $\mathfrak{M}$. Let $\mathbb{K} = L^2(\mathcal{G}, \mathbb{H})$ be the Hilbert space of square-summable $\mathbb{H}$-valued functions on $\mathcal{G}$. We define representations π_α of $\mathfrak{M}$ and λ of $\mathcal{G}$ on $\mathbb{K}$ as follows:

$$(\pi_\alpha(u).x)(g) = (\rho(\alpha(g)^{-1}(u))x(g) \qquad (u \in \mathfrak{M}, x \in \mathbb{H}, g \in \mathcal{G})$$
$$(\lambda(g).x)(h) = x(g^{-1}h) \qquad (g, h \in \mathcal{G}, x \in \mathbb{K})$$

Then the von Neumann algebra on $\mathbb{K}$ generated by $\pi_\alpha(\mathfrak{M})$ and $\lambda(\mathcal{G})$ is called the crossed product of $(\mathbb{H}, \rho)$ by α.

An important fact is that the result of the crossed product does not depend on the chosen representation of $\mathfrak{M}$.

Theorem A.2.1 (Takesaki [127, Theorem 1.7, p. 241])**.** *Let $(\mathbb{H}, \rho)$ and $(\mathbb{K}, \rho')$ be two faithful representations of a von Neumann algebra $\mathfrak{M}$, and let $\mathcal{G}$ be a locally compact group together with an action α on $\mathfrak{M}$. Then there exists an isomorphism between the crossed product of $(\mathbb{H}, \rho)$ by α and the crossed product $(\mathbb{K}, \rho')$ by α.*

As a consequence, one can define *the* crossed product of a von Neumann algebra and a group acting on it by choosing a particular representation. Of course, the natural choice is to consider the *standard representation*.

Definition A.2.12 (Crossed product)**.** Let $\mathfrak{M}$ be a von Neumann algebra, $\mathcal{G}$ a group and α an action of $\mathcal{G}$ on $\mathfrak{M}$. The algebra $\mathfrak{M} \rtimes_\alpha \mathcal{G}$ is defined as the crossed product of the standard representation of $\mathfrak{M}$ by α.

A particular case of crossed product is the crossed product of $\mathbb{C}$ by a (trivial) action of a group $\mathcal{G}$. The resulting algebra is usually called the group von Neumann algebra $\mathfrak{N}(\mathcal{G})$ of $\mathcal{G}$. As it turns out, the operation of internalizing automorphisms of algebras (the crossed product) and the operation of internalizing automorphisms of groups (the semi-direct product) correspond: the algebra $\mathfrak{N}(\mathcal{G} \rtimes_\alpha H)$ is isomorphic to $\mathfrak{N}(\mathcal{G}) \rtimes_{\tilde{\alpha}} H$ where $\tilde{\alpha}$ is the action of H on $\mathfrak{N}(\mathcal{G})$ induced by the action of H on $\mathcal{G}$.

APPENDIX B

An Alternate Proof of co-NL ⊆ NPM

Contents

This short appendix is here to recollect the first version [10] of the proof that **co-NL** ⊆ **NPM**, a part of Corollary 4.3.2 according to the notation in this thesis. It is quite different to the one given in Chapter 4 for it does not use the characterization of **NL** provided by FA, but exhibits a NPM that solves a **co-NL**-complete problem and a *transducer* that perform reductions between any **co-NL** problem and the problem we solve. We develop a "pointer reduction" a bit complex and yet interesting, for it was not previously developed as far as we know.

This proof was written when we were not aware of the existence of FA: it is in some sense more direct, but it does not present our PMs as a member of the large family of "reasonable" computational model. We will use the most common method[1]: first we define and solves a **co-NL**-complete problem with a NPM, and then define a "pointer reduction". Of course a **co-NL** problem is also a **NL** problem, but once again we won't take this equality for granted.

Numerous elements of Chapter 1 and Chapter 4 are needed to understand this appendix, for we won't recall for instance the definitions of reductions (Definition 1.1.8 and Definition 1.1.9), complete problem (Definition 1.1.10), computation graph (Definition 1.1.5), NPM (Definition 4.3.1). Some of the claims stated in Section 4.2 will be used "as it", even if they were proved for FA and not for PM, we consider the proofs could be adapted really easily.

[1] That can be reminded to the reader for instance in Arora and Barak [4, pp. 88–89].

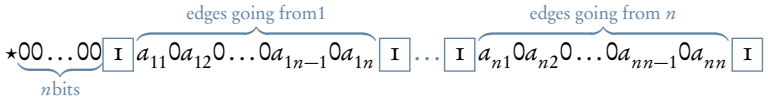
B.1 Solving a co-NL-Complete Problem

We are going to solve the most common **co-NL**-complete problem, the complementary of *stconn* (Problem B). It is reasonable to modify *stconn* so that the path we are looking for is always from the node numbered 1 to the "last" node, i.e. the one with the highest number, which corresponds to the size of the graph.

We define the set

$$STConnComp = \{k \in \mathbb{N} \mid k \text{ encode a directed graph of size } n \text{ with } \textbf{no} \text{ path from 1 to } n\}$$

As we will produce the actual transition function that solves this problem, we should be precise regarding the encoding of a graph we adopt: given a graph of size n, the input will be



where (a_{ij}) is the adjacency list , i.e. $a_{ij} = 1$ if there is an edge from the node numbered by i to the node numbered by j, 0 elsewhere. The boxed bits in the figure above are "separating" bits, between the unary encoding of n and the adjacency list, and between the coding of the edges going from i to the coding of the edges going from $i+1$. This will allow us a trick in our algorithm: when we read the bit following a_{ij}, if it is a 0 then the next bit corresponds to a_{ij+1}, elsewhere it corresponds to a_{i+11}.

Proposition B.1.1. *There exists* $M \in \text{NPM}(5)$ *that decides STConnComp .*

Proof. We define M such that $M_s(k)$ with $s = \{\star, \star, \star, \star, \textbf{Init}\}$ accepts iff $k \in STConnComp$.

The transition relation of M is presented in Figure B.1. Informally, our algorithm goes as follow:

- We have a counter on the size of the path followed, p_1. It moves right on the unary expression of n every time an edge is found (on the second line of Equation B.10). If it reads a 1 after moving (Equation B.12), it means we already followed n edges, so we accept because if there is a path from 1 to n then there exists a path of size at most n. Note that p_1 does not move in any other transition, except for the initialisation.

- The couple p_2, p_3 acts as follow: when p_2 is reading a_{ij}, p_3 is at a_{j1}. If $a_{ij} = 1$ (premise of Equation B.10), a non-deterministic transition takes place: on one way we continue to scan the outgoing edges from i, on the other we increment p_1, place p_2 at a_{j1} and p_3 at a_{11}.

- The last pointer p_4 is here to check when $a_{ij} = 1$ if $j = n$, it also stays on the unary encoding of n. If p_4 reaches a 1 when p_2 reads that there is an edge, it means that there is an edge whose target is n, and so we reject (Equation B.11).

- When p_2 finishes to browse the adjacency list of an edge, we accept (Equation B.6).

