Aix-Marseille Université
École doctorale 184
UFR sciences

Institut de Mathématiques de Marseille

Thèse présentée pour obtenir le grade universitaire de docteur

Spécialité: Mathématiques

On the Resolution Semiring

Marc Bagnol

Jury:

Pierre-Louis	Curien	Université Paris Diderot	
Jean-Yves	Girard	Aix-Marseille Université	*(directeur)*
Ugo	dal Lago	Università di Bologna	*(rapporteur)*
Paul-André	Melliès	Université Paris Diderot	
Myriam	Quatrini	Aix-Marseille Université	
Ulrich	Schöpp	LMU München	
Philip	Scott	University of Ottawa	*(rapporteur)*
Kazushige	Terui	Kyoto University	

Soutenue le 4/12/2014 à Marseille.

Résumé

On étudie dans cette thèse une structure de semi-anneau dont le produit est basé sur la règle de résolution de la programmation logique. Cet objet mathématique a été initialement introduit dans le but de modéliser la procédure d'élimination des coupures de la logique linéaire, dans le cadre du programme de *géométrie de l'interaction*. Il fournit un cadre algébrique et abstrait, tout en étant présenté sous une forme syntaxique et concrète, dans lequel mener une étude théorique du calcul.

On reviendra dans un premier temps sur l'interprétation interactive de la théorie de la démonstration dans ce semi-anneau, *via* l'axiomatisation catégorique de l'approche de la géométrie de l'interaction. Cette interprétation établit une traduction des programmes fonctionnels vers une forme très simple de programmes logiques.

Dans un deuxième temps, on abordera des problématiques de théorie de la complexité: bien que le problème de la nilpotence dans le semi-anneau étudié soit indécidable en général, on fera apparaître des restrictions qui permettent de caractériser le calcul en espace logarithmique (déterministe et non-déterministe) et en temps polynomial (déterministe).

Mots-clés: *Résolution, Programmation Logique, Complexité Implicite, Géométrie de l'Interaction, Catégories à Trace, Automates, Algèbre.*

Abstract

We study in this thesis a semiring structure with a product based on the resolution rule of logic programming. This mathematical object was introduced initially in the setting of the *geometry of interaction* program in order to model the `cut`-elimination procedure of linear logic. It provides us with an algebraic and abstract setting, while being presented in a syntactic and concrete way, in which a theoretical study of computation can be carried on.

We will review first the interactive interpretation of proof theory within this semiring *via* the categorical axiomatization of the geometry of interaction approach. This interpretation establishes a way to translate functional programs into a very simple form of logic programs.

Secondly, complexity theory problematics will be considered: while the nilpotency problem in the semiring we study is undecidable in general, it will appear that certain restrictions allow for characterizations of (deterministic and non-deterministic) logarithmic space and (deterministic) polynomial time computation.

Keywords: *Resolution, Logic Programming, Implicit Complexity, Geometry of Interaction, Traced Categories, Automata, Algebra.*

Remerciements

Mes premiers remerciements vont à Jean-Yves Girard, mon directeur de thèse, sans qui ce document n'aurait évidemment jamais existé. Il a su me laisser une grande liberté dans mon travail tout en me donnant de précieux conseils quand mon inspiration faisait défaut. La radicalité de son approche à la logique et à la science en général restera une inspiration, je l'espère, pour ma future vie de chercheur.

Je remercie également Ugo dal Lago et Philip Scott d'avoir accepté d'être rapporteurs de ce texte ainsi que tous les membres du jury, dont certains se sont déplacés de loin pour en faire partie.

Merci à Clément Aubert et Adrien Guatto[1] pour leur relecture de ce texte.

Je tiens à remercier les personnes avec qui j'ai eu l'occasion de collaborer, et d'écrire des articles pour certains, durant ces trois années: Alois Brunel et Damiano Mazza; Amina Doumane et Alexis Saurin; Clément Aubert, Paolo Pistone et Thomas Seiller. Il serait difficile de surestimer le rôle décisif que ces trois derniers ont joué dans une période ou mon travail n'avançait plus.

Merci également à Caudia Faggian pour l'énergie qu'elle a consacré au fonctionnement du projet LOGOI, au sein duquel j'ai évolué, appris et suis entré dans le monde de la recherche.

Merci aux permanents de l'équipe LDP qui m'on accueilli et accompagné pendant ma thèse: Emmanuel Beffara, Yves Lafont, Myriam Quatrini, Laurent Regnier et Lionel Vaux.

Merci aux thésards (passés, présents et assimilés) de l'IML qui, non contents de m'avoir enseigné l'art de la Belote contrée et d'avoir été des coéquipiers de football mémorables, on fait que les longues journées dans les murs du TPR2 soient un peu plus agréables: Anna, Danilo, Eugenia, Émilie, Étienne, Florent, Florian, Francesca, Hamish, Irene, Joël, Jordi, Julia, Lionel, Marc, Marcelo, Mathias, Matteo, Michele, Paolo, Pierro, Sarah, Stéphanie, Virgile... à cette liste il faut bien sûr ajouter Jean-Baptiste, avec qui j'ai eu le plaisir de partager un bureau pendant trois ans.

Merci aux membres et assimilés, anciens et nouveaux, du groupe du Groupe de Travail Logique de l'ENS avec qui j'ai beaucoup appris, beaucoup discuté, beaucoup ri (comme quoi la logique peut parfois être *fun*): Alois Brunel,

[1] Cela vaut bien une *footnote*, pour ta collection.

Adrien Guatto, Charles Grellois, Marie Kerjean, Guillaume Munch-Maccagnoni, Pierre-Marie Pédrot, Maël Pégny, Luc Pellissier, Pablo Rauzy, Silvain Rideau, Gabriel Scherer, Anne-Sophie de Suzzoni, et bien d'autres.

J'ai également une pensée toute particulière pour ceux qui m'ont hébergé durant mes nombreux passages à Paris, au premiers rangs desquels les colocataires de Corentin Cariou et Marcel Sembat, dont la chaleur de l'accueil et le confort du canapé ne sont plus à démontrer.

Merci à mes amis, anciens et moins anciens, qui n'ont rien à voir avec le monde de la logique ou le campus de Luminy: Adrien, Alex, Alice, Amélie, Anne-Camille, Arthur, Benoît, Benoît, Bonnie, Camille, Chloé, Christophe, Clément, David, Delphine, Édouard, Éric, Flora, Hugo, Hugo, Jean-Cyril, Judith, Julien, Laélia, Laurie, Marc, Marie, Matthieu, Michaël, Micka, Nicolas, Paul, Pauline, Péva, Rémi, Rémi, Renaud, Robin, Sébastien, Sébastien, Sihem, Vincent. Bien que vous n'en lirez probablement jamais une ligne, cette thèse vous doit beaucoup.

Salut à mes camarades de la lutte pour la titularisation des précaires de l'ENS et à ceux de SUD Étudiant Aix-Marseille. C'est à regret que je réalise à quel point je me suis éloigné de la plupart d'entre vous en me rapprochant du monde de la recherche.

Merci à toute ma famille pour son soutien sans faille, et ce malgré mon mauvais caractère.

Merci à Eugenia.

... à Caroline

Contents

Introduction

Unification, the theory of formal solving of equations, is a fundamental subject of study in computer science and its wide application range include type inference algorithms as well as automated theorem proving. Indeed, the resolution technique introduced by J. A. Robinson [Rob65] is a core component of logic programming.

We study in this thesis an algebraic structure with a product based on the resolution rule: a restricted class of logic programs, which we call *wirings*, happens to enjoy a semiring structure, which enables the use of tools and vocabulary of abstract algebra for reasoning about logic and computer science.

The origins of this semiring are rooted in the geometry of interaction (GoI) research program [Gir89b], which aims at giving a semantical account of the dynamics of proofs and programs, while abstracting away from proof systems. The *resolution algebra* was first introduced by J.-Y. Girard [Gir95a] to build a concrete interpretation of linear logic in this perspective.

The variant we consider is a semiring instead of an algebra, obtained by avoiding the introduction of complex coefficients and considering a discrete structure. Our goal will be to explore how this hybrid mathematical object can be used to study logic and computation, with a focus on complexity issues.

We will see through the categorical axiomatization of the GoI program that any term of the pure λ-CALCULUS can be represented as an element of the resolution semiring. In that respect, the syntactical and concrete nature of the semiring, when compared with the operator algebras used in the original approach [Gir89a, Gir90], yields some advantages: first, the construction should be more accessible to computer scientists as it does not require a background in functional analysis; second, this establishes a link between GoI, proof theory and logic programming; third, the study of computation, and in particular its space and time complexity, is more natural in this setting.

The study of complexity theory we will carry on in this semiring is motivated by the following idea: as the dynamics of proofs and programs can be modeled by the GoI approach, and more specifically in the resolution semiring, then it should be possible and fruitful to give an account of complexity theory in this setting, especially in view of the work relating linear logic and implicit computational complexity [DL12, Bai08] we have seen in the last two decades. Moreover, the complexity of the unification problem is a very well studied subject: it is known to be Ptime-complete in general, and some subcases of interest are known to belong to smaller classes.

We will set up tools to achieve some first results in this direction and we will see that it is indeed possible to obtain characterizations of standard complexity classes such as (N)Logspace [AB14, ABPS14] and Ptime, the main contribution of this thesis.

Outline of the thesis

The first chapter contains background material on the origins of the resolution algebra, unification theory and traced monoidal categories.

The second chapter is a general study of the properties of the resolution semiring and some restricted semirings. We will start by setting the basic definitions, notations and constructions, then the last two sections will detail the specific properties of the two restricted semirings that will be at work in the characterizations of complexity classes.

In the third chapter we show that the resolution semiring provides a setting where one can model the dynamics of pure λ-calculus, by proving that it satisfies the categorical axiomatization of the geometry of interaction introduced in chapter I. The chapter ends with a brief discussion of the link this establishes between logic programming and proof theory.

The last chapter is devoted to complexity theory. We set up the general framework, in particular the representation of inputs, in which we will characterize complexity classes. The results of chapter II are then put to work in order to obtain characterizations of logarithmic space and polynomial time computation.

Notations of complexity classes

Logspace	Languages recognizable in logarithmic space by a deterministic Turing machine.
NLogspace	The non-deterministic variant of the class above.
coNLogspace	Languages with their complementary in NLogspace.
FLogspace	Functions computable in logarithmic space by a deterministic Turing machine.
Ptime	Languages recognizable in polynomial time by a deterministic Turing machine.
FPtime	Functions computable in polynomial time by a deterministic Turing machine.
NC	Languages recognizable in polylogarithmic time on a parallel computer with a polynomial number of processors.

Chapter I

Background

The purpose of this chapter is to set some elements of technical and historical background that will hopefully ease the reading of the thesis. It also provides some further references.

We begin by retracing the origins of the resolution semiring. We will see that the want for a fine-grained analysis of the dynamics of logic, after the work of Gentzen, lead J.-Y. Girard to introduce linear logic and the geometry of interaction (GoI) research program [Gir89b]. This lead in turn to the introduction of an algebra with a product based on the resolution rule, in order to be able to manipulate by means of finite mathematical objects the potential infinity at work in this dynamics.

Unification theory [Kni89] is a cornerstone of both theoretical and practical computer science, and is at work in the resolution rule our semiring is based on. In the second section, we provide the reader with a brief reminder of the subject, settling notations and reviewing some significant complexity results.

We present in the last section the framework of traced categories and unique decomposition categories [HS06], the result of the work of various authors aimed at giving a categorical axiomatization of the GoI interpretation of proof theory. We will use this framework in chapter III to establish that the pure λ-calculus can be interpreted in the resolution semiring without needing to prove from scratches the crucial properties of this interpretation.

Contents

I.1 From sequent calculus to the resolution algebra

Proof theory is the domain of logic which focuses on the study of *proofs* and their formation, rewriting, equivalences, rather than provability or validity of *propositions*. In this perspective, the way proofs are built, described and manipulated becomes of great importance. We draw in this section a short history of the line of ideas that lead to the introduction of an algebra based on the resolution rule as a tool to describe proofs.

Sequent calculus

Following the work of Gentzen on logical formalisms [Gen34b] and the consistency of arithmetics [Gen34a], two tools became of widespread use in proof theory: natural deduction and sequent calculus. These emphasize on the geometrical structure of proofs, seen as trees which nodes are rules of the logic under consideration.

More precisely, a *sequent* is an expression of the form

$$\mathbf{S} = \ H_1, \dots, H_n \vdash C_1, \dots, C_m$$

that is, two multisets of formulas, the *hypothesis* and the *conclusions* of the sequent **S**. The informal meaning of a sequent would be "under the hypotheses $H_1, \dots, H_n$ at least one of the $C_1, \dots, C_m$ holds".

A *rule* of sequent calculus is an expression of the form

$$\frac{\mathbf{P}_1 \quad \cdots \quad \mathbf{P}_n}{\mathbf{C}}\ \texttt{R}$$

where $\mathbf{P}_1, \dots, \mathbf{P}_n$ and $\mathbf{C}$ are sequents, the *premises* and the *conclusion* of the rule, respectively. A *prooftree* is then a tree labeled by sequents, built by applying rules, the *conclusion* of the proof is the sequent that is the conclusion of its downmost rule.

Among rules of sequent calculus for, say, classical logic, the `cut` rule

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \texttt{cut}$$

plays a specific role: it enables deductive reasoning; in operational terms, the possibility to *compose* proofs. Indeed a particular case of the rule

$$\frac{A \vdash B \qquad B \vdash C}{A \vdash C}\ \texttt{cut}$$

simply says that from $A \Rightarrow B$ and $B \Rightarrow C$, one can deduce $A \Rightarrow C$.

One of the main results of Gentzen was that in certain systems, among which the sequent calculus for classical logic (**LK**), the `cut` rule is redundant.

Theorem I.1 (*Hauptsatz* [Gen34b, § 2.5])

Any proof π of **LK** can be rewritten in a `cut`-free proof π' with the same conclusions.

This result has important consequences on the logical system, as `cut`-free proofs are much easier to manipulate and study due to a number of structuring properties they enjoy, such as the *subformula property*[1] in the case of **LK**.

The consistency of **LK** immediately derives from this theorem.

Linear logic

The results of Gentzen initiated a shift in the focus of proof theory, towards a more operational point of view. The starting point is the remark that the rewriting of proofs introduced in the `cut`-elimination theorem can be regarded as a computation system.

From this perspective, a proof of $A \vdash B$ can be seen as a program that inputs (through the `cut` rule) a proof of A and outputs (*via* the `cut`-elimination procedure) a `cut`-free proof of B. This point of view is known as the Curry-Howard correspondence [Gal95], or the proofs-as-programs/propositions-as-types interpretation.

One of the products of this point of view on logic is *linear logic*, a system introduced by J.-Y. Girard [Gir87a] in order to carry on a finer analysis of the `cut`-elimination procedure.

Among other things, linear logic makes apparent the distinction between data that can or cannot be copied and erased *via* its *exponential modalities* **!**, **?** and retains the symmetry of classical logic: the linear negation $(\cdot)^\perp$ is an involutive operation, which allows for a one-sided sequent presentation. In this setting, the `cut` rule becomes

$$\dfrac{\vdash A,\Gamma \qquad \vdash A^\perp,\Delta}{\vdash \Gamma,\Delta}\ \texttt{cut}$$

As a case study, let us give the rules of two dual connectives of linear logic:

$$\dfrac{\vdash A,\Gamma \qquad \vdash B,\Delta}{\vdash A\otimes B,\Gamma,\Delta}\ \otimes \qquad\qquad \dfrac{\vdash A,B,\Gamma}{\vdash A\parr B,\Gamma}\ \parr$$

and an associated elimination step (the π_i denote a proof tree with the corresponding sequent as a conclusion):

$$\dfrac{\dfrac{\overset{\vdots\,\pi_1}{\vdash A^\perp,B^\perp,\Omega}}{\vdash A^\perp\parr B^\perp,\Omega}\ \parr \qquad \dfrac{\overset{\vdots\,\pi_2}{\vdash A,\Gamma} \qquad \overset{\vdots\,\pi_3}{\vdash B,\Delta}}{\vdash A\otimes B,\Gamma,\Delta}\ \otimes}{\vdash \Omega,\Gamma,\Delta,}\ \texttt{cut} \quad\rightarrow\quad \dfrac{\dfrac{\overset{\vdots\,\pi_1}{\vdash A^\perp,B^\perp,\Omega} \qquad \overset{\vdots\,\pi_2}{\vdash A,\Gamma}}{\vdash B^\perp,\Omega,\Gamma}\ \texttt{cut} \qquad \overset{\vdots\,\pi_3}{\vdash B,\Delta}}{\vdash \Omega,\Gamma,\Delta}\ \texttt{cut}$$

[1] Any formula that occur in a `cut`-free proof is a subformula of the conclusions of the proof.

This example is typical of the principle behind the cut-elimination procedure: transform a cut on a pair of formulas into “simpler” cut rules on their subformulas, thus lifting up the cut rules of the proof until its leaves.

But then, defining and studying the full cut-elimination procedure in the context of sequent calculus, one stumbles upon a difficulty: it may happen that the configuration described above does not occur because the cut does not follow two dual rules, for instance in the situation

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash A^{\perp}, B^{\perp}, C, D, \Omega\end{array}}{\vdash A^{\perp}\parr B^{\perp}, C, D, \Omega}\,\parr}{\vdash A^{\perp}\parr B^{\perp}, C\parr D, \Omega}\,\parr \qquad \dfrac{\begin{array}{c}\vdots\\ \vdash A, \Gamma\end{array} \quad \begin{array}{c}\vdots\\ \vdash B, \Delta\end{array}}{\vdash A\otimes B, \Gamma, \Delta}\,\otimes}{\vdash C\parr D, \Omega, \Gamma, \Delta}\,\texttt{cut}$$

the elimination step is not immediately possible and one has to *commute* the cut rule and the ⅋ rule to recover a situation where the elimination step applies:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash A^{\perp}, B^{\perp}, C, D, \Omega\end{array}}{\vdash A^{\perp}\parr B^{\perp}, C, D, \Omega}\,\parr \qquad \dfrac{\begin{array}{c}\vdots\\ \vdash A, \Gamma\end{array} \quad \begin{array}{c}\vdots\\ \vdash B, \Delta\end{array}}{\vdash A\otimes B, \Gamma, \Delta}\,\otimes}{\vdash C, D, \Omega, \Gamma, \Delta}\,\texttt{cut}}{\vdash C\parr D, \Omega, \Gamma, \Delta}\,\parr$$

These commutation situations make the study of the cut elimination procedure much more complex, as one needs to work *modulo* commutations, an equivalence on proofs that is not orientable into a rewriting procedure in an obvious way. Consider for instance the commutation:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \vdash A^{\perp}, B^{\perp}, \Gamma\end{array} \quad \begin{array}{c}\vdots\,\pi_2\\ \vdash A, \Delta\end{array}}{\vdash B^{\perp}, \Gamma, \Delta}\,\texttt{cut} \quad \begin{array}{c}\vdots\,\pi_3\\ \vdash B, \Omega\end{array}}{\vdash \Gamma, \Delta, \Omega}\,\texttt{cut} \quad \leftrightarrow \quad \dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \vdash A^{\perp}, B^{\perp}, \Gamma\end{array} \quad \begin{array}{c}\vdots\,\pi_3\\ \vdash B, \Omega\end{array}}{\vdash A^{\perp}, \Gamma, \Omega}\,\texttt{cut} \quad \begin{array}{c}\vdots\,\pi_2\\ \vdash A, \Delta\end{array}}{\vdash \Gamma, \Omega, \Delta}\,\texttt{cut}$$

Here it is not possible to favor a side of this equivalence without further non-local knowledge of the proof.

Proofnets and the Geometry of Interaction approach

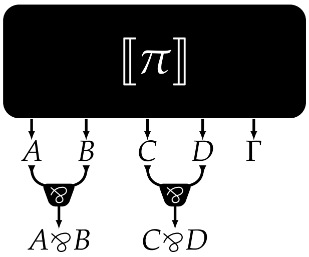
It appears after the discussion above that the study of the cut-elimination procedure in sequent calculus is a delicate matter. The point is that, as a language for describing proofs, sequent calculus is somewhat *too explicit*. For instance, the fact the the two following proofs

$$\cfrac{\cfrac{\overset{\vdots\ \pi}{\vdash A, B, C, D, \Gamma}}{\vdash A \parr B, C, D, \Gamma}\ \parr}{\vdash A \parr B, C \parr D, \Gamma}\ \parr \qquad \text{and} \qquad \cfrac{\cfrac{\overset{\vdots\ \pi}{\vdash A, B, C, D, \Gamma}}{\vdash A, B, C \parr D, \Gamma}\ \parr}{\vdash A \parr B, C \parr D, \Gamma}\ \parr$$

are different objects from the point of view of sequent calculus generates the first commutation situation we saw above.

A solution to this issue is too look for more intrinsic description of proofs, to find a language that is more *synthetic*; if possible to the point where commutation situations disappear.

Introduced at the same time as linear logic, the theory of *proofnets* [Gir87a, Gir96] partially addresses this issue. The basic idea is to describe proofs as graphs rather than trees, where application of logical rules become local graph construction, thus erasing some inessential sequential information. Indeed, the two proofs above translate into the same proofnet:



(where $[\![\pi]\!]$ is the proofnet translation of the rest of the proof) and the corresponding commutation situation disappear.

For the multiplicative fragment of linear logic, proofnets yield an entirely satisfactory solution to the problem of proof description through a language that rely only on local operations. This is no longer true with wider fragments of linear logic, indeed as soon as erasing and copying are involved.

The geometry of interaction research program [Gir89b] aims at extending the situation of multiplicative linear logic to wider fragments, implementing them by local operations. The first step in that direction was to give an algebraic account of proofnets, in terms of finite permutations [Gir87b].

But to be able to implement the *exponential* connectives of linear logic, that allow for an unbounded reuse of the same resource, an infinite space becomes necessary. This lead to constructions formulated in terms of $\mathbb{C}^*$-algebras [Gir89a, Gir95a] and von Neumann algebras [Gir06, Gir11].

The resolution algebra

In this picture, the *resolution algebra* holds a rather in-between position, being both an object of algebraic and syntactical nature.

The basic idea is to build an algebra (in this thesis: a semiring) whose composition law is the *resolution rule*, a key component of logic programming. This allows for a finite description of potentially infinite sets involved in the cut-elimination procedure.

An interpretation of full linear logic has been given by Girard within this algebra [Gir95a] and investigations in terms of complexity were also carried on by P. Baillot and M. Pedicini [BP01]. The syntactical nature of this algebra is at work in the complexity studies, as one can naturally speak about the size of objects involved or the cost of performing the operations of the algebra.

I.2 Unification

Unification can be generally thought of as the theory of formal solving of equations. The topic was introduced originally by Herbrand, but became really widespread after the work of J. A. Robinson [Rob65] on automated theorem proving. The unification theory is also at the core of logic programming and type inference for functional programming languages.

This section provides a brief technical reminder of the subject.

Unification, matching and Most General Unifiers

Specifically, unification theory is concerned with the following problem:

Does the equation $t \doteq u$ have a formal solution?

Let us first settle some notations and basic concepts about terms.

Notation (terms)

We consider first-order terms which we write $t, u, v, \ldots$ seen as labeled trees, built with variables (that can therefore occur only at leaves) and function symbols which have an assigned finite arity. The symbols of arity 0 will be called *constants*.

We assume the set of variables and the sets of function symbols of each arity are infinite. We will write variables as $x, y, z, x_i, \ldots$ (in italics font) and function symbols as $\mathtt{c}, \mathtt{f}(\cdot), \mathtt{g}(\cdot,\cdot), \ldots$ (in typewriter font).

We distinguish a binary function symbol $\bullet$ (written in *infix notation*) and a constant symbol $\star$.

The symbol $\bullet$ is not associative. However, we will write it as *right associating* by convention, to lighten notations, that is $t \bullet u \bullet v := t \bullet (u \bullet v)$.

We write $\mathtt{var}(t)$ the set of variables occurring in the term t, and we say that t is *closed* when $\mathtt{var}(t) = \varnothing$. The *height* $\mathtt{h}(t)$ of a term t is the maximal distance from the root to any leaf, and likewise the height of a variable occurrence is its distance from root.

The variable replacements are handled by means of the standard notion of substitution.

Definition I.2 (substitution)

A *substitution* is a map θ from variables to terms such that $\theta x = x$ for all but finitely many x. We will write $\{ x_1 \mapsto t_1, \dots, x_n \mapsto t_n \}$ the substitution θ such that $\theta x_i = t_i$ for all i, and that is the identity on the other variables.

Substitutions act on terms and can be composed (we write the composition of substitutions omitting the $\circ$ symbol) the usual way, so that $\theta(\psi t) = (\theta\psi)t$.

We say that a substitution ψ is an *instance* of θ if there exists a substitution σ such that $\psi = \sigma\theta$.

Renamings form a subclass of substitutions that only change the names of the variables.

Definition I.3 (renaming)

A *renaming* is a substitution that maps variables to variables and is bijective.

A term t is a *renaming* of u if there is some renaming α such that $t = \alpha u$.

Two substitutions θ, ψ are *equal up to renaming* if there is a renaming α such that $\theta = \alpha\psi$.

Remark I.4. If θ is an instance of ψ and ψ is an instance of θ, then they are equal up to renaming.

Let us now introduce the vocabulary for various situations where terms can be equated or not by substitutions.

Definition I.5 (unification, matching)

The pair of terms $t \doteq u$ is:

- *Unifiable* when there is a substitution θ such that $\theta t = \theta u$. In that case, θ is called a *unifier* of $t \doteq u$. If moreover any other unifier of $t \doteq u$ is an instance of θ, then it is called a *most general unifier* (MGU) of $t \doteq u$.
- *Matchable* if $t' \doteq u'$ is unifiable, where t', u' are two respective renamings of t, u such that $\mathtt{var}(t') \cap \mathtt{var}(u') = \varnothing$.
- *Disjoint* when it is not matchable.

Remark I.6. As a consequence of remark I.4, two MGUs of a pair of terms must be equal up to renaming.

Example I.7.

$\mathtt{f}(x) \doteq \mathtt{f}(\mathtt{c})$ is unified by $\{x \mapsto \mathtt{c}\}$.

$\mathtt{g}(\mathtt{c}, x) = \mathtt{h}(\mathtt{c}, x)$ is not unifiable.