We can make easily sure that $M_s(k)$ accepts iff $k \in STConnComp$, elsewhere $M_s(k)$ rejects, i.e. $M_s(k)$ always halts. As defined, M is not a NPM, for it does not fulfil all the requisites we listed in Section 4.3. But we can easily "complete" the transition function: for all $c \in C_M$ such that $c \to \emptyset$, add $c \to \textbf{accept}$. We may also reasonably modify it so that at most one head move at each transition. **Q.E.D.**

$$(\star,\star,\star,\star,\textbf{Init}) \to (p_1+,p_2+,p_3+,p_4+,\textbf{Init}) \tag{B.1}$$
$$(*,0,*,*,\textbf{Init}) \to (\epsilon_1,p_2+,p_3+,\epsilon_4,\textbf{Init}) \tag{B.2}$$
$$(*,1,*,*,\textbf{Init}) \to (\epsilon_1,p_2+,\epsilon_3,\epsilon_4,\textbf{out.edge?}) \tag{B.3}$$
$$(*,0,*,*,\textbf{out.edge?}) \to (\epsilon_1,p_2+,\epsilon_3,p_4+,\textbf{no.edge}) \tag{B.4}$$
$$(*,0,*,*,\textbf{no.edge}) \to (\epsilon_1,\epsilon_2,p_3+,\epsilon_4,\textbf{p3.next.node}) \tag{B.5}$$
$$(*,1,*,*,\textbf{no.edge}) \to \textbf{accept} \tag{B.6}$$
$$(*,*,*,*,\textbf{p3.next.node}) \to (\epsilon_1,\epsilon_2,p_3+,\epsilon_4,\textbf{reading.sep.bit}) \tag{B.7}$$
$$(*,*,0,*,\textbf{reading.sep.bit}) \to (\epsilon_1,\epsilon_,p_3+,\epsilon_4,\textbf{p3.next.node}) \tag{B.8}$$
$$(*,*,1,*,\textbf{reading.sep.bit}) \to (\epsilon_1,p_2+,\epsilon_3,\epsilon_4,\textbf{out.edge?}) \tag{B.9}$$
$$(*,1,*,*,\textbf{out.edge?}) \to \begin{cases}(\epsilon_1,p_2+,\epsilon_3,p_4+,\textbf{no.edge})\\(p_1+,\epsilon_2,\epsilon_3,p_4+,\textbf{edge.found})\end{cases} \tag{B.10}$$
$$(*,*,*,1,\textbf{edge.found}) \to \textbf{reject} \tag{B.11}$$
$$(1,*,*,0,\textbf{edge.found}) \to \textbf{accept} \tag{B.12}$$
$$(*,*,*,0,\textbf{edge.found}) \to (\epsilon_1,p_2-,\epsilon_3,p_4-,\textbf{rewind.p2.p4}) \tag{B.13}$$
$$(*,*,*,[\text{not }\star],\textbf{rewind.p2.p4}) \to (\epsilon_1,p_2-,\epsilon_3,p_4-,\textbf{rewind.p2.p4}) \tag{B.14}$$
$$(*,*,*,\star,\textbf{rewind.p2.p4}) \to (\epsilon_1,p_2-,\epsilon_3,\epsilon_4,\textbf{rewind.p2}) \tag{B.15}$$
$$(*,[\text{not }\star],*,*,\textbf{rewind.p2}) \to (\epsilon_1,p_2-,\epsilon_3,\epsilon_4,\textbf{rewind.p2}) \tag{B.16}$$
$$(*,\star,*,*,\textbf{rewind.p2}) \to (\epsilon_1,p_2+,p_3-,\epsilon_4,\textbf{exchange.p2.p3.}) \tag{B.17}$$
$$(*,*,[\text{not }\star],*,\textbf{exchange.p2.p3}) \to (\epsilon_1,p_2+,p_3-,\epsilon_4,\textbf{exchange.p2.p3.}) \tag{B.18}$$
$$(*,*,\star,*,\textbf{exchange.p2.p3}) \to (\epsilon_1,\epsilon_2,p_3+,\epsilon_4,\textbf{get.p3.to.start}) \tag{B.19}$$
$$(*,*,0,*,\textbf{get.p3.to.start}) \to (\epsilon_1,\epsilon_2,p_3+,\epsilon_4,\textbf{get.p3.to.start}) \tag{B.20}$$
$$(*,*,1,*,\textbf{get.p3.to.start}) \to (\epsilon_1,p_2+,\epsilon_3,\epsilon_4,\textbf{out.edge?}) \tag{B.21}$$

Recall that $*$ (resp. [not $\star$]) is any symbol among $\{0,1,\star\}$ (resp. $\{0,1\}$).

Figure B.1: The transition relation that decides *STConnComp*

B.2 Pointer-Reduction

So now, we have to prove that we can reduce any **co-NL**-problem to *STConnComp* using only pointers. We will adapt the classical log-space implicitly computable function that reduces any problem in **NL** to *stconn*. Given a **co-NL**-problem P, there exists a universally non-deterministic log-space Turing Machine M that decides it. To solve P is just to establish if there is **no** path in the computational graph of M from the initial configuration to a rejecting configuration.

We will in the following:

- shows how PMs can express "any integer" and perform some computation,
- explains the properties of the encoding of $G_{M(n)}$ as an integer,
- prove that a DPM can "compute" $G_{M(n)}$ given M and n,
- explain how to compose PMs, and that will complete the demonstration.

Claim 8: *A PM can express any power of the size of the input.*

Proof. Let fix an input n of size $|n|$, our claim is that given j pointers, a PM can count up to $|n|^j$. We use once again the additional pointers as the hands of a clock: we make the 1st pointer go right, and every time the ith pointer made a round-trip (that is, is back on $\star$), the $i+1$th pointer goes one cell right. Clearly, we can express any distance inferior to $|n|^j$. **Q.E.D.**

We will for the sake of simplicity consider that any distance inferior to $|n|^j$ can be expressed by a single pointer, even if it may require several pointer to be properly expressed.

Claim 9: *A PM can compute addition, subtraction, power, division, multiplication, modulo and binary logarithm.*

Proof. We can adopt the same proof as the one we developed for Claim 5 regarding the FA. To compute log, just remember that $\log(x) = y$ iff $x = 2^y$, so our PM can put a pointer (that will be an estimation for y) on a cell, computes its power of two, and looks if it is the same value as x. It it is, the value we are looking for is found, elsewhere it just make the pointer move one cell right and makes again this computation. **Q.E.D.**

Given any problem $P \in$ **co-NL**, there exists a universally non-deterministic Turing Machine M such that M accepts n[2] iff $n \in P$. We assume that P is a decision problem, that M works on alphabet $\{0,1\}$, has one read-only tape and one read-write working tape whose precise bound is $k \times (\log(|n|))$. We may also assume that the name of the states is written in binary, so that for $|Q| = q$ the number of states of M, any state may be written with $\lceil \log(q) \rceil$ bits. At last, we may assume that the instructions to move the heads are written with two bits. All those assumptions make clear that M may be entirely described with a binary string.