$x \bullet v = u \bullet y$ (where u, v can be any terms) is unified by $\{x \mapsto u\,,\, y \mapsto v\}$.

$x = \mathtt{f}(x)$ is not unifiable, but it is matchable by first renaming $\mathtt{f}(x)$ as $\mathtt{f}(y)$.

$\mathtt{f}(x) = \mathtt{f}(\mathtt{c} \bullet y)$ is unified by both $\{x \mapsto \mathtt{c} \bullet \mathtt{f}(\mathtt{c})\,,\, y \mapsto \mathtt{f}(\mathtt{c})\}$ (which is not a MGU) and $\{x \mapsto \mathtt{c} \bullet z\,,\, y \mapsto z\}$ (which is a MGU).

The crucial result about unification of first order terms is the existence of MGUs, and the possibility to effectively compute them.

Theorem I.8 (MGU)

If two terms are unifiable, they have a MGU.

Whether two terms are unifiable and, in case they are, finding a MGU is a decidable problem.

Unification procedure

We give in this section a naive unification procedure, proving theorem I.8, which will be useful in section II.3 to prove that certain properties are preserved by unification simply remarking that they are preserved by the basic steps of the procedure. We follow the presentation of A. Martelli and U. Montanari [MM82], the reader can consult the original article for more detailed proofs.

To allow for an easier manipulation, the problem of unifying two terms needs to be generalized into the problem of simultaneously unifying several pairs of terms.

Definition I.9 (unification problem)

A *unification problem* is a finite set of equations $P = \{\, t_1 = u_1\,, \ldots\,, t_n = u_n \,\}$. It is in *solved form* if $P = \{\, x_1 = t_1\,, \ldots\,, x_n = t_n \,\}$ with the x_i pairwise distinct variables.

A *unifier* of P is a substitution θ such that $\theta t_i = \theta u_i$ for all i, it is a *most general unifier* of P if any other unifier of P is an instance of θ.

Two unification problems are *equivalent* if they have the same unifiers.

We first notice that a unification problem $P = \{\, x_1 = t_1\,, \ldots\,, x_n = t_n \,\}$ in solved form has an obvious MGU $\{\, x_1 \mapsto t_1\,, \ldots\,, x_n \mapsto t_n \,\}$.

The following two operations allow to simplify a unification problem into another equivalent one.

Notation

If $P = \{\, t_1 = u_1\,, \ldots\,, t_n = u_n \,\}$ is a unification problem and θ a substitution, we write $\theta P = \{\, \theta t_1 = \theta u_1\,, \ldots\,, \theta t_n = \theta u_n \,\}$

Lemma I.10 (variable reduction)

If $e \in P$ is $x = t$ or $t = x$, let $P' := P \setminus \{e\}$. We have either:

- $t = x$, then P is equivalent to P'.
- $t \neq x$ and $x \in \mathtt{var}(t)$, then P is not unifiable.
- $t \neq x$ and $x \notin \mathtt{var}(t)$, then P is equivalent to $\{e\} \cup \big(\{x \mapsto t\}P'\big)$.

Lemma I.11 (function reduction)

If $e = \mathtt{p}(u_1, \dots, u_n) = \mathtt{q}(t_1, \dots, t_k) \in P$, let $P' := P \setminus \{e\}$. We have either:

- $p \neq q$, then P has no unifier.
- $p = q$, then $n = k$ and P is equivalent to $P' \cup \{\, u_1 = t_1 \,,\, \dots \,,\, u_n = t_n \,\}$.

These two lemmas can be combined to design the following algorithm, that inputs a unification problem P and determines if it is unifiable and outputs an equivalent problem S in solved form in that case:

```
$S := \varnothing$
while $P \neq \varnothing$ do
  pick an equation $e \in P$
  if $e$ is $x = x$ for some variable $x$ then
    $P := P \setminus \{e\}$
  else if $e$ is $t = x$ or $x = t$ for some variable $x$ then
    if $x$ occurs in $t$ then
      return non-unifiable: cyclic
    else
      $P := P \setminus \{e\}$
      $P := \{x \mapsto t\}P$
      $S := \{x \mapsto t\}S$
      $S := S \cup \{x = t\}$
    end if
  else if $e$ is $\mathtt{p}(u_1, \dots, u_n) = \mathtt{p}(t_1, \dots, t_n)$ then
    $P := P \setminus \{e\}$
    $P := P \cup \{\, u_1 = t_1 \,,\, \dots \,,\, u_n = t_n \,\}$
  else
    return non-unifiable: symbol clash
  end if
end while
return $S$
```

Proof (of theorem I.8) ▸ The correctness of the algorithm follows from the two lemmas: after any iteration of the **while** loop, $P \cup S$ is equivalent to the original problem, and S is in solved form. The termination follows from a lexicographical well-ordering of P by: number of distinct variables, total number of function symbols and number of equations. ◂

We just saw that we can determine if a unification problem has a solution. We note the following fact, that will imply the associativity property of the resolution semiring: solving a unification problem can be done incrementally.

Lemma I.12

Let $P = Q \cup R$ be a unification problem. The following statements are equivalent:

- P is unifiable.
- Q has a MGU θ and θR has a MGU ψ.

In that case, we have moreover that $\psi\theta$ is a MGU of P.

Indeed, the algorithm in the above proof is non-deterministic in its choice of a pair to treat at line 3. But, as the induction measure strictly decreases no matter what choice is made, one can force a particular strategy. We can therefore have the procedure always selecting an equation from Q (or one of its descendant), so that the problem Q will be solved first, yielding the MGU θ if it is unifiable and leaving the variable P containing θR.

Complexity

In his article introducing the resolution method, Robinson gave a procedure to compute the MGU of two terms if it exists, basically the naive one we presented above.

This procedure is however quite inefficient, with potential exponential blowups in some cases. This can easily be seen by considering the following unification problem:

$$P := \{\, x_1 = x_2 \bullet x_2\,,\; x_2 = x_3 \bullet x_3\,,\; \dots\,,\; x_n = x_{n+1} \bullet x_{n+1} \,\}$$

which has a linear size in n, but with a MGU of exponential size in n. Hence one needs at least a more efficient representation of substitutions to improve the algorithm.

It turns out that the general unification problem can be solved in linear time [PW78, MM82]. A lower bound was also established, ruling out the potentiality for efficient parallel unification algorithms: the problem is Ptime-complete [DKM84], and remains so even under all sorts of restrictions: bounded arity of function symbols or height of the terms [OYY87, Theorems 4.2.1 and 4.3.1], linearity or absence of shared variables [DKM84, DKS88].

More recently, a constraint on variables helped to discover an upper bound of the unification classes that are proven to be in NC [BO03].

Regarding space complexity, we will be using the result stating that the *closed matching* problem is in Logspace [DKM84] which we recall here.

Theorem I.13 (closed matching is in Logspace [DKM84, p. 49])

Given two terms t and u such that either t or u is closed, deciding if they are unifiable, and if so finding a MGU, can be done within logarithmic space.

The resolution rule

We expose briefly one of the key ideas of logic programming: the *resolution rule*, which gives its name to the semiring we introduce in the next chapter.

To make a long story short, a *logic program* is defined as a set of *Horn clauses*, which are written as (reverted) sequents:

$$H \dashv B_1, \dots, B_n$$

where the formulas $H, B_1, \dots, B_n$ are *atoms* of the form $\mathbf{u}(t_1, \dots, t_k)$, the $t_1, \dots, t_k$ being terms as above and $\mathbf{u}$ being a predicate symbol. The formula H is called the *head* of the clause, while the $B_1, \dots, B_n$ constitute its *body*. The clause is said to be *safe* if $\mathtt{var}(H) \subseteq \mathtt{var}(B_1, \dots, B_n)$. A safe clause with an empty body is called a *fact*, while a clause with no head is called a *goal*.

The notion of unifiability obviously extends to such atoms, which allows to formulate the following rule:

$$\frac{H \dashv B_1, \dots, B_n, U \qquad V \dashv T_1, \dots, T_n \qquad \theta \text{ is a MGU of } U, V}{\theta H \dashv \theta B_1, \dots, \theta B_n, \theta T_1, \dots, \theta T_n}\ \mathtt{Res}$$

This rule generalizes in the case of Horn clauses the cut rule we presented in the previous section which allows for a composition of implication that incorporates the unification mechanism.

Then, a technique used to look for a derivation of a goal $\dashv G_1, \dots, G_n$ from a set of facts F and a logic program P consists in choosing a G_i and try to unify it with one of the heads of the Horn clauses in either P of F. If this succeeds, then the resolution rule can be applied and this leads to an updated goal. The procedure continues until the goal to derive is empty, backtracking if it has made the wrong choices. This is only a semi-decision procedure: depending on the strategy employed, it can enter loops (for instance unifying repeatedly the goal $\mathbf{u}(t)$ with the clause $\mathbf{u}(x) \dashv \mathbf{u}(x)$) with no way to decide when to stop. The problem is indeed only semi-decidable in general [DEGV01].

I.3 Traced categories and geometry of interaction

After the early work by Girard, various authors have been working on pinning down the basic requirements for a mathematical structure that allows for a GoI interpretation of logic, and more specifically of pure λ-calculus.

A first approach to this problem was the notion of *dynamic algebra* [Dan90].

Definition I.14 (dynamic algebra)

A *dynamic algebra* is a monoid M (with unit I) endowed with operations $(\cdot)^\dagger, !(\cdot) : M \leftarrow M$ and an absorbing element 0 satisfying

1. $(\cdot)^\dagger$ is an involution: for all x, $(x^\dagger)^\dagger = x$.
2. $!0 = 0^\dagger = 0$.
3. $!I = I^\dagger = I$.
4. For any x, $!(x^\dagger) = (!x)^\dagger$.
5. For any x, y, $!(xy) = !x!y$ and $(xy)^\dagger = y^\dagger x^\dagger$.

together with distinguished elements p, q, r, s, t, d satisfying

6. $pp^\dagger = qq^\dagger = rr^\dagger = ss^\dagger = dd^\dagger = tt^\dagger = I$.
7. $p^\dagger q = q^\dagger p = r^\dagger s = s^\dagger r = 0$.
8. For any x, $(!x)r = r(!x)$ and $(!x)s = s(!x)$.
9. For any x, $(!x)t = t(!!x)$ and $(!x)d = dx$.

However this setting turns out to be slightly too strict for our purposes. In particular we will not be able to satisfy the property $!I = I$ in the resolution semiring, but only a weaker variant of it.

More generally, some settings where a GoI construction can be carried on happen not to fit in the axioms of dynamic algebras, which led to further investigations towards a more flexible framework based on category theory.

The starting point in that direction was the original article on *traced monoidal categories* [JSV96], which already observed the connection between the notion of trace and Girard's *execution formula* [Gir89a]. This observation has been formalized in a series of papers [AJ94, Hag00b, AHS02, HS06] and investigated in depth in E. Haghverdi's thesis [Hag00a].

Traced categories

We review here the notion of traced monoidal category [JSV96] in a very specific case: namely the case where most of the notions are *strict* and the underlying monoidal category is *symmetric*, as we will be in this situation in section III.1 and it will spare us a lot of coherence diagrams checking. The general non-strict definitions can be found in the literature [ML71, Mel09].

Notation

We will abbreviate "f is a morphism from A to B" into $f : B \leftarrow A$. We use the same convention for functors and natural transformations.

We often omit the parenthesis when writing the application of a functor $\mathbf{F}$ to either an object or a morphism, which gives for instance $\mathbf{F}f : \mathbf{F}B \leftarrow \mathbf{F}A$.

Moreover, we omit the composition operator of morphisms: if we have $g : C \leftarrow B$ and $f : B \leftarrow A$, then $gf : C \leftarrow A$.

The basic setting is that of a *symmetric monoidal category*, in which one finds a very primitive notion of product, and swapping operations.

Definition I.15 (symmetric monoidal category)

A *monoidal category* is a category $\mathfrak{C}$ together with a bifunctor $\oplus : \mathfrak{C} \leftarrow \mathfrak{C} \times \mathfrak{C}$ and a distinguished object $\mathbf{0}$ satisfying:

- The functor $\oplus$ is associative: for any objects A, B, C of $\mathfrak{C}$, we have that $A \oplus (B \oplus C) = (A \oplus B) \oplus C$, which we then write $A \oplus B \oplus C$, and the same holds for morphisms.
- The object $\mathbf{0}$ is neutral for $\oplus$: for any object A, $A \oplus \mathbf{0} = \mathbf{0} \oplus A = A$, and $\mathrm{Id}_{\mathbf{0}} \oplus f = f \oplus \mathrm{Id}_{\mathbf{0}} = f$ for any f.

It is called a *symmetric* monoidal category if it moreover enjoys a natural family of isomorphisms $\sigma_{A,B} : B \oplus A \leftarrow A \oplus B$ (which we call *symmetries*) such that $\sigma^{-1}_{B,A} = \sigma_{A,B}$ and $\sigma_{A \oplus B, C} = (\sigma_{A,C} \oplus \mathrm{Id}_B)(\mathrm{Id}_A \oplus \sigma_{B,C})$.

Remark I.16. The most standard notation for the monoidal product would be "$\otimes$" but the choice of $\oplus$ fits better the algebraic intuition in the category we will study in chapter III: the monoidal product in section III.1 is similar to a *direct sum* in the language of linear algebra.