We remind the reader that the computation graph $G_{M(n)}$ is finite and acyclic and that its size is the number of configuration of $M(n)$. We can take M to always halt, so $M(n)$ rejects iff there is a branch that reaches **reject**, and we can also assume that only one configuration has for state **reject**: this will allow us to test only for the existence of a path in $G_{M(n)}$ from the initial configuration to that configuration.

[2] Meaning that **all** branches reach **accept** after a finite number of transitions.

We know that $M(n)$ has less than

$$2^{(k\times(\log(|n|)))}\times(k\times(\log(|n|)))\times(\log(|n|))\times\lceil\log(q)\rceil$$

different configurations. It reflects respectively the content of the working tape, the position of the read-write and the read-only heads and the state. This is equivalent to $2^{\mathcal{O}(\log(|n|))}$, so we know there exists a d such that $M(n)$ has less than $|n|^d$ different configurations.

Any configuration of $M(n)$ may be described as

$$\underbrace{0100011110\ldots010110}_{\text{Position of the read head}}\overbrace{\sigma0\sigma0\ldots\sigma0\sigma1\sigma0\ldots\sigma0\sigma0\sigma0\sigma0\sigma0\sigma0\sigma0}^{\text{Working tape and position of the working head}}\underbrace{01\ldots10}_{\text{state}}$$

where the $\lceil\log(|n|)\rceil$ first bits encode in binary the position of the reading head, the symbols σ correspond to the bits on the working tape, the bit that follows σ equals 1 iff the working head is on that cell. The remaining $\lceil\log(q)\rceil$ bits express the current state.

This binary string is of length $\lceil\log(|n|)\rceil\times(2\times(k\times\lceil\log|n|\rceil))\times\lceil\log(q)\rceil$, i.e. there exists a e such that this string is of length inferior to $e\times\log(|n|)^2$.

Among all the binary string of size $e\times\log(|n|)^2$, some of them corresponds to configurations, and some of them don't (for instance because the working head is supposed to be in several places at the same time), we will call them "phantom configurations".

Lemma B.2.1 (Pointer-reduction)**.** *For all universally non-deterministic* log*-space Turing Machine M, there exists a DPM T such that for all n, given a pointer p_d with $\sharp p_d = j$, T accepts if the jth bit of the encoding of $G_{M(n)}$ is* 1*, rejects if it is* 0*.*

Proof. Recall we use the encoding exposed earlier to express the encoding of $G_{M(n)}$ the computation graph of M on input n. Let T proceed as follows:

- It computes the number of binary strings of size $e\times\log(|n|)^2$. This number is bounded by $2^{(e\times\log(|n|))^2}$ and we saw previously that a PM could express such distances. Then T compares this value to j : if j is inferior, it rejects, if j is equal, it accepts, elsewhere it goes on. This reflects the n initial bits set to 0 to express in unary the size of the graph.
- Elsewhere T establishes if j corresponds to a "separating bit" and accepts or rejects accordingly, that can be simply made with the division and modulo operations.
- Elsewhere, j encodes a query regarding the presence or absence of transition between two configurations a and b. If $a = b$, there is no need to explore this transition,[3] and T rejects. Elsewhere T establishes if there is a transition between a and b, and accepts or rejects accordingly.

This last point needs to be made more precise: if $j > 2^{e\times\log(|n|)^2}$ and if j does not corresponds to a "separating bit", it means that the value of j corresponds to the absence or presence of an edge between two nodes. So there exists a and b such that $j = a_{ab}$. A simple arithmetic of pointers allows us to retrieve those two values expressed as integers (i.e. as a distance).

Then, they are converted to binary strings: the positions of the read-only heads needs a bit of pointer arithmetic to be obtained and compared, but the rest of the integer just needs to be compared

[3] Because that would imply that there is a transition from a configuration to itself, and so $M(n)$ is stuck in a loop.

bitwise. The rest of the binary expression of the node encodes directly the configuration, and as all the transitions makes only local changes in them, there is only a constant number of information to be remembered.

Every time there is a difference between the binary expression of a and the binary expression of b, T checks that the difference between them is legal regarding the transition function of M —that may be encoded in the states of T or may be given as a parameter.

The transducer T also have to check that a and b are not "phantom configurations" and that j is not "too big", i.e. does not represent a query on nodes that does not exists. **Q.E.D.**

One should notice that even if this computation is quite violent, for the number of pointer of T will be *very* high, we did not used non-determiancy, so T is a DPM.

Corollary B.2.2.

co-NL ⊆ ***NPM***

Proof. Let $P \in$ **co-NL**, there exists a universally non-deterministic Turing Machines N such that $\mathcal{L}(N) = P$. Suppose given $n \in \mathbb{N}$, we will compose the NPM M that solves *STConnComp* with the transducer T that compute $G_{M(n)}$.

Every time M has to read a value, it asks T by letting a pointer be on the position j of the value it wants to know. There is some kind of layer of abstraction in this composition, for M go through the input tape without ever reading the actual values, but asks the values to T, which actually read n.

We have to make sure that the j of Lemma B.2.1 can be big enough: what is the size of the encoding of $G_{N(n)}$? We encode the graph as being of size $2^{e \times \log(|n|)^2}$, i.e. we also take "phantom configurations" to be nodes of $G_{N(n)}$. The encoding of this "completed" graph —for every string of size $e \times \log(|n|)^2$ is taken to be one of its node, even if it is not reachable— is of size $\mathcal{O}((2^{\log(|n|)^2})^2)$, an expression bounded by a power of $|n|$, so we can express it.

We can suppose moreover that there is a transition between the "phantom configuration" encoded by `000...001` and the initial configuration, and that there exists a transition between any rejecting configuration and the "phantom configuration" encoded by 111...111. This allows us to keep the *STConnComp* algorithm as it it, computing only if there is no path from the node 1 to the node n.

The transducer T can compute the configuration graph of $N(x)$ bit-by-bit and pass it to N which solves *STConnComp* . So $M \circ T(n)$ accepts iff there is no path from 1 to an rejecting configuration in $G_{N(n)}$, i.e. iff $N(n)$ accepts. So $P \in$ **NPM**. **Q.E.D.**

Index

Bibliography

[1] Eric W. Allender, Michael C. Loui, and Kenneth W. Regan. "Complexity Classes". In: *Algorithms and Theory of Computation Handbook*. Edited by Mikhail J. Atallah and Marina Blanton. CRC, 1998, pages 674–696. DOI: 10.1201/9781584888239 (cited on p. 4).

[2] Eric W. Allender, Michael C. Loui, and Kenneth W. Regan. "Reducibility and Completeness". In: *Algorithms and Theory of Computation Handbook*. Edited by Mikhail J. Atallah and Marina Blanton. CRC, 1998, pages 697–724. DOI: 10.1201/9781584888239 (cited on p. 4).

[3] Jean-Marc Andreoli, Roberto Maieli, and Paul Ruet. "Non-commutative proof construction: A constraint-based approach". In: *Annals of Pure and Applied Logic* 142.1–3 (2006), pages 212–244. DOI: 10.1016/j.apal.2006.01.002 (cited on p. 54).

[4] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Volume 1. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511804090 (cited on pp. 4, 10, 141).