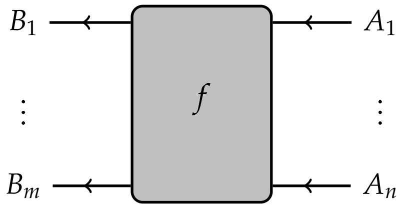
Example I.17. Classic example of (non-strict) symmetric monoidal categories include the category of sets equipped with either the cartesian product or the disjoint union and the category of vector spaces over a field equipped with the usual algebraic tensor product of vector spaces.

More generally, a category with finite products (resp. co-products) is always symmetric monoidal: the universal properties imply the required functoriality and the empty (co)product provides a terminal (resp. initial) object that can play the role of $\mathbf{0}$.

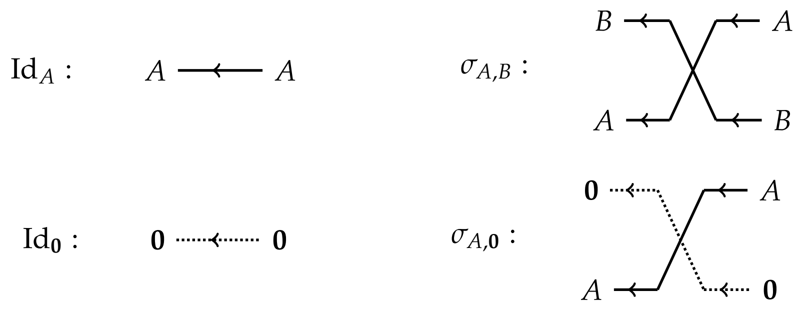
Symmetric monoidal categories enjoy nice presentations in terms of graphical language [Sel11]. This idea was introduced along the notion of trace and makes its presentation very natural.

Although the idea of a graphical language can be made totally rigorous, we will use it here only as a guide for intuition.

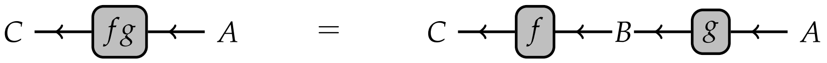
The principle is that a morphism $f : B_1 \oplus \cdots \oplus B_n \leftarrow A_1 \oplus \cdots \oplus A_m$ is depicted as a box



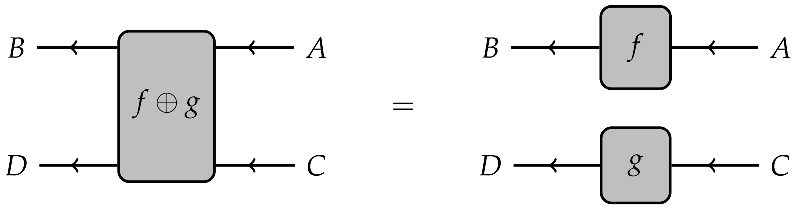
the identity morphisms and symmetries as wires (dotted when involving the object **0**)



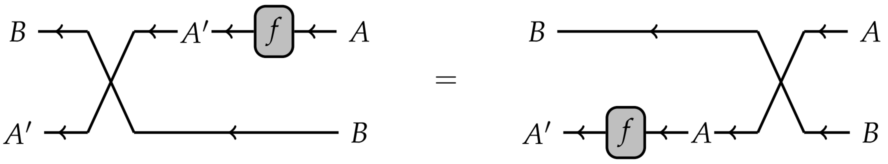
the composition is depicted as plugging boxes



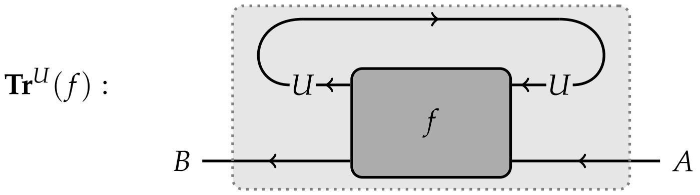
and the action of the bifunctor is depicted as putting boxes side by side



This notation allows in particular for an intuitive representation of the naturality of the family of symmetries:



With this language of circuits and wires in mind, it is easy to formulate the idea behind the notion of trace: one wants to make sense of the picture below



where a wire connects an output to an input of the morphism f, these interfaces becoming internal.

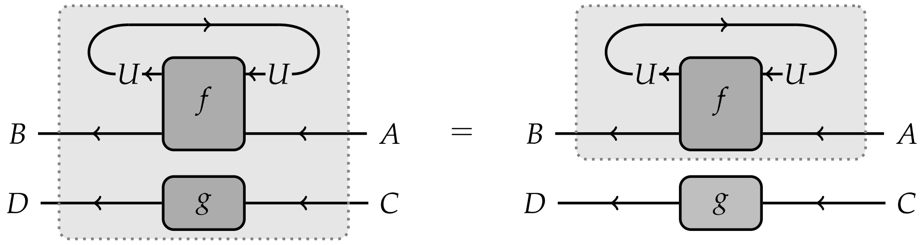
The concept of *trace* in a symmetric monoidal category thus axiomatizes categorically the notion of *feedback*: we may think of a flow of information that is indeed fed back from the output to the input.

Definition I.18 (trace)

A *trace* in a symmetric monoidal category is an operation on morphisms (parametrized by an object U) $\mathbf{Tr}^U(\cdot)$ such that if we start with a morphism $f : U \oplus B \leftarrow U \oplus A$, we obtain $\mathbf{Tr}^U(f) : B \leftarrow A$. It must moreover satisfy the following axioms (followed by their graphical counterparts):

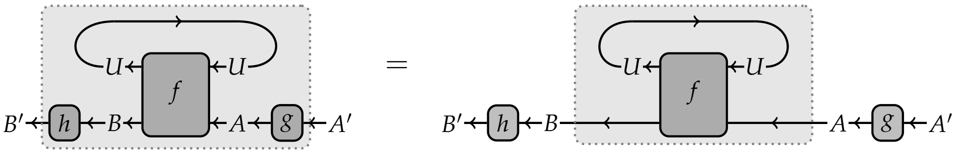
1. Superposing: for all $f : U \oplus B \leftarrow U \oplus A$ and $g : D \leftarrow C$, we have

$$\mathbf{Tr}^U(f \oplus g) = \mathbf{Tr}^U(f) \oplus g : B \oplus D \leftarrow A \oplus C$$



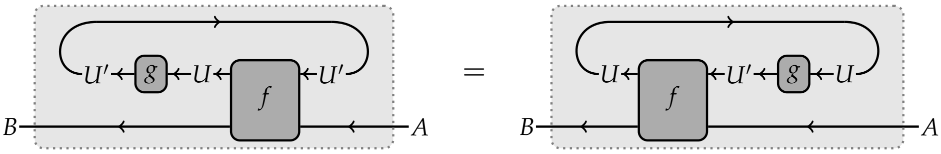
2. Tightening: for all $f : U \oplus B \leftarrow U \oplus A$, $g : A \leftarrow A'$ and $h : B' \leftarrow B$, we have

$$\mathbf{Tr}^U\big((\mathrm{Id}_U \oplus h) f (\mathrm{Id}_U \oplus g)\big) = (\mathrm{Id}_U \oplus h)\mathbf{Tr}^U(f)(\mathrm{Id}_U \oplus g) : B \leftarrow A$$



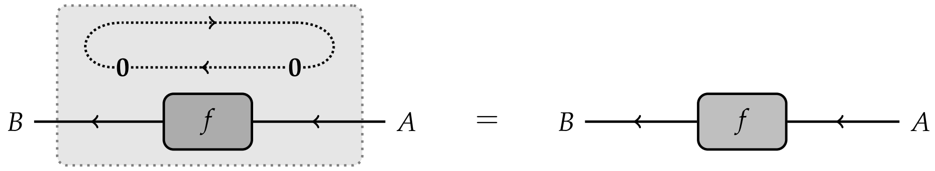
3. Sliding: for all $f : U \oplus B \leftarrow U' \oplus A$ and $g : U' \leftarrow U$, we have

$$\mathbf{Tr}^{U'}\big((g \oplus \mathrm{Id}_B) f\big) : B \leftarrow A = \mathbf{Tr}^U\big(f(g \oplus \mathrm{Id}_A)\big)$$



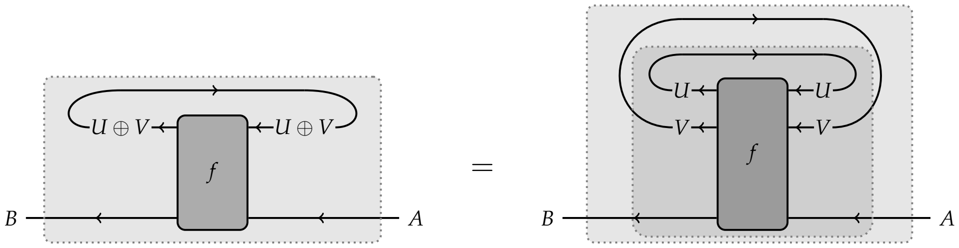
4. Vanishing: for all $f = \mathrm{Id}_{\mathbf{0}} \oplus f : \mathbf{0} \oplus B \leftarrow \mathbf{0} \oplus A$, we have

$$\mathbf{Tr}^{\mathbf{0}}(f) = f : B \leftarrow A$$



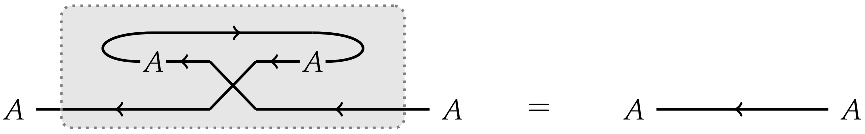
5. Associativity: for all $f : U \oplus V \oplus B \leftarrow U \oplus V \oplus A$, we have

$$\mathbf{Tr}^{U\oplus V}(f) = \mathbf{Tr}^{V}\big(\mathbf{Tr}^{U}(f)\big) \;:\; B \leftarrow A$$



6. Yanking: for any object A, we have

$$\mathbf{Tr}^{A}(\sigma_{A,A}) = \mathrm{Id}_A$$



A symmetric monoidal category equipped with a notion of trace will be called a *traced category*.

With these notions of categories come notions of functor and natural transformations that preserves their structure.

Definition I.19 (traced functor)

A *monoidal functor* $\mathbf{F} : \mathfrak{D} \leftarrow \mathfrak{C}$ between monoidal categories is a functor that satisfies for any A, B, f, g (we use superscripts to indicate in which category operations occur):

- $\mathbf{F}(\mathbf{0}^{\mathfrak{C}}) = \mathbf{0}^{\mathfrak{D}}$.
- $\mathbf{F}(A \oplus^{\mathfrak{C}} B) = \mathbf{F}A \oplus^{\mathfrak{D}} \mathbf{F}B$ and $\mathbf{F}(f \oplus^{\mathfrak{C}} g) = \mathbf{F}f \oplus^{\mathfrak{D}} \mathbf{F}g$.

when the categories are symmetric, we say that $\mathbf{F}$ is a *symmetric monoidal functor* if it moreover satisfies

- $\mathbf{F}(\sigma^{\mathfrak{C}}_{A,B}) = \sigma^{\mathfrak{D}}_{\mathbf{F}A,\mathbf{F}B}$.

and if the categories are traced, $\mathbf{F}$ is a *traced functor* if it moreover satisfies

- $\mathbf{F}\big(\mathbf{Tr}^{U}(f)\big) = \mathbf{Tr}^{\mathbf{F}U}(\mathbf{F}f)$ for any $f : U \oplus B \leftarrow U \oplus A$.

Definition I.20 (monoidal natural transformation)

A natural transformation α between monoidal functors is called *monoidal* whenever $\alpha_{A \oplus B} = \alpha_A \oplus \alpha_B$ and $\alpha_{\mathbf{0}} = \mathrm{Id}_{\mathbf{0}}$.

Again, we gave the notions in their *strict* variant that are simpler to formulate and which will be enough in our case.

GoI **situations**

We now have at hand an abstract way to make sense of the notion of feedback, and this is enough for building GoI models of multiplicative linear logic.

Still, if we want to interpret exponential modalities of linear logic or the pure λ-CALCULUS, where erasing and duplication may occur, some further structure will be required: this leads to GoI *situations* [HS06] which rely on the notion of *retraction*. The basic idea is that a retraction can be seen as a sort of embedding of an object into another, and this will be at work for instance to interpret the contraction rule: two copies of the same object $\mathbf{!}A \oplus \mathbf{!}A$ can be embedded in one $\mathbf{!}A$.

Definition I.21 (retraction)

A *retraction* between two objects A, B of a category is a pair of morphisms $f : A \leftarrow B$ and $g : B \leftarrow A$ such that $fg = \mathrm{Id}_A$. We abbreviate this as

$$(f, g) : A \lhd B$$

A *natural retraction* between two functors $\mathbf{F}$ and $\mathbf{G}$ is a pair of natural transformations $\alpha : \mathbf{F} \leftarrow \mathbf{G}$ and $\beta : \mathbf{G} \leftarrow \mathbf{F}$ such that for any object A, $(\alpha_A, \beta_A) : \mathbf{F}A \lhd \mathbf{G}A$. We abbreviate this as

$$(\alpha, \beta) : \mathbf{F} \lhd \mathbf{G}$$

If moreover the functors are monoidal and the natural transformations are monoidal, we say that we have a *monoidal retraction*.