[5] Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. "Paths in the lambda-calculus". In: *LICS*. IEEE Computer Society, 1994, pages 426–436. DOI: 10.1109/LICS.1994.316048 (cited on p. 17).

[6] Mikhail J. Atallah and Marina Blanton, editors. *Algorithms and Theory of Computation Handbook*. CRC, 1998. DOI: 10.1201/9781584888239 (cited on p. 4).

[7] Clément Aubert. "Réseaux de preuves booléens sous-logarithmiques". Master's thesis. L.I.P.N.: L.M.F.I., Paris VII, Sept. 2010 (cited on p. 32).

[8] Clément Aubert. "Sublogarithmic uniform Boolean proof nets". In: *DICE*. Edited by Jean-Yves Marion. Volume 75. Electronic Proceedings in Theoretical Computer Science. 2011, pages 15–27. DOI: 10.4204/EPTCS.75.2 (cited on p. 32).

[9] Clément Aubert and Marc Bagnol. "Unification and Logarithmic Space". In: *RTA-TLCA*. Edited by Gilles Dowek. Volume 8650. Lecture Notes in Computer Science. Springer, 2014, pages 77–92. DOI: 10.1007/978-3-319-08918-8_6 (cited on pp. 90, 120).

[10] Clément Aubert and Thomas Seiller. "Characterizing co-NL by a group action". In: *Arxiv preprint* abs/1209.3422 (2012). arXiv: 1209.3422 [cs.LO] (cited on pp. 58, 69 sq., 82 sq., 94, 106, 110, 112, 141).

[11] Clément Aubert and Thomas Seiller. "Logarithmic Space and Permutations". In: *Arxiv preprint* abs/1301.3189 (2013). arXiv: 1301.3189 [cs.LO] (cited on pp. 58, 82, 94, 106, 110).

[12] Sheldon Axler. *Linear Algebra Done Right*. Undergraduate Texts in Mathematics. Springer, 1997. DOI: 10.1007/b97662 (cited on p. 125).

[13] Patrick Baillot and Damiano Mazza. "Linear Logic by Levels and Bounded Time Complexity". In: *Theoretical Computer Science* 411.2 (2010), pages 470–503. DOI: 10.1016/j.tcs.2009.09.015 (cited on p. 18).

[14] Patrick Baillot and Marco Pedicini. "Elementary Complexity and Geometry of Interaction". In: *Fundamenta Informaticae* 45.1–2 (2001), pages 1–31 (cited on pp. 18, 58).

[15] David A. Barrington, Neil Immerman, and Howard Straubing. "On uniformity within NC^1". In: *Journal of Computer and System Sciences* 41.3 (1990), pages 274–306. DOI: 10.1016/0022-0000(90)90022-D (cited on pp. 11, 23).

[16] Amir M. Ben-Amram. "What is a "Pointer Machine"?" In: *Science of Computer Programming*. Volume 26. 2. Association for Computing Machinery, June 1995, pages 88–95. DOI: 10.1145/202840.202846 (cited on p. 94).

[17] Amir M. Ben-Amram and Neil D. Jones. "Computational Complexity via Programming Languages: Constant Factors Do Matter". In: *Acta Informatica* 37.2 (Oct. 2000), pages 83–120. DOI: 10.1007/s002360000038 (cited on p. 123).

[18] Guillaume Bonfante and Virgile Mogbil. *A circuit uniformity sharper than DLogTime*. Technical report. CARTE - LIPN, Apr. 2012 (cited on p. 23).

[19] Allan Borodin. "On Relating Time and Space to Size and Depth". In: *SIAM Journal on Computing* 6.4 (1977), pages 733–744. DOI: 10.1137/0206054 (cited on p. 22).

[20] Allan Borodin, Stephen Arthur Cook, Patrick W. Dymond, Walter L. Ruzzo, and Martin Tompa. "Two Applications of Inductive Counting for Complementation Problems". In: *SIAM Journal on Computing* 18.3 (1989), pages 559–578. DOI: 10.1137/0218038 (cited on p. 54).

[21] Egon Börger, editor. *Journal for Universal Computer Science* 3.4 (1997): *Ten years of Gurevich's Abstract State Machines*. DOI: 10.3217/jucs-003-04 (cited on p. 97).

[22] Jin-Yi Cai, Martin Fürer, and Neil Immerman. "An optimal lower bound on the number of variables for graph identifications". In: *Combinatorica* 12.4 (1992), 389––410. DOI: 10.1007/BF01305232 (cited on p. 95).

[23] Liming Cai and Jianer Chen. "On input read-modes of alternating Turing machines". In: *Theoretical Computer Science* 148.1 (1995), pages 33–55. DOI: 10.1016/0304-3975(94)00253-F (cited on p. 23).

[24] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. "Alternation". In: *Journal of the ACM* 28.1 (Jan. 1981), pages 114–133. DOI: 10.1145/322234.322243 (cited on pp. 5 sq., 9).

[25] Ashok K. Chandra, Larry J. Stockmeyer, and Uzi Vishkin. "Constant Depth Reducibility". In: *SIAM Journal on Computing* 13.2 (1984), 423–439. DOI: 10.1137/0213028 (cited on p. 11).

[26] John B. Conway. *A Course in Functional Analysis*. Volume 96. Graduate Texts in Mathematics. Springer, 1990 (cited on p. 126).

[27] Stephen Arthur Cook. "A Taxinomy of Problems with Fast Parallel Algorithms". In: *Information and Control* 64.1-3 (1985), pages 2–22. DOI: 10.1016/S0019-9958(85)80041-3 (cited on p. 45).

[28] Stephen Arthur Cook and Pierre McKenzie. "Problems Complete for Deterministic Logarithmic Space". In: *Journal of Algorithms* 8.3 (1987), pages 385–394. DOI: 10.1016/0196-6774(87)90018-6 (cited on p. 12).

[29] Stephen Arthur Cook and Charles W. Rackoff. "Space Lower Bounds for Maze Threadability on Restricted Machines". In: *SIAM Journal on Computing* 9.3 (1980), pages 636–652. DOI: 10.1137/0209048 (cited on p. 95).

[30] Pierre-Louis Curien. *Notes on operator algebras, part I*. Lecture Notes. Mar. 2011 (cited on p. 125).

[31] Ugo Dal Lago. "Context Semantics, Linear Logic, and Computational Complexity". In: *ACM Transactions on Computational Logic* 10.4 (Aug. 2009), 25:1–25:32. DOI: 10.1145/1555746.1555749 (cited on p. 58).

[32] Ugo Dal Lago and Claudia Faggian. "On Multiplicative Linear Logic, Modality and Quantum Circuits". In: *QPL*. Edited by Bart Jacobs, Peter Selinger, and Bas Spitters. Volume 95. Electronic Proceedings in Theoretical Computer Science. 2011, pages 55–66. DOI: 10.4204/EPTCS.95.6 (cited on p. 59).

[33] Ugo Dal Lago and Olivier Laurent. "Quantitative Game Semantics for Linear Logic". In: *CSL*. Edited by Michael Kaminski and Simone Martini. Volume 5213. Lecture Notes in Computer Science. Springer, 2008, pages 230–245. DOI: 10.1007/978-3-540-87531-4_18 (cited on p. 58).