Example I.22. In the category of sets and functions, the notion of retraction corresponds to the pairs (f, g) where g is an injective function and f is such that $f \circ g = \mathrm{Id}$.

In this category, consider the functor $\mathbf{F}_A := \cdot \times A$ that builds the cartesian product with some fixed set A and acts on morphisms as $\mathbf{F}_A f := f \times \mathrm{Id}_A$. Any retraction $(a, b) : A \lhd B$ induces a natural retraction $(\alpha, \beta) : \mathbf{F}_A \lhd \mathbf{F}_B$ defined as $\alpha_X := \mathrm{Id}_X \times a$ and $\beta_X := \mathrm{Id}_X \times b$.

Definition I.23 (GoI **situation)**

A GoI *situation* is a triple $(\mathfrak{C}, !, U)$, where

- $\mathfrak{C}$ is a traced category.
- $!$ is a traced functor with the following monoidal retractions:
  1. *Digging* retraction $(t, t') : \mathbf{!!} \lhd \mathbf{!}$
  2. *Contraction* retraction $(c, c') : \mathbf{!} \oplus \mathbf{!} \lhd \mathbf{!}$
  3. *Weakening* retraction $(w, w') : \mathbf{0} \lhd \mathbf{!}$
  4. *Dereliction* retraction $(d, d') : \mathrm{Id} \lhd \mathbf{!}$
- U is an object of $\mathfrak{C}$ with retractions
  1. $(a, a') : !U \lhd U$
  2. $(b, b') : U \oplus U \lhd U$
  3. $(c, c') : \mathbf{0} \lhd U$

In that case U is called the *reflexive object* of the triple.

These retraction pairs enable the interpretation of the structural rules (with the corresponding names) of linear logic, in charge of the non-linear manipulation of data. As the $\mathbf{Tr}(\cdot)$ operation corresponds to the cut rule in the GoI interpretation, the traced nature of $\mathbf{!}$ reflects the possibility to commute a promotion rule with a cut rule in the course of the cut-elimination procedure. Also, the naturality of these families of morphisms correspond to the other cut-elimination steps of exponential connectives. This becomes particularly apparent when drawing the naturality equations in the graphical language.

Remark I.24. As soon as a functor $\mathbf{!}$ is defined, the image $!X$ of any object is a potential reflexive object, the monoidal retractions giving the particular retractions for $!X$ automatically. In particular the choice of a reflexive object is not unique.

Remark I.25. The interpretation of λ-calculus in such a category only makes use of the structure we exposed so far. That is, the interpretation of a λ-term will always be a combination of the identity, symmetry and retraction morphisms and their images through the trace operation and the $\mathbf{!}$ and $\oplus$ functors. Moreover, the interpretation of the linear fragment of linear logic (or the linear λ-calculus) would make no use of the $\mathbf{!}$ functor and rely only on $(b, b') : U \oplus U \lhd U$ among the retractions.

The reader will find in the appendix a summary of the GoI interpretation of λ-calculus that can be carried on as soon as a GoI situation is available.

Unique decomposition categories

In order to have an axiomatization that is closer to Girard's approach, Haghverdi introduced *unique decomposition categories*, inspired from earlier categorical analysis of programming [MA86]. The purpose is to capture the notion of matrix in a categorical framework.

The interest for us is that it gives a straightforward way to show that a category is traced: as we shall see, in the variant we consider, unique decomposition categories are automatically traced. This will spare us in section III.1 the tedious one-by-one checking of the axioms of definition I.18.

Basically, unique decomposition categories are symmetric monoidal categories where one can take sums of morphisms and where composition distributes over sums, together with projection and injection morphisms associated to the $\oplus$ functor. In these categories morphisms between products of objects can be decomposed into components.

Definition I.26 ($\sum$-monoid)

A $\sum$-*monoid* is a set X together with a map $\sum$ that associates to any family $(A_i)_{i\in I}$ of elements of X an element $\sum_{i\in I} A_i$ (the *sum* of the family) of X satisfying:

- For any one element family (A_0), $\sum_{i\in\{0\}} A_i = A_0$.
- For any family $(A_i)_{i\in I}$ and any partition $(J_k)_{k\in K}$ of I,

$$\sum_{k\in K}\sum_{j\in J_k} A_j = \sum_{i\in I} A_i$$

The sum of the empty family will be written as 0.

It is easy to see that any powerset is an example of this notion.

Lemma I.27

For any set X, $\mathcal{P}(X)$ with union as a $\sum$ operator is a $\sum$-monoid.

Remark I.28. In general, $\sum$-monoids can have a partially defined sum operator, yielding a *partially traced category* [MSS12] through the full version of theorem I.32. Also, the families that are summed are usually required to be denumerable.

In our case (section III.1), the sums will always be defined and the denumerability will not matter (indeed we will be in the situation of the above lemma) so we chose to give a simpler variant here.

Definition I.29 (unique decomposition category)

A *unique decomposition category* (UDC) is a symmetric monoidal category where for any object A, B the set of morphisms from A to B has a structure of $\sum$-monoid, satisfying a distributivity property: for all $(f_i) : C \leftarrow B$ and $(g_j) : B \leftarrow A$ we have

$$\Big(\sum_i f_i\Big)\Big(\sum_j g_j\Big) = \sum_{i,j} f_i g_j$$

moreover, for any finite product $A_1 \oplus \cdots \oplus A_n$ there are morphisms (which are respectively called *injections* and *projections*)

$$\iota_i : A_1 \oplus \cdots \oplus A_n \leftarrow A_i \qquad \pi_i : A_i \leftarrow A_1 \oplus \cdots \oplus A_n$$

such that

$$\pi_i \iota_i = \mathrm{Id}_{A_i} \qquad \pi_i \iota_j = 0 \;\text{ if }\; i \neq j \qquad \sum_i \iota_i \pi_i = \mathrm{Id}_{A_1 \oplus \cdots \oplus A_n}$$

The theory of unique decomposition categories and the GoI interpretation in this setting is studied in details in Haghverdi's thesis [Hag00a]. They are related to traced categories through their decomposition property and the execution formula.

The decomposition property expresses the expected fact that morphisms of a UDC can be seen as matrices, and therefore decomposed into components. We give only the case of interest for the execution formula defined just after.

Proposition I.30 (decomposition property [HS06, proposition 5])

Any morphism $f : U \oplus A \leftarrow U \oplus B$ of a UDC can be decomposed as:

$$f = f_{B,A} + f_{U,A} + f_{B,U} + f_{U,U}$$

where $f_{Y,X} := \iota_Y f \pi_X : Y \leftarrow X$ for $Y \in \{U, B\}$ and $X \in \{U, A\}$.

The execution of a morphism is then defined by iterating its $f_{U,U}$ component, which fits the intuition behind the diagrams of trace we have seen before.

Definition I.31 (execution formula)

In a UDC, the *iteration* of a morphism $g : U \leftarrow U$ is defined as the sum of the family $(g^n)_{n\in\mathbb{N}}$, where g^n is the composition of n copies of g:

$$\mathbf{IT}(g) := \sum_{n\in\mathbb{N}} g^n$$

Given a morphism $f : U \oplus A \leftarrow U \oplus B$ its *execution* is defined as

$$\mathbf{EX}^U(f) := f_{B,A} + f_{B,U}\,\mathbf{IT}(f_{U,U})\, f_{U,A}$$

If $f_{U,U}$ is nilpotent,[2] we say that the execution of f with respect to U is *finite*.

[2] Anticipating on definition II.13: $f_{U,U}$ is nilpotent if there is an integer n such that $(f_{U,U})^n = 0$.

Then, we automatically get a traced category with the execution formula as a trace.

Theorem I.32 (execution and trace [HS06, proposition 6])

A UDC is a traced category with $\mathbf{EX}(\cdot)$ as a trace.

Remember (remark I.28) that this holds only for our restricted definition of UDC: in case not all sums are defined, the execution formula only yields a partial trace.

Moreover, in this situation, functors can be showed to be traced very easily: a sufficient condition is that they are *additive*, a version of linearity compatible with infinite sums.

Definition I.33 (additive functor)

A symmetric monoidal functor $\mathbf{F}$ between UDCs is called *additive* if for any family of morphisms $(f_i) : Y \leftarrow X$ we have

$$\mathbf{F}\big(\sum_i f_i\big) = \sum_i \mathbf{F} f_i$$

Lemma I.34 (traced and additive functors [Hag00a, lemma 8.1.1])

An additive functor between UDC is traced.

Chapter II

The Resolution Semiring

We turn now to the definition and study of the disclaimed resolution semiring, introduced by J.-Y. Girard [Gir95a] to build a GoI interpretation of linear logic. As we already said, this semiring will use the resolution rule (page 19) as its product, hence its name.

The two first sections define the semiring, settle some notations and review a few basic constructions that will be used throughout the thesis.

Then, in view of chapter IV and the complexity results, we introduce two specific restricted semirings: the balanced semiring $\mathcal{R}_b$ and the $\mathcal{S}tack$ semiring. We explore their algebraic properties related to the notion of nilpotency, which will be used as an acceptance condition when characterizing the complexity classes LOGSPACE, NLOGSPACE and PTIME.

Contents

II.1 Flows and wirings

Flows are very specific Horn clauses: safe (the variables of the head must occur in the body) clauses with exactly one atom in the body.

As it is not relevant for the moment, we make no difference between predicate symbols and function symbols, for it makes the presentation easier.

Definition II.1 (flow)

A *flow* is a pair f of terms, which we write $t \leftharpoondown u$, with $\mathtt{var}(t) \subseteq \mathtt{var}(u)$. If moreover $\mathtt{var}(t) = \mathtt{var}(u)$, we define the *adjoint* $f^\dagger := u \leftharpoondown t$ of f. Flows are considered up to renaming: for any renaming α, $t \leftharpoondown u = \alpha t \leftharpoondown \alpha u$.

Facts, that are usually defined as closed clauses with an empty body, can still be represented as a special kind of flows.

Definition II.2 (fact)

A *fact* is a flow of the form $t \leftharpoondown \star$ (remember $\star$ is a fixed constant symbol).

Remark II.3. Note that this implies that t is closed.

The main interest of the restriction to flows is that it yields an algebraic structure: a monoid with a partially defined product.

Definition II.4 (product of flows)

Let $t \leftharpoondown u$ and $v \leftharpoondown w$ be two flows. Suppose we picked representatives of the renaming classes such that $\mathtt{var}(u) \cap \mathtt{var}(v) = \varnothing$.

The *product* of $t \leftharpoondown u$ and $v \leftharpoondown w$ is defined, if $u \doteq v$ is unifiable with MGU θ, as $(t \leftharpoondown u)(v \leftharpoondown w) := \theta t \leftharpoondown \theta w$.

Remark II.5. The specific choice of a MGU does not matter because of remark I.6. Moreover, the product is associative, a consequence of lemma I.12.

Remark II.6. The condition on variables ensures that facts form a "left ideal" of the set of flows: if $\mathbf{u}$ is a fact and f a flow, then $f\mathbf{u}$ is a fact whenever it is defined.

The product of flows is the resolution rule: given two flows $t \leftharpoondown u$ and $v \leftharpoondown w$ with u and v matchable, seen as clauses $t \dashv u$ and $v \dashv w$, the resolution rule applied to $t \leftharpoondown u$ and $v \leftharpoondown w$ would yield the clause that is the result of the product $(t \leftharpoondown u)(v \leftharpoondown w)$.

Example II.7. Let us illustrate the above definitions with some computations:

$$\begin{aligned}
&\big(\mathtt{f}(x) \leftharpoonup x\big)\big(\mathtt{f}(y) \leftharpoonup \mathtt{g}(y)\big) = \mathtt{f}(\mathtt{f}(y)) \leftharpoonup \mathtt{g}(y)\\
&\big(\mathtt{c} \leftharpoonup (x \bullet x)\big)\big((\mathtt{c} \bullet y) \leftharpoonup \mathtt{f}(y)\big) = \mathtt{c} \leftharpoonup \mathtt{f}(\mathtt{c})\\
&\big(\mathtt{f}(x \bullet \mathtt{c}) \leftharpoonup x \bullet \mathtt{d}\big)\big(\mathtt{d} \bullet \mathtt{d} \leftharpoonup \star\big) = \mathtt{f}(\mathtt{d} \bullet \mathtt{c}) \leftharpoonup \star\\
&\big(\mathtt{f}(x) \leftharpoonup \mathtt{g}(x)\big)\big(\mathtt{f}(x) \leftharpoonup x \bullet \mathtt{c}\big) \text{ is undefined}
\end{aligned}$$

We will need to consider formal sums of flows or, in other words, a basic structure of semiring. The simplest way to obtain this is to consider sets of flows, which we call *wirings*. Wirings therefore correspond to logic programs.

Definition II.8 (wiring)

Wirings are (possibly infinite) sets of flows. The product of wirings is defined as
$$FG := \{\, fg \mid f \in F,\ g \in G,\ fg \text{ defined} \,\}$$
We write $\mathcal{R}$ the set of wirings and refer to it as the *resolution semiring*.

Note that the product (of wirings) is then a total operation, as we added the empty set as a representative of the failure of unification: we have for instance $\{\, \mathtt{f}(x) \leftharpoonup \mathtt{g}(x) \,\}\{\, \mathtt{h}(x) \leftharpoonup \mathtt{f}(x) \,\} = \varnothing$.

The set of wirings $\mathcal{R}$ is indeed a semiring: a structure similar to a ring, but with no negative elements.

More precisely, let us recall that a *semiring* is a set R equipped with two operations $+$ (the sum) and $\times$ (the product, which symbol is usually omitted), together with an element $0 \in R$ such that: $(R, +, 0)$ is a commutative monoid; $(R, \times)$ is a *semigroup*, *i.e.* a monoid without a neutral element; the product distributes over the sum; the element 0 is absorbent: $0r = r0 = 0$ for all $r \in R$.

We will use an *additive notation* for sets of flows to highlight this situation:

- The symbol $+$ will be used in place of $\cup$.
- We write sets as the sum of their elements: $\{\, f_1, \dots, f_n \,\} := f_1 + \cdots + f_n$.
- We write 0 for the empty set.
- Moreover, we have a neutral element for the product, the unit $I := x \leftharpoonup x$.

Example II.9. The wirings written $W = I + x \leftharpoonup \mathtt{f}(x)$ and $W' = \mathtt{f}(x) \leftharpoonup \mathtt{f}(x)$ respectively stand for the sets $\{\, x \leftharpoonup x\,,\, x \leftharpoonup \mathtt{f}(x) \,\}$ and $\{\, \mathtt{f}(x) \leftharpoonup \mathtt{f}(x) \,\}$. We can compute their product

$$\begin{aligned}
WW' &= \big(x \leftharpoonup x + x \leftharpoonup \mathtt{f}(x)\big)\big(\mathtt{f}(x) \leftharpoonup \mathtt{f}(x)\big)\\
&= \big(x \leftharpoonup x\big)\big(\mathtt{f}(x) \leftharpoonup \mathtt{f}(x)\big) + \big(x \leftharpoonup \mathtt{f}(x)\big)\big(\mathtt{f}(x) \leftharpoonup \mathtt{f}(x)\big)\\
&= \mathtt{f}(x) \leftharpoonup \mathtt{f}(x) + \mathtt{f}(x) \leftharpoonup \mathtt{f}(\mathtt{f}(x))
\end{aligned}$$

Subsets of $\mathcal{R}$ inherit its semigroup/semiring structure when they are stable by the corresponding operations, as is usual with algebraic structures.

Proposition II.10 (semigroup, semiring)

A subset $\mathcal{A}$ of $\mathcal{R}$ is a *semigroup iff.* it satisfies:

1. If $F \in \mathcal{A}$ and $G \in \mathcal{A}$ then $FG \in \mathcal{A}$.

It is a *semiring iff.* it moreover satisfies:

2. $0 \in \mathcal{A}$
3. If $F, G \in \mathcal{A}$, then $F + G \in \mathcal{A}$.

As we will always be working within $\mathcal{R}$, "semigroup" and "semiring" will always mean respectively "subsemigroup of $\mathcal{R}$" and "subsemiring of $\mathcal{R}$".

The notion of *adjoint* is then extended to wirings as $(\sum_i f_i)^\dagger := \sum_i f_i^\dagger$ when $f_i^\dagger$ is defined for all i.

Note that we allow infinite sets in the definition of wirings, which will give a Σ-monoid (definition I.26) structure to wirings. However, wirings coming from the interpretation of terminating programs (theorem III.18) or representing bounded complexity computation (chapter IV) will always be finite.

Remark II.11. Now that we have introduced some material, we can state more precisely what we meant by "a finite description of an infinite set" in section I.1. Indeed, to any $f = t \leftharpoonup u$ we can associate its *closure* $[f]$, defined as

$$[f] := \sum_{\substack{\theta \mid \theta x \text{ is closed} \\ \text{for all } x \in \mathtt{var}(u)}} \theta t \leftharpoonup \theta u$$

This sum is infinite as soon as $\mathtt{var}(u) \neq \varnothing$, and the operation is compatible with the product of flows: we have $[fg] = [f][g]$ for all f and g. Moreover, for any wiring F and any fact $\mathbf{u}$, we have (extending $[\cdot]$ to wirings by linearity) $[F]\mathbf{u} = F\mathbf{u}$, so that this operation preserves the action on facts.

Definition II.12 (height)

The *height* $\mathtt{h}(f)$ of a flow $f = t \leftharpoonup u$ is $\mathtt{max}\{\mathtt{h}(t), \mathtt{h}(u)\}$. The *height* $\mathtt{h}(F)$ of a finite wiring F is the maximal height of flows in it, by convention the height of the empty wiring is 0.

Nilpotency is a standard algebraic concept which will appear in various places of the thesis. In the geometry of interaction approach, it corresponds to strong normalization. From the logic programming point of view, it is related to the notion of boundedness [DEGV01]. We will come back to this in section III.3.

Definition II.13 (nilpotency)

A wiring F is *nilpotent* if $F^n = 0$ for some $n \in \mathbb{N}$.

Example II.14. The wiring $F = \mathtt{g}(x) \leftharpoondown \mathtt{f}(\mathtt{f}(x)) + \mathtt{h}(x) \leftharpoondown \mathtt{g}(x)$ is nilpotent, as $F^2 = \mathtt{h}(x) \leftharpoondown \mathtt{f}(\mathtt{f}(x))$ and $F^3 = 0$.

On the other hand $G = \mathtt{f}(x) \leftharpoondown \mathtt{g}(x) + \mathtt{g}(x) \leftharpoondown \mathtt{f}(x)$ is not nilpotent, as $G^{2n} = \mathtt{f}(x) \leftharpoondown \mathtt{f}(x) + \mathtt{g}(x) \leftharpoondown \mathtt{g}(x)$ and $G^{2n+1} = G$.

Among wirings, those that will produce at most one fact from any fact will be of interest when considering deterministic *vs.* non-deterministic computation in section IV.3.

Definition II.15 (deterministic wirings)

A wiring F is *deterministic* if given any fact $\mathbf{u}$, $\mathit{card}(F\mathbf{u}) \leq 1$.

Remark II.16. It is clear from the definition that if $\mathcal{A}$ is a semigroup, then the set of deterministic wirings of $\mathcal{A}$ is also a semigroup.

The lemma below provides a class of wirings that are deterministic and easy to characterize, due to a more syntactic definition.

Lemma II.17

Let $F = \sum_i t_i \leftharpoondown u_i$. If the u_i are pairwise disjoint (definition I.5) then F is deterministic.

Proof ▸ Given a closed term t, there is at most one of the u_i that matches t, therefore $F(t \leftharpoondown \star)$ is either a single fact or 0. ◂

Example II.18. The wiring $x \leftharpoondown \mathtt{f}(x) + x \leftharpoondown \mathtt{g}(x)$ is deterministic because of the above lemma.

The converse of the lemma does not hold: $x \bullet x \leftharpoondown x \bullet x + x \bullet y \leftharpoondown x \bullet y$ is also deterministic, but does not satisfy the hypothesis of lemma II.17.

We can define a stronger variant of deterministic wirings, which will correspond to the interpretations of λ-terms (chapter III).

Definition II.19 (isometry)

An *isometry* is a wiring $\sum_i t_i \leftharpoondown u_i$ such that all the $t_i \leftharpoondown u_i$ are linear (that is, $\mathtt{var}(t_i) = \mathtt{var}(u_i)$ and any variable occurs exactly once in each term), the t_i are pairwise disjoint and the u_i are pairwise disjoint.

Remark II.20. Again if $\mathcal{A}$ is a semigroup, then the set of isometries of $\mathcal{A}$ is also a semigroup. This case is less obvious because of the more syntactical definition of isometries, but is established by some easy computations.

Finally, let us consider a class of wirings that behave as partial identities.

Definition II.21 (projection)

A projection is a wiring of the form $P = \sum_i t_i \leftharpoonup t_i$.

Proposition II.22

Let P, Q be a projections, we have that

- PQ is a projection.
- For any fact $\mathbf{u}$, either $P\mathbf{u} = \mathbf{u}$ or $P\mathbf{u} = 0$. (we call the *domain* of P the set of facts such that $P\mathbf{u} = \mathbf{u}$)
- For any nilpotent wiring F, PF is also nilpotent.

Proof ▸ The first two assertions are obvious. As for the third one, anticipating on remark II.35, we have that for any wiring G, $G = 0$ *iff.* $G\mathbf{u} = 0$ for any fact $\mathbf{u}$. Therefore $F^n = 0$ implies that for any $\mathbf{u}$, $F^n\mathbf{u} = 0$ which gives $(PF)^n\mathbf{u} = 0$ by the second assertion. ◂

II.2 Semirings constructions

In this section, we define various ways to build semirings that will serve later on, specifically in chapter IV to define the semirings used to capture complexity classes.

Notation

If E is a set of wirings we write

- $\mathtt{vect}(E)$ the set of all finite sums of elements of E.
- $\mathtt{sgroup}(E)$ the set of all finite products of elements of E.
- $\mathtt{sring}(E) := \mathtt{vect}(\mathtt{sgroup}(E))$, the semiring spawned by E.

It is possible to combine together semirings into a new one, using function symbols. This construction is the syntactical counterpart of the standard algebraic notion of tensor product.

Definition II.23

Let g be a n-ary function symbol and $\mathcal{A}_1, \dots, \mathcal{A}_n$ be semirings.

If $t_1 \leftharpoonup u_1, \dots, t_n \leftharpoonup u_n$ are flows (with representatives of the renaming classes that share no variables), we define the flow

$$\mathsf{g}(t_1 \leftharpoonup u_1, \dots, t_n \leftharpoonup u_n) := \mathsf{g}(t_1, \dots, t_n) \leftharpoonup \mathsf{g}(u_1, \dots, u_n)$$

and this is extended to wirings by linearity.

The semiring $\mathtt{g}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ is then defined as

$$\mathtt{sring}\big(\{\, \mathtt{g}(F_1, \dots, F_n) \mid F_i \in \mathcal{A}_i \,\}\big)$$

Notation

In case the function symbol is the binary symbol $\bullet$ we will carry on with the convention for terms and write it in infix notation and as right associating: $\mathcal{A} \bullet \mathcal{B} \bullet \mathcal{C} := \mathcal{A} \bullet (\mathcal{B} \bullet \mathcal{C})$.

Note that this operation is compatible with the product of wirings, in the following sense: $\mathtt{g}(F_1, \dots, F_n)\mathtt{g}(G_1, \dots, G_n) = \mathtt{g}(F_1G_1, \dots, F_nG_n)$. This implies that we have

$$\mathtt{sring}\big(\{\, \mathtt{g}(F_1, \dots, F_n) \mid F_i \in \mathcal{A}_i \,\}\big) = \mathtt{vect}\big(\{\, \mathtt{g}(F_1, \dots, F_n) \mid F_i \in \mathcal{A}_i \,\}\big)$$

so we could have used $\mathtt{vect}$ instead of $\mathtt{sring}$ in the above definition.

Example II.24. Considering $f = \mathtt{c} \leftharpoonup \mathtt{f}(x)$, $g = \mathtt{g}(y) \leftharpoonup y$ and $h = \mathtt{d} \leftharpoonup \mathtt{f}(\mathtt{c})$, we get $f \bullet (g + h) = f \bullet g + f \bullet h = \mathtt{c} \bullet \mathtt{g}(y) \leftharpoonup \mathtt{f}(x) \bullet y + \mathtt{c} \bullet \mathtt{d} \leftharpoonup \mathtt{f}(x) \bullet \mathtt{f}(\mathtt{c})$.

We also introduce a notation for the semiring containing only the unit and zero elements.

Definition II.25 (unit semiring)

The *unit semiring* is defined as the set $\mathcal{I} := \{0, I\}$.

Any set of closed terms induces a semiring: in this case there is no unification involved, only equalities, which ensures that no new term is created by a product (hence we are again in the situation where we can use either $\mathtt{sring}$ or $\mathtt{vect}$ in the definition below).

Definition II.26 (closed semiring)

If E is a set of closed terms, we define the semiring

$$E^{\leftrightharpoons} := \mathtt{sring}\big(\{\, t \leftharpoonup u \mid t, u \in E \,\}\big)$$

Example II.27. Putting the above definition together, we can compute the semiring $\{\mathtt{c}\}^{\leftrightharpoons} = \{0, \mathtt{c} \leftharpoonup \mathtt{c}\}$ and then $\mathcal{I} \bullet \{\mathtt{c}\}^{\leftrightharpoons} = \{0, x \bullet \mathtt{c} \leftharpoonup x \bullet \mathtt{c}\}$.

Finally, one can create new semirings by restricting the use of certain symbols.

Definition II.28 (restriction semiring)

Given P a set of symbols and a semiring $\mathcal{A}$, we define the semiring $\mathcal{A}^{\setminus P}$ as the semiring of elements of $\mathcal{A}$ that do not use any of the symbols in P.

All these construction will be used in chapter IV to define the setting in which we will represent computation and capture complexity classes.

II.3 The balanced semiring

In this section, we study a syntactical constraint on variable height of flows which we call *balance*. This syntactic constraint may be compared with similar ones proposed in order to obtain logic programs that are *finitely ground* [CCIL08, LL09]. Balanced wirings will enjoy properties that will allow to decide their nilpotency efficiently in terms of space.

Definition II.29 (balance)

A flow $f = t \leftharpoonup u$ is *balanced* if for any variable $x \in \mathtt{var}(u)$ all occurrences of x in either t or u have the same height (recall notations page 14).

A wiring F is *balanced* if it is a sum of balanced flows.