[34] Ugo Dal Lago, Simone Martini, and Margherita Zorzi. "General Ramified Recurrence is Sound for Polynomial Time". In: *DICE*. Edited by Patrick Baillot. Volume 23. Electronic Proceedings in Theoretical Computer Science. 2010, pages 47–62. DOI: 10.4204/EPTCS.23.4 (cited on p. 3).

[35] Ugo Dal Lago and Ulrich Schöpp. "Type Inference for Sublinear Space Functional Programming". In: *APLAS*. Edited by Kazunori Ueda. Volume 6461. Lecture Notes in Computer Science. Springer, 2010, pages 376–391. DOI: 10.1007/978-3-642-17164-2_26 (cited on p. 18).

[36] Vincent Danos. "La Logique Linéaire appliquée à l'étude de divers processus de normalisation (principalement du λ-calcul)". PhD thesis. Université Paris VII, 1990 (cited on pp. 37 sq.).

[37] Vincent Danos and Jean-Baptiste Joinet. "Linear Logic & Elementary Time". In: *Information and Computation* 183.1 (2003), pages 123–137. DOI: 10.1016/S0890-5401(03)00010-5 (cited on p. 18).

[38] Vincent Danos and Laurent Regnier. "The Structure of Multiplicatives". In: *Archive for Mathematical logic* 28.3 (1989), pages 181–203. DOI: 10.1007/BF01622878 (cited on pp. 19, 24, 26).

[39] Daniel de Carvalho, Michele Pagani, and Lorenzo Tortora de Falco. "A Semantic Measure of the Execution Time in Linear Logic". In: *Theoretical Computer Science* 412.20 (Apr. 2011): *Girard's Festschrift*. Edited by Thomas Ehrhard, Claudia Faggian, and Olivier Laurent, pages 1884–1902. DOI: 10.1016/j.tcs.2010.12.017 (cited on p. 18).

[40] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Perspectives in Mathematical Logic. Springer, 1995, pages I–XV, 1–327 (cited on p. 94).

[41] Thomas Ehrhard, Claudia Faggian, and Olivier Laurent, editors. *Theoretical Computer Science* 412.20 (Apr. 2011): *Girard's Festschrift*.

[42] Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Philip J. Scott, editors. *Linear Logic in Computer Science*. London Mathematical Society Lecture Note Series 316. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511550850.

[43] Igor Fulman. *Crossed products of von Neumann algebras by equivalence relations and their subalgebras*. Volume 126. Memoirs of the American Mathematical Society 602. AMS Bookstore, 1997. DOI: 10.1090/memo/0602 (cited on p. 90).

[44] Merrick L. Furst, James B. Saxe, and Michael Sipser. "Parity, Circuits, and the Polynomial-Time Hierarchy". In: *Theory of Computing Systems* 17.1 (1984), pages 13–27. DOI: 10.1007/BF01744431 (cited on p. 11).

[45] Gerhard Gentzen. "Untersuchungen über das Logische Schliessen". In: *Mathematische Zeitschrift* 39 (1935), 176–210 and 405–431 (cited on p. 14).

[46] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen*. Edited by M. E. Szabo. Studies in Logic and the Foundations of Mathematics. North-Holland, 1969 (cited on p. 14).

[47] Jean-Yves Girard. "Linear logic". In: *Theoretical Computer Science* 50.1 (1987), pages 1–101. DOI: 10.1016/0304-3975(87)90045-4 (cited on pp. 12, 16, 28).

[48] Jean-Yves Girard. "Geometry of interaction 1: Interpretation of System F". In: *Studies in Logic and the Foundations of Mathematics* 127 (1989), pages 221–260. DOI: 10.1016/S0049-237X(08)70271-4 (cited on pp. 58, 106, 110).

[49] Jean-Yves Girard. "Towards a geometry of interaction". In: *Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference held June 14-20, 1987*. Edited by John W. Gray and Andre Ščedrov. Volume 92. Categories in Computer Science and Logic. American Mathematical Society, 1989, pages 69–108. DOI: 10.1090/conm/092/1003197 (cited on pp. 16 sq., 19, 54, 63, 106).

[50] Jean-Yves Girard. "Geometry of interaction III: accommodating the additives". In: *Advances in Linear Logic*. Edited by Jean-Yves Girard, Yves Lafont, and Laurent Regnier. London Mathematical Society Lecture Note Series 222. Cambridge University Press, June 1995, pages 329–389. DOI: 10.1017/CBO9780511629150.017 (cited on pp. 76, 126).

[51] Jean-Yves Girard. "Proof-Nets: The Parallel Syntax for Proof-Theory". In: *Logic and Algebra*. Lecture Notes in Pure and Applied Mathematics 180 (May 1996), pages 97–124 (cited on p. 14).

[52] Jean-Yves Girard. "Light Linear Logic". In: *Information and Computation* 143.2 (1998), pages 175–204. DOI: 10.1006/inco.1998.2700 (cited on p. 18).

[53] Jean-Yves Girard. *Introduction aux algèbres d'opérateurs I : Des espaces de Hilbert aux algèbres stellaires*. Lecture Notes. Sept. 2002 (cited on p. 125).

[54] Jean-Yves Girard. "Between Logic and Quantic: a Tract". In: *Linear Logic in Computer Science*. Edited by Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Philip J. Scott. London Mathematical Society Lecture Note Series 316. Cambridge University Press, 2004, pages 346–381. DOI: 10.1017/CBO9780511550850.011 (cited on p. 59).

[55] Jean-Yves Girard. *Le Point Aveugle I : Vers la Perfection*. Visions des sciences. Hermann, July 2006 (cited on p. 12).

[56] Jean-Yves Girard. *Le Point Aveugle II : Vers l'Imperfection*. Visions des sciences. Hermann, Mar. 2007 (cited on pp. 12, 90).

[57] Jean-Yves Girard. "Geometry of Interaction V: logic in the hyperfinite factor". In: *Theoretical Computer Science* 412.20 (Apr. 2011): *Girard's Festschrift*. Edited by Thomas Ehrhard, Claudia Faggian, and Olivier Laurent, pages 1860–1883. DOI: 10.1016/j.tcs.2010.12.016 (cited on pp. 16, 57, 61).

[58] Jean-Yves Girard. *The Blind Spot: Lectures on Logic*. Edited by European Mathematical Society. Sept. 2011, page 550. DOI: 10.4171/088 (cited on pp. 12, 60).

[59] Jean-Yves Girard. "Normativity in Logic". In: *Epistemology versus Ontology*. Edited by Peter Dybjer, Sten Lindström, Erik Palmgren, and Göran Sundholm. Volume 27. Logic, Epistemology, and the Unity of Science. Springer, 2012, pages 243–263. DOI: 10.1007/978-94-007-4435-6_12 (cited on pp. 58, 61, 67, 80–82, 93, 111).

[60] Jean-Yves Girard. "Three lightings of logic". In: *CSL*. Edited by Simona Ronchi Della Rocca. Volume 23. Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pages 11–23. DOI: 10.4230/LIPIcs.CSL.2013.11 (cited on p. 90).

[61] Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors. *Advances in Linear Logic*. London Mathematical Society Lecture Note Series 222. Cambridge University Press, June 1995.

[62] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Volume 7. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989 (cited on pp. 12, 63, 90).

[63] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. "Bounded linear logic: a modular approach to polynomial-time computability". In: *Theoretical Computer Science* 97.1 (1992), pages 1–66. DOI: 10.1016/0304-3975(92)90386-T (cited on p. 18).

[64] Leslie M. Goldschlager. "A Unified Approach to Models of Synchronous Parallel Machines". In: *STOC*. Edited by Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho. Association for Computing Machinery, 1978, pages 89–94. DOI: 10.1145/800133.804336 (cited on p. 5).

[65] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation : P-Completeness theory*. Oxford UP, 1995 (cited on pp. 4, 22).

[66] Erich Grädel and Gregory L. McColm. "On the Power of Deterministic Transitive Closures". In: *Information and Computation* 119.1 (1995), 129––135. DOI: 10.1006/inco.1995.1081 (cited on p. 95).

[67] Stefano Guerrini and Andrea Masini. "Parsing MELL proof nets". In: *Theoretical Computer Science* 254.1–2 (2001), pages 317–335. DOI: 10.1016/S0304-3975(99)00299-6 (cited on p. 37).

[68] Uffe Haagerup. "The Standard Form of von Neumann Algebras". In: *Mathematica Scandinavia* 37.271–283 (1975) (cited on p. 137).

[69] Juris Hartmanis. "On Non-Determinancy in Simple Computing Devices". In: *Acta Informatica* 1.4 (1972), pages 336–344. DOI: 10.1007/BF00289513 (cited on pp. 98 sq., 104).

[70] William Hesse, Eric W. Allender, and David A. Barrington. "Uniform constant-depth threshold circuits for division and iterated multiplication". In: *Journal of Computer and System Sciences* 65.4 (2002), pages 695–716. DOI: 10.1016/S0022-0000(02)00025-9 (cited on p. 23).

[71] Martin Hofmann, Ramyaa Ramyaa, and Ulrich Schöpp. "Pure Pointer Programs and Tree Isomorphism". In: *FoSSaCS*. Edited by Frank Pfenning. Volume 7794. Lecture Notes in Computer Science. Springer, 2013, pages 321–336. DOI: 10.1007/978-3-642-37075-5_21 (cited on p. 95).

[72] Martin Hofmann and Ulrich Schöpp. "Pointer Programs and Undirected Reachability". In: *LICS*. IEEE Computer Society, 2009, pages 133–142. DOI: 10.1109/LICS.2009.41 (cited on p. 95).

[73] Martin Hofmann and Ulrich Schöpp. "Pure Pointer Programs with Iteration". In: *ACM Transactions on Computational Logic* 11.4 (2010). DOI: 10.1145/1805950.1805956 (cited on p. 95).

[74] Markus Holzer and Martin Kutrib. "Descriptional and computational complexity of finite automata - A survey". In: *Information and Computation* 209.3 (2011), pages 456–470. DOI: 10.1016/j.ic.2010.11.013 (cited on p. 106).

[75] Markus Holzer, Martin Kutrib, and Andreas Malcher. "Multi-Head Finite Automata: Characterizations, Concepts and Open Problems". In: *CSP*. Edited by Turlough Neary, Damien Woods, Anthony Karel Seda, and Niall Murphy. Volume 1. Electronic Proceedings in Theoretical Computer Science. 2008, pages 93–107. DOI: 10.4204/EPTCS.1.9 (cited on pp. 98, 104).

[76] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. international edition (2. ed). Addison-Wesley, 2003, pages I–XIV, 1–521 (cited on pp. 98, 107).

[77] Neil Immerman. "Languages that Capture Complexity Classes". In: *SIAM Journal on Computing* 16.4 (1987), pages 760–778. DOI: 10.1137/0216051 (cited on p. 3).

[78] Neil Immerman. "Nondeterministic space is closed under complementation". In: *CoCo*. IEEE Computer Society, 1988, pages 112–115. DOI: 10.1109/SCT.1988.5270 (cited on p. 11).

[79] Neil Immerman. *Descriptive complexity*. Graduate Texts in Computer Science. Springer, 1999, pages I–XVI, 1–268. DOI: 10.1007/978-1-4612-0539-5 (cited on p. 94).

[80] Paulin Jacobé de Naurois and Virgile Mogbil. "Correctness of linear logic proof structures is NL-complete". In: *Theoretical Computer Science* 412.20 (Apr. 2011): *Girard's Festschrift*. Edited by Thomas Ehrhard, Claudia Faggian, and Olivier Laurent, pages 1941–1957. DOI: 10.1016/j.tcs.2010.12.020 (cited on p. 17).

[81] Birgit Jenner, Klaus-Jörn Lange, and Pierre McKenzie. *Tree Isomorphism and Some Other Complete Problems for Deterministic Logspace*. Technical report. Université de Montréal - DIRO, 1997 (cited on p. 11).

[82] Tao Jiang, Ming Li, and Bala Ravikumar. "Basic notions in computational complexity". In: *Algorithms and Theory of Computation Handbook*. Edited by Mikhail J. Atallah and Marina Blanton. CRC, 1998. DOI: 10.1201/9781584888239 (cited on p. 4).

[83] Jean-Baptiste Joinet. "Logique et interaction". Habilitation à diriger des recherches. Université Paris Diderot (Paris 7), Dec. 2007 (cited on pp. 12, 15).

[84] Neil D. Jones. "Constant Time Factors *Do* Matter". In: *STOC*. Edited by S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal. San Diego, California, USA: Association for Computing Machinery, 1993, pages 602–611. doi: 10.1145/167088.167244 (cited on p. 123).

[85] Neil D. Jones, Y. Edmund Lien, and William T. Laaser. "New Problems Complete for Nondeterministic Log Space". In: *Mathematical Systems Theory* 10.1 (1976), pages 1–17. doi: 10.1007/BF01683259 (cited on p. 11).

[86] Richard V. Kadison and John R. Ringrose. *Fundamentals of the theory of operator algebras. Vol. I*. Volume 15. Graduate Studies in Mathematics. 1997 (cited on p. 125).

[87] Max I. Kanovich. "Horn Programming in Linear Logic Is NP-Complete". In: *LICS*. IEEE Computer Society, 1992, 200––210. doi: 10.1109/LICS.1992.185533 (cited on p. 17).

[88] K. N. King. "Alternating Multihead Finite Automata". In: *Theoretical Computer Science* 61.2–3 (1988), pages 149–174. doi: 10.1016/0304-3975(88)90122-3 (cited on p. 106).

[89] Andrey Nikolaevic Kolmogorov and Vladimir Andreyevich Uspenskiĭ. "On the definition of an algorithm". In: *Uspekhi Matematicheskikh Nauk* 13.4(82) (1958), 3––28 (cited on p. 96).

[90] Andrey Nikolaevic Kolmogorov and Vladimir Andreyevich Uspenskiĭ. "On the definition of an algorithm". In: *American Mathematical Society Translations, Series 2* 29 (1963), 217––245 (cited on p. 96).