We write $\mathcal{R}_\mathbf{b}$ the set of balanced wirings and refer to it as the *balanced semiring*.

Example II.30. The flows $\mathtt{f}(x) \leftharpoonup \mathtt{g}(x)$ and $\mathtt{f}(x \bullet x) \leftharpoonup \mathtt{g}(y) \bullet \mathtt{g}(x)$ are balanced, while $\mathtt{f}(x) \leftharpoonup x$ and $\mathtt{f}(x) \bullet y \leftharpoonup y \bullet x$ are not. Note that in example II.7, only the second product is a product of balanced flows.

Remark II.31. The type of flows rejected by this definition may help to understand why this semiring is related to bounded space computation: flows of the form $\mathtt{f}(x) \leftharpoonup x$ could be used to store information by pushing the information x we already have under a new symbol $\mathtt{f}$, while with balanced flows one can only move around, compare, erase, *etc.* information that is already available. Intuitively, this is the type of handling of information that is possible when manipulating read-only data.

The following lemma summarizes the properties that are preserved by the product of balanced flows. It implies in particular that $\mathcal{R}_\mathbf{b}$ is indeed a semiring.

Lemma II.32

When it is defined, the product fg of two balanced flows f and g is still balanced and its height (definition II.12) is at most $\mathit{max}\{h(f), h(g)\}$.

Proof ▸ We show that the variable height condition and the global height are both preserved by the basic steps of the unification procedure (see the proof of theorem I.8).

Let $f = t \leftharpoonup u$ and $g = v \leftharpoonup w$ We tweak the procedure by

- Adding two variables H, B initialized as $H := t$ and $B := w$ and initializing

P as $P := \{u \doteq v\}$.

- Each time a substitution is applied to S (line 12) also apply it to H and B.
- Associating to each pair in P an integer that remembers the "height of the equation": initially associate 0 to $u \doteq v$, and increment this integer for any new pair produced by a function reduction (lemma I.11).
- Given an element $u' \doteq v'$ of P we call the *corrected height* of a leaf in u' (resp. v') the sum of its height in the term u' (resp. v') and the integer associated to the equation $u' \doteq v'$.

At any point of the procedure, the corrected height of any leaf has not varied. Therefore, for any variable, the corrected height of all its occurrences in any term is the same. Also, the maximal corrected height of any leaf can never increase.

Moreover, if the procedure succeeds, the terms stored in H and B provide the result of the product: $fg = H \leftharpoonup B$ and by the invariant on the height of leaves, we have therefore that fg is balanced and that its height cannot be more than that of f or g. ◂

Corollary II.33 (balanced semiring)

$\mathcal{R}_{\mathbf{b}}$ is a semiring.

The interest of balanced wirings in terms of complexity is that we have an *indirect* way to determine their nilpotency: from a (finite) balanced wiring, we are able to build a (finite) graph containing enough information to solve the problem. Therefore, given a balanced wiring F, one will have the possibility to look at this graph rather than computing the iterations F^n until eventually obtaining 0 as a result, which would be a naive semi-decision procedure.

In what follows we focus on the algebraic aspects of this technique, and leave the complexity issues to section IV.3.

The notion of *separating space* is reminiscent of the notion of separating vector of functional analysis,[1] but needs to be tweaked a little to work properly in our setting.

Definition II.34 (separating space)

A *separating space* for a wiring F is a set of facts $\mathbf{U}$ such that

- $F\mathbf{U} \subseteq \mathbf{U}$.
- For all $f_1, \ldots, f_n \in F$, $(f_1 \cdots f_n)\mathbf{U} = 0$ implies $f_1 \cdots f_n = 0$.

[1] A separating vector v for an operator algebra $\mathcal{A}$ is such that for all $H \in \mathcal{A}$ one has that $H(v) = 0$ implies $H = 0$ or equivalently $H(v) = G(v)$ implies $H = G$: the action on this vector provides enough information to separate elements of $\mathcal{A}$.

Remark II.35. Note that the set of *all* facts is separating for any wiring and that if $\mathbf{U}$ is separating for F and $G \subseteq F$, then $\mathbf{U}$ is separating for G.

Also, $\mathbf{U}$ separating for F immediately implies the following property: if $F^n\mathbf{U} = 0$, then $F^n = 0$.

A separating space can be thought of as a subset of the Herbrand universe [DEGV01] associated to a logic program, stable by resolution with rules of the program and large enough to determine if a sequence of composition of clauses has a null product.

We can define such a space for balanced wirings with lemma II.32 in mind: balanced wirings behave well with respect to the height of terms.

Definition II.36 (computation space)

Given a wiring F, we define its *computation space* $\mathbf{Comp}(F)$ as the set of facts of height at most $\hbar(F)$, built using only the symbols appearing in F and the constant symbol $\star$.

Lemma II.37 (separation)

If F is balanced, then $\mathbf{Comp}(F)$ is separating for F.

Proof ▸ By lemma II.32, $F\,\mathbf{Comp}(F)$ is of height at most $\hbar(F)$ and it contains only symbols occurring in F and $\star$, therefore $F\,\mathbf{Comp}(F) \subseteq \mathbf{Comp}(F)$.

By lemma II.32 again, if $f_1, \dots, f_n \in F$ then the product $f_1 \cdots f_n$ is still of height at most $\hbar(F)$. If $(f_1 \cdots f_n)\mathbf{Comp}(F) = 0$, it means that $f_1 \cdots f_n$ does not match any closed term of height at most $\hbar(F)$ built with the symbols of F and $\star$. This is only possible if $f_1 \cdots f_n = 0$. ◂

If F is a finite wiring, thus built with finitely many symbols, $\mathbf{Comp}(F)$ is also a finite set. We can be a little more precise and give a bound to its cardinal. This gives an idea of the amount of space needed to enumerate the elements of $\mathbf{Comp}(F)$, a crucial point in the proof of lemma IV.11.

Proposition II.38 (cardinality)

Let F be a balanced and finite wiring, A the maximal arity of function symbols occurring in F and S the number of distinct symbols occurring in F, then

$$\mathit{card}\big(\mathbf{Comp}(F)\big) \;\leq\; (S+1)^{P_{\hbar(F)}(A)}$$

where $P_h(X) = 1 + X + \cdots + X^h$.

Proof ▸ The number of terms of height $\hbar(F)$ built over the set of symbols $S \cup \{\star\}$ of arity at most A is bounded by the number of complete trees of degree A and height $\hbar(F)$ (that is, trees where nodes of height less than $\hbar(F)$

have exactly A childs) with nodes labeled by elements of $S \cup \{\star\}$. ◂

Then, we can encode in a directed graph[2] the action of the wiring on its computation space.

Definition II.39 (computation graph)

If F is a balanced wiring, we define its *computation graph* $\mathbf{G}(F)$ as the directed graph:

- The vertices of $\mathbf{G}(F)$ are the elements of $\mathbf{Comp}(F)$.
- There is an edge from $\mathbf{u}$ to $\mathbf{v}$ in $\mathbf{G}(F)$ if $\mathbf{v} \in F\mathbf{u}$.

Indeed the computation graph of a wiring contains enough information on the latter to determine its nilpotency. This will be a key ingredient in the logarithmic space decision procedure of section IV.3, as the search for paths and cycles in graphs are problems that are well-known to be solvable within logarithmic space.

Theorem II.40

A finite and balanced wiring F is nilpotent (definition II.13) *iff.* $\mathbf{G}(F)$ is acyclic.

Proof ▸ Suppose there is a cycle of length n in $\mathbf{G}(F)$, and let $\mathbf{u}$ be the label of a vertex which is part of this cycle. By definition of $\mathbf{G}(F)$, $\mathbf{u} \in (F^n)^k\mathbf{u}$ for all k, which means that $(F^n)^k \neq 0$ for all k and therefore F cannot be nilpotent.

Conversely, suppose there is no cycle in $\mathbf{G}(F)$. As it is a finite graph, this entails a maximal length N of paths in $\mathbf{G}(F)$. By definition of $\mathbf{G}(F)$, this means that $F^{N+1}\mathbf{u} = 0$ for all $\mathbf{u} \in \mathbf{Comp}(F)$. As $\mathbf{Comp}(F)$ is separating for F (lemma II.37) we get $F^{N+1} = 0$ by remark II.35. ◂

Moreover, the computation graph of a deterministic wiring (definition II.15) has a specific shape, which will in turn induce a deterministic decision procedure in this case.

Lemma II.41

If F is a balanced and deterministic (definition II.15) wiring, $\mathbf{G}(F)$ has an out-degree bounded by 1.

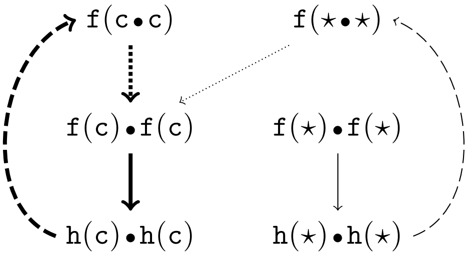
Proof ▸ A direct consequence of the definitions of $\mathbf{G}(F)$ and determinism. ◂

[2] Here by directed graph we mean the standard notion: a set of *vertices* V together with a set of *edges* $E \subseteq V \times V$. The *source* of an edge (e, f) is e and its *target* is f. We say that there is an edge *from* $e \in V$ *to* $f \in V$ when $(e, f) \in E$. The out-degree of a graph is the maximal number of edges a vertex can be the source of.

Let us illustrate the technique above on a simple example. Consider the balanced flow

$$\begin{aligned} F := \quad & \mathtt{h}(x)\bullet\mathtt{h}(x) \leftharpoonup \mathtt{f}(x)\bullet\mathtt{f}(x) \\ + \quad & \mathtt{f}(\mathtt{c})\bullet\mathtt{f}(\mathtt{c}) \leftharpoonup \mathtt{f}(x\bullet x) \\ + \quad & \mathtt{f}(x\bullet x) \leftharpoonup \mathtt{h}(x)\bullet\mathtt{h}(x) \end{aligned}$$

Its computation space is the set of terms of height at most 2 built with the symbols $\mathtt{f}, \mathtt{h}, \bullet, \mathtt{c}, \star$. We draw below $\mathbf{G}(F)$, omitting vertices connected to no edges. We draw the edges induced by the first flow as plain lines, those induced by the second one as dotted lines and those induced by the third one as dashed lines. We also highlight the cycle in the graph using thicker lines.



The cycle in this graph corresponds to the fact that $F^3(\mathtt{f}(\mathtt{c}\bullet\mathtt{c}) \leftharpoonup \star)$ contains $\mathtt{f}(\mathtt{c}\bullet\mathtt{c}) \leftharpoonup \star$. Note that F being deterministic by lemma II.17, we obtain a graph with an out-degree 1 as an illustration of lemma II.41.

II.4 The stack semiring

We saw that the typical example of a non-balanced flow is $\mathtt{f}(x) \leftharpoonup x$. In this section, we study the semiring of flows built only with unary function symbols and a variable, thought as manipulations of a "stack" of symbols.

The restriction to unary function symbols implies a number of properties with respect to product, and therefore nilpotency, that we will use in section IV.4 to build a polynomial time decision procedure for their nilpotency problem.

Definition II.42 (*Stack* semiring)

A *unary flow* is a flow $t \leftharpoonup u$ built using only unary function symbols and a unique variable, occurring both in t and u.

The semiring *Stack* is the set of wirings of the form $\sum_i t_i \leftharpoonup u_i$ where the $t_i \leftharpoonup u_i$ are all unary flows.

Example II.43. The flows $\mathtt{f}(\mathtt{f}(x)) \leftharpoonup \mathtt{g}(x)$ and $x \leftharpoonup \mathtt{g}(\mathtt{h}(x))$ are unary, while $\mathtt{f}(\mathtt{c}) \leftharpoonup \mathtt{g}(x)$ and $x\bullet\mathtt{f}(x) \leftharpoonup \mathtt{g}(x)$ are not.

Let us fix some notations for elements of this semiring.

Notation (stack operations)

If $\tau = \mathtt{f}_1, \dots, \mathtt{f}_n$ is a finite sequence of unary function symbols and t is a term, we write $\tau(t) := \mathtt{f}_1(\mathtt{f}_2(\cdots \mathtt{f}_n(t) \cdots))$.

Given two sequences τ and σ we define the flow:

$$\mathtt{OP}_{\tau,\sigma} := \tau(x) \leftharpoonup \sigma(x)$$

which we call a *stack operation*.

Finally, $\tau\sigma$ will denote the concatenation of the sequences τ and σ.

By definition, any element of $\mathcal{S}tack$ is a sum of stack operations. Moreover, it is clear that any $\mathtt{OP}_{\tau,\sigma}$ can be decomposed as a product of elements of the form $\mathtt{PUSH}_{\mathtt{f}} := \mathtt{f}(x) \leftharpoonup x$ and $\mathtt{POP}_{\mathtt{f}} := x \leftharpoonup \mathtt{f}(x)$, hence the name name of this semiring.

This type of flows have already been studied in an article on elementary complexity and geometry of interaction [BP01].[3] We can therefore borrow a result from this work and state it in our setting. For this purpose, let us introduce the notion of *cyclicity*.

Definition II.44 (cyclicity)

A flow $t \leftharpoonup u$ is a *cycle* if t and u are matchable (definition I.5).