[91] Dexter Kozen. "On Two Letters versus Three". In: *FICS*. Edited by Zoltán Ésik and Anna Ingólfsdóttir. Volume NS-02-2. BRICS Notes Series. University of Aarhus, July 2002, pages 44–50 (cited on p. 100).

[92] Jean-Louis Krivine. *Lambda-calculus, types and models*. Ellis Horwood series in computers and their applications. Masson, 1993, pages I–VIII, 1–180 (cited on p. 14).

[93] Yves Lafont. "From Proof-Nets to Interaction Nets". In: *Advances in Linear Logic*. Edited by Jean-Yves Girard, Yves Lafont, and Laurent Regnier. London Mathematical Society Lecture Note Series 222. Cambridge University Press, June 1995, pages 225–247. doi: 10.1017/CBO9780511629150.012 (cited on p. 37).

[94] Joachim Lambek. "The Mathematics of Sentence Structure". In: *The American Mathematical Monthly* 65.3 (1958), 154–170. doi: 10.2307/2310058 (cited on p. 12).

[95] Olivier Laurent. *Théorie de la démonstration*. LMFI's master *Polycopié* (cited on p. 13).

[96] Olivier Laurent. "A Token Machine for Full Geometry of Interaction (Extended Abstract)". In: *Typed Lambda Calculi and Applications*. Edited by Samson Abramsky. Volume 2044. Lecture Notes in Computer Science. Springer, May 2001, pages 283–297. doi: 10.1007/3-540-45413-6_23 (cited on p. 17).

[97] Daniel Leivant and Jean-Yves Marion. "Evolving Graph-Structures and Their Implicit Computational Complexity". In: *ICALP (2)*. Edited by Fedor V. Fomin, Rusins Martins Freivalds, Marta Z. Kwiatkowska, and David Peleg. Volume 7966. Lecture Notes in Computer Science. Springer, July 2013, 349–360. doi: 10.1007/978-3-642-39212-2_32 (cited on p. 97).

[98] Patrick D. Lincoln. "Deciding Provability of Linear Logic Formulas". In: *London Mathematical Society Lecture Note Series* (1995), 109–122. DOI: 10.1017/CBO9780511629150.006 (cited on p. 17).

[99] Philip J. Maher. "Some operator inequalities concerning generalized inverses". In: *Illinois Journal of Mathematics* 34.3 (1990), 503–514 (cited on p. 118).

[100] Harry Mairson and Kazushige Terui. "On the Computational Complexity of Cut-Elimination in Linear Logic". In: *Theoretical Computer Science* (2003), pages 23–36. DOI: 10.1007/978-3-540-45208-9_4 (cited on pp. 17, 50, 59).

[101] François Maurel. "Nondeterministic Light Logics and NP-Time". In: *Typed Lambda Calculi and Applications*. Edited by Martin Hofmann. Volume 2701. Lecture Notes in Computer Science. Springer, 2003, pages 241–255. DOI: 10.1007/3-540-44904-3_17 (cited on p. 54).

[102] Damiano Mazza. "Non-uniform Polytime Computation in the Infinitary Affine Lambda-Calculus". In: *ICALP (2)*. Edited by Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias. Volume 8573. Lecture Notes in Computer Science. Springer, 2014, pages 305–317. DOI: 0.1007/978-3-662-43951-7_26 (cited on p. 55).

[103] Paul-André Melliès. "A Topological Correctness Criterion for Multiplicative Non-Commutative Logic". In: *Linear Logic in Computer Science*. Edited by Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Philip J. Scott. London Mathematical Society Lecture Note Series 316. Cambridge University Press, 2004, pages 283–322. DOI: 10.1017/CBO9780511550850.009 (cited on p. 54).

[104] Paul-André Melliès and Nicolas Tabareau. "Resource modalities in tensor logic". In: *Annals of Pure and Applied Logic* 161.5 (2010), pages 632–653. DOI: 10.1016/j.apal.2009.07.018 (cited on p. 123).

[105] Virgile Mogbil. "Non-deterministic Boolean Proof Nets". In: *FOPARA*. Edited by Marko C. J. D. van Eekelen and Olha Shkaravska. Volume 6324. Lecture Notes in Computer Science. Springer, 2009, pages 131–145. DOI: 10.1007/978-3-642-15331-0_9 (cited on pp. 22, 32, 54).

[106] Virgile Mogbil. "Complexité en logique linéaire, et logique linéaire en complexité implicite". Habilitation à diriger des recherches en Informatique. Université Paris XIII, Nov. 2012 (cited on pp. 13–15).

[107] Virgile Mogbil and Vincent Rahli. "Uniform Circuits, & Boolean Proof Nets". In: *LFCS*. Edited by Sergei N. Artëmov and Anil Nerode. Volume 4514. Lecture Notes in Computer Science. Springer, 2007, pages 401–421. DOI: 10.1007/978-3-540-72734-7_28 (cited on pp. 22, 32, 40, 43 sq.).

[108] Burkhard Monien. "Two-Way Multihead Automata Over a One-Letter Alphabet". In: *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications* 14.1 (1980), pages 67–82 (cited on p. 104).

[109] Gerard J. Murphy. *C^*-algebras and operator theory*. Academic Press, 1990 (cited on pp. 125 sq., 132).

[110] Michael Aaron Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2010. DOI: 10.1017/CBO9780511976667 (cited on p. 59).

[111] Christos H. Papadimitriou. *Computational Complexity*. Mathematics / a second level course v. 3-4. Addison-Wesley, 1995 (cited on pp. 4 sq.).

[112] Bruno Poizat. *Les petits cailloux*. Aléas, 1995 (cited on p. 22).

[113] Kenneth W. Regan and Heribert Vollmer. "Gap-Languages and Log-Time Complexity Classes". In: *Theoretical Computer Science* 188 (1997), pages 101–116. doi: 10.1016/S0304-3975(96)00288-5 (cited on p. 23).

[114] Omer Reingold. "Undirected connectivity in log-space". In: *Journal of the ACM* 55.4 (Sept. 2008), 17:1–17:24. doi: 10.1145/1391289.1391291 (cited on p. 12).

[115] Walter L. Ruzzo. "On Uniform Circuit Complexity". In: *Journal of Computer and System Sciences* 22.3 (June 1981), pages 365–383. doi: 10.1016/0022-0000(81)90038-6 (cited on pp. 11, 23).

[116] Arnold Schönhage. "Storage Modification Machines". In: *SIAM Journal on Computing* 9.3 (1980), pages 490–508. doi: 10.1137/0209036 (cited on pp. 96 sq.).

[117] Ulrich Schöpp. "Stratified Bounded Affine Logic for Logarithmic Space". In: *LICS*. IEEE Computer Society, 2007, pages 411–420. doi: 10.1109/LICS.2007.45 (cited on p. 18).

[118] Dana S. Scott. "Domains for Denotational Semantics". In: *ICALP*. Edited by Mogens Nielsen and Erik Meineche Schmidt. Volume 140. Lecture Notes in Computer Science. Springer, 1982, 577––613. doi: 10.1007/BFb0012801 (cited on p. 16).

[119] Thomas Seiller. "Interaction Graphs: Additives". In: *Arxiv preprint* abs/1205.6557 (2012). arXiv: 1205.6557 [cs.LO] (cited on p. 69).

[120] Thomas Seiller. "Interaction Graphs: Multiplicatives". In: *Annals of Pure and Applied Logic* 163 (Dec. 2012), pages 1808–1837. doi: 10.1016/j.apal.2012.04.005 (cited on pp. 69, 81, 126).

[121] Thomas Seiller. "Logique dans le facteur hyperfini : géometrie de l'interaction et complexité". PhD thesis. Université de la Méditerranée, 2012 (cited on pp. 64 sq., 69, 81–83, 123, 126).

[122] Peter Selinger. "Towards a quantum programming language". In: *Mathematical Structures in Computer Science* 14.4 (2004), pages 527–586. doi: 10.1017/S0960129504004256 (cited on p. 59).

[123] Michael Sipser. "Borel Sets and Circuit Complexity". In: *STOC*. Edited by David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas. Association for Computing Machinery, 1983, 61––69. doi: 10.1145/800061.808733 (cited on p. 11).

[124] Michael Sipser. *Introduction to the Theory of Computation*. 3rd edition. Cengage Learningy, 2012, pages I–XXII, 1–458 (cited on p. 5).

[125] Ivan Hal Sudborough. "On Tape-Bounded Complexity Classes and Multi-Head Finite Automata". In: *SWAT (FOCS)*. IEEE Computer Society, 1973, 138––144. doi: 10.1109/SWAT.1973.20 (cited on p. 104).

[126] Masamichi Takesaki. *Theory of Operator Algebras 1*. Volume 124. Encyclopedia of Mathematical Sciences. Springer, 2001 (cited on pp. 126, 132).

[127] Masamichi Takesaki. *Theory of Operator Algebras 2*. Volume 125. Encyclopedia of Mathematical Sciences. Springer, 2003 (cited on pp. 126, 137).

[128] Masamichi Takesaki. *Theory of Operator Algebras 3*. Volume 127. Encyclopedia of Mathematical Sciences. Springer, 2003 (cited on p. 126).

[129] Robert Endre Tarjan. "Reference Machines Require Non-linear Time to Maintain Disjoint Sets". In: *STOC*. Edited by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. Association for Computing Machinery, 1977, pages 18–29. DOI: 10.1145/800105.803392 (cited on p. 94).

[130] Kazushige Terui. "Proof Nets and Boolean Circuits". In: *LICS*. IEEE Computer Society, 2004, pages 182–191. DOI: 10.1109/LICS.2004.1319612 (cited on pp. 21, 28, 43).

[131] Peter van Emde Boas. "Space Measures for Storage Modification Machines". In: *Information Processing Letters* 30.2 (1989), 103––110. DOI: 10.1016/0020-0190(89)90117-8 (cited on p. 97).

[132] Peter van Emde Boas. "Machine Models and Simulation". In: *Handbook of Theoretical Computer Science. volume A: Algorithms and Complexity*. Edited by Jan Van Leeuwen. Elsevier, 1990, pages 1–66 (cited on p. 4).

[133] André van Tondervan. "A Lambda Calculus for Quantum Computation". In: *SIAM Journal on Computing* 33.5 (2004), 1109––1135. DOI: 10.1137/S0097539703432165 (cited on p. 59).

[134] Hari Venkateswaran. "Properties that Characterize LOGCFL". In: *Journal of Computer and System Sciences* 43.2 (1991), pages 380–404. DOI: 10.1016/0022-0000(91)90020-6 (cited on p. 55).

[135] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Texts in Theoretical Computer Science An EATCS Series. Springer, 1999. DOI: 10.1007/978-3-662-03927-4 (cited on pp. iii, 6, 8, 11, 22, 46 sq.).

[136] Avi Wigderson. "The Complexity of Graph Connectivity". In: *MFCS*. Edited by Ivan M. Havel and Václav Koubek. Volume 629. Lecture Notes in Computer Science. Springer, 1992, 112––132. DOI: 10.1007/3-540-55808-X_10 (cited on pp. 12, 44).

[137] Andrew Chi-Chih Yao and Ronald L. Rivest. "$k+1$ Heads Are Better Than k". In: *Journal of the ACM* 25.2 (1978), 337––340. DOI: 10.1145/322063.322076 (cited on p. 104).

Linear Logic and Sub-polynomial Classes of Complexity

Abstract This research in Theoretical Computer Science extends the gateways between Linear Logic and Complexity Theory by introducing two innovative models of computation. It focuses on sub-polynomial classes of complexity: AC and NC —the classes of efficiently parallelizable problems— and L and NL —the deterministic and non-deterministic classes of problems efficiently solvable with low resources on space. Linear Logic is used through its Proof Net presentation to mimic with efficiency the parallel computation of Boolean Circuits, including but not restricted to their constant-depth variants. In a second moment, we investigate how operators on a von Neumann algebra can be used to model computation, thanks to the method provided by the Geometry of Interaction, a subtle reconstruction of Linear Logic. This characterization of computation in logarithmic space with matrices can naturally be understood as a wander on simple structures using pointers, parsing them without modifying them. We make this intuition formal by introducing Non Deterministic Pointer Machines and relating them to other well-known pointer-like-machines. We obtain by doing so new implicit characterizations of sub-polynomial classes of complexity.

Keywords Linear Logic, Complexity, Geometry of Interaction, Operator Algebra, Alternating Turing Machines, Boolean Circuits, Uniformity, Proof Nets.

Discipline Computer science

Logique linéaire et classes de complexité sous-polynomiales

Résumé Cette recherche en informatique théorique construit de nouveaux ponts entre logique linéaire et théorie de la complexité. Elle propose deux modèles de machines abstraites qui permettent de capturer de nouvelles classes de complexité avec la logique linéaire, les classes des problèmes efficacement parallélisables (NC et AC) et celle des problèmes solutionnables avec peu d'espace, dans ses versions déterministes et non-déterministes (L et NL). La représentation des preuves de la logique linéaire comme réseaux de preuves est employée pour représenter efficacement le calcul parallèle des circuits booléens, y compris à profondeur constante. La seconde étude s'inspire de la géométrie de l'interaction, une délicate reconstruction de la logique linéaire à l'aide d'opérateurs d'une algèbre de von Neumann. Nous détaillons comment l'interaction d'opérateurs représentant des entiers et d'opérateurs représentant des programmes peut être reconnue nilpotente en espace logarithmique. Nous montrons ensuite comment leur itération représente un calcul effectué par des machines à pointeurs que nous définissons et que nous rattachons à d'autres modèles plus classiques. Ces deux études permettent de capturer de façon implicite de nouvelles classes de complexité, en dessous du temps polynomial.

Mot-clés Logique Linéaire, Complexité, Géométrie de l'interaction, Algèbre d'opérateurs, Machines de Turing Alternantes, Circuits Booléens, Uniformité, Réseaux de Preuves.

Discipline Informatique

Laboratoire d'Informatique de Paris Nord
Institut Galilée - Université Paris-Nord
99, avenue Jean-Baptiste Clément
93430 Villetaneuse, France