A wiring F is *cyclic* if there is a n such that F^n contains a cycle.[4]

Let $\vec{s} = f_1, \dots, f_n$ be a sequence of stack operations. We define:

- The *height* of the sequence as $\hbar(\vec{s}) := max_i\{\hbar(f_i)\}$ (the notation $\hbar(\cdot)$ coming from definition II.12).
- The *cardinality* of the sequence as $card(\vec{s}) := card\{\, f_i \mid 1 \leq i \leq n \,\}$. That is, the number of distinct stack operations appearing in $\vec{s}$.
- We write $p(\vec{s})$ the result of the product $f_1 \cdots f_n$.

The sequence $\vec{s}$ is said to be *cyclic* if there is a sub-sequence $\vec{s}_{i,j} = f_i, \dots, f_j$ with $1 \leq i \leq j \leq n$ such that $p(\vec{s}_{i,j})$ is a cycle.

Remark II.45. It is immediate that a flow f is a cycle *iff.* $f^2 \neq 0$.

Example II.46. The flow $\mathtt{f}(x) \leftharpoonup x$ is a cycle with $(\mathtt{f}(x) \leftharpoonup x)^2 = \mathtt{f}(\mathtt{f}(x)) \leftharpoonup x$.

Consider the sequence

$$\vec{s} := \mathtt{h}(\mathtt{h}(x)) \leftharpoonup \mathtt{g}(x)\ ,\ \mathtt{f}(x) \leftharpoonup \mathtt{h}(x)\ ,\ \mathtt{h}(\mathtt{h}(x)) \leftharpoonup \mathtt{g}(x)\ ,\ \mathtt{g}(x) \leftharpoonup \mathtt{f}(x)$$

which is such that $\hbar(\vec{s}) = 2$ and $card(\vec{s}) = 3$, as $\vec{s}$ is of length 4 but its first and third elements are equal.

[3] In this article, flows of the form $\mathbf{u}(t_1, \dots, t_n) \leftharpoonup \mathbf{v}(u_1, \dots, u_m)$ with the t_i and u_j unary satisfying some additional properties are considered. Unary flows correspond to the special case where both $\mathbf{u}$ and $\mathbf{v}$ are unary.

[4] Note that the wiring 0 is therefore acyclic.

Because $\big(x \leftharpoonup \mathtt{g}(x)\big)\big(\mathtt{f}(x) \leftharpoonup \mathtt{h}(x)\big) = 0$ we have $p(\vec{s}) = 0$. Still, $\vec{s}$ is cyclic because its sub-sequence

$$\vec{r} := \mathtt{f}(x) \leftharpoonup \mathtt{h}(x)\ ,\ \mathtt{h}(\mathtt{h}(x)) \leftharpoonup \mathtt{g}(x)\ ,\ \mathtt{g}(x) \leftharpoonup \mathtt{f}(x)$$

is such that $p(\vec{r}) = \mathtt{f}(\mathtt{h}(x)) \leftharpoonup \mathtt{f}(x)$.

The following lemma gives a bound for the maximal height the product of a sequence can reach without the sequence being cyclic. This bound depends on the height and cardinal of the sequence.

Lemma II.47 (acyclic sequence [BP01, lemma 5.3])

Let $\vec{s}$ be a sequence of stack operations.

If $\vec{s}$ is acyclic, then $h\left(p(\vec{s})\right) \leq h\left(\vec{s}\right)\left(card(\vec{s}) + 1\right)$.

The interest of the notion of cyclicity is that in the case of stack operations (*i.e.* when manipulating unary function symbols) a cycle can be composed with itself indefinitely, thus being non-nilpotent.

This relies on the fact that stack operations behave in a particular way with respect to product. Indeed when a unification succeeds for two flows of this form, only one of them is modified by the resulting MGU in order to match the other.

Proposition II.48 (product of stack elements)

Given two stack operations $\mathtt{OP}_{\tau,\sigma}$ and $\mathtt{OP}_{\rho,\chi}$, such that $\mathtt{OP}_{\tau,\sigma}\mathtt{OP}_{\rho,\chi} \neq 0$, we have a sequence μ such that either

$$\mathtt{OP}_{\tau,\sigma}\mathtt{OP}_{\rho,\chi} = \mathtt{OP}_{\tau,\chi\mu} \qquad \text{or} \qquad \mathtt{OP}_{\tau,\sigma}\mathtt{OP}_{\rho,\chi} = \mathtt{OP}_{\tau\mu,\chi}$$

Proof ▸ If $\mathtt{OP}_{\tau,\sigma}\mathtt{OP}_{\rho,\chi} \neq 0$, then $\sigma(x)$ and $\rho(x)$ are matchable and we have a μ such that either $\sigma = \rho\mu$ or $\sigma\mu = \rho$. ◂

Corollary II.49

If the stack operation $\mathtt{OP}_{\tau,\sigma}$ is a cycle, then $(\mathtt{OP}_{\tau,\sigma})^n \neq 0$ for all n.

Proof ▸ If $\mathtt{OP}_{\tau,\sigma}$ is a cycle then there is a μ such that either $\sigma\mu = \tau$ or $\sigma = \tau\mu$. Suppose for instance we are in the first situation (the second being symmetric). Then we can compute $(\mathtt{OP}_{\tau,\sigma})^{n+1} = \mathtt{OP}_{\tau\mu^n,\sigma} \neq 0$. ◂

Example II.50. For instance with the flow $f := \mathtt{f}(\mathtt{h}(x)) \leftharpoonup \mathtt{f}(x)$, we have $\tau = \mathtt{fh}$, $\sigma = \mathtt{f}$ and therefore $\sigma\mu = \tau$ with $\mu = \mathtt{h}$.

This gives the iterations $f^{n+1} = \mathtt{f}(\mathtt{h}(\underbrace{\mathtt{h}(\cdots\mathtt{h}}_{n\text{ times}}(x)\cdots))) \leftharpoonup \mathtt{f}(x)$.

Remark II.51. This property does not hold in general: consider $f = x \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet x$, this flow is a cycle as $f^2 = \mathtt{c} \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet \mathtt{d} \neq 0$ (here we rely on remark II.45) but $f^3 = (x \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet x)(\mathtt{c} \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet \mathtt{d}) = 0$. Note that the use of a binary symbol is crucial to obtain this situation.

One of the key consequences of lemma II.47 is that cyclicity turns out to be the only way for finite elements of $\mathcal{S}\mathit{tack}$ not to be nilpotent.

Theorem II.52 (nilpotency in $\mathcal{S}\mathit{tack}$)

A finite wiring $F \in \mathcal{S}\mathit{tack}$ is not nilpotent *iff.* it is cyclic.

Proof ▸ Suppose F is not nilpotent, so that there is at least one stack operation $f \in F^n$ for any n and let S be the number of different function symbols appearing in F.

We set $k := (S^{\mathfrak{h}(F)(\mathfrak{card}(F)+1)} + S^{\mathfrak{h}(F)(\mathfrak{card}(F)+1)-1} + \dots + 1)^2$, which is the cardinal of the set of flows of height at most $\mathfrak{h}(F)(\mathfrak{card}(F)+1)$ using the function symbols appearing in F.

Let $f \neq 0$ be an element of F^{k+1}, it is the product $\mathfrak{p}(\vec{s})$ of a sequence $\vec{s} = f_1, \dots, f_{k+1}$ of stack operations that belong to F. We show by contradiction that this sequence must be cyclic, so let us suppose it is not. By lemma II.47, we know that for any $i > 0$, setting $\vec{s}_i := f_1, \dots, f_i$ we have

$$\mathfrak{h}\left(\mathfrak{p}(\vec{s}_i)\right) \leq \mathfrak{h}(\vec{s}_i)(\mathfrak{card}(\vec{s}_i) + 1) \leq \mathfrak{h}(F)(\mathfrak{card}(F) + 1)$$

Therefore for any $i > 0$ the flow $\mathfrak{p}(\vec{s}_i)$ is of height at most $\mathfrak{h}(F)(\mathfrak{card}(F) + 1)$ and uses only symbols appearing in F, *i.e.* it wanders in a set of cardinal k, so there must be $1 \leq i < j \leq k+1$ such that $\mathfrak{p}(\vec{s}_i) = \mathfrak{p}(\vec{s}_j)$.

Now, setting $\vec{s}_{i+1,j} := f_{i+1}, \dots, f_j$ we have that $\mathfrak{p}(\vec{s}_i)\mathfrak{p}(\vec{s}_{i+1,j}) = \mathfrak{p}(\vec{s}_j) = \mathfrak{p}(\vec{s}_i)$ hence $\mathfrak{p}(\vec{s}_i)\mathfrak{p}(\vec{s}_{i+1,j})^2 = \mathfrak{p}(\vec{s}_i) \neq 0$ and thus $\mathfrak{p}(\vec{s}_{i+1,j})^2 \neq 0$. That is, $\mathfrak{p}(\vec{s}_{i+1,j})$ is a cycle.

As $\mathfrak{p}(\vec{s}_{i+1,j}) \in F^{j-i}$ we can conclude that F is cyclic.

The converse immediately follows from corollary II.49. ◂

Example II.53. Consider the wiring

$$\begin{aligned} F := \quad & \mathtt{f}_1(x) \leftharpoonup \mathtt{f}_0(x) \\ + \quad & \mathtt{f}_0(\mathtt{f}_1(x)) \leftharpoonup \mathtt{f}_1(\mathtt{f}_0(x)) \\ + \quad & \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_1(x))) \leftharpoonup \mathtt{f}_1(\mathtt{f}_1(\mathtt{f}_0(x))) \\ + \quad & \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \leftharpoonup \mathtt{f}_1(\mathtt{f}_1(\mathtt{f}_1(x))) \end{aligned}$$

which implements a sort of counter from 0 to 7 in binary notation (we see the sequence $\mathtt{f}_x\mathtt{f}_y\mathtt{f}_z$ as the integer $x + 2y + 4z$) that resets to 0 when it reaches 8.

It is quite clear with this intuition in mind that this wiring is cyclic. Indeed, an easy computation shows that $\mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \leftharpoonup \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \in F^8$.

If we lift this example to the case of a counter from 0 to $2^n - 1$ that resets to 0 when it reaches 2^n, we obtain an example of a wiring F of cardinal n and height $n-1$ such that F^{2^n} contains a cycle, but F^{2^n-1} does not. Even if this does not fully reach the bound we used in the above proof, it shows that the number of iterations needed to find a cycle may be exponential in the height and the cardinal of F, which rules out a polynomial time decision procedure for the nilpotency problem that would simply compute the iterations of F until it finds a cycle in it.

Note moreover that the flow F is balanced (definition II.29), therefore balanced flows are also concerned with these remarks.

With this result we obtained a first reduction of nilpotency to a simpler property: acyclicity. As we saw in the above example, we need to go further than this if we want to be able to decide the problem in polynomial time.

The first step in this direction is to remark that under certain conditions, the product of two stack operations does not grow in height.

Notation

If $\mathcal{h}\left(\tau(x)\right) \geq \mathcal{h}\left(\sigma(x)\right)$ we say that $\texttt{op}_{\tau,\sigma}$ is *increasing*.

If $\mathcal{h}\left(\tau(x)\right) \leq \mathcal{h}\left(\sigma(x)\right)$ we say it is *decreasing*.

Lemma II.54

Let $f = \texttt{op}_{\tau,\sigma}$ and $g = \texttt{op}_{\rho,\chi}$. If f is decreasing and g is increasing, then we have $\mathcal{h}(fg) \leq \mathcal{max}\{\mathcal{h}(f), \mathcal{h}(g)\}$.

Proof ▸ If $fg = 0$, the property is satisfied because $\mathcal{h}(0) = 0$. Otherwise, we have either $\sigma = \rho\mu$ or $\sigma\mu = \rho$.

Suppose we are in the first case (the second being symmetric). Then we have $fg = \texttt{op}_{\tau,\chi\mu}$ and $\mathcal{h}(\sigma) = \mathcal{h}(\rho\mu)$. As g is increasing, $\mathcal{h}(\chi) \leq \mathcal{h}(\rho)$ and therefore $\mathcal{h}(\chi\mu) \leq \mathcal{h}(\rho\mu) = \mathcal{h}(\sigma) \leq \mathcal{h}(f) \leq \mathcal{max}\{\mathcal{h}(f), \mathcal{h}(g)\}$. ◂

Example II.55. We have that

$$\big(\texttt{f}(x) \leftharpoonup \texttt{h}(\texttt{f}(x))\big)\big(\texttt{h}(\texttt{f}(\texttt{f}(x))) \leftharpoonup x\big) = \texttt{f}(\texttt{f}(x)) \leftharpoonup x$$

Note that the lemma does not give any information on the increasing or decreasing nature of the result, only its height.

The reversed property (when increasing and decreasing are swapped) does not hold. For instance if we revert the product above, we get

$$\big(\texttt{h}(\texttt{f}(\texttt{f}(x))) \leftharpoonup x\big)\big(\texttt{f}(x) \leftharpoonup \texttt{h}(\texttt{f}(x))\big) = \texttt{h}(\texttt{f}(\texttt{f}(\texttt{f}(x)))) \leftharpoonup \texttt{h}(\texttt{f}(x))$$

which is a product of flows of height 3 and 2 that yields a result of height 4.

With this lemma in mind we can define a notion of *saturation* of an element of $\mathcal{S}tack$, by recursively composing its decreasing and increasing stack operations.

Definition II.56 (saturation)

If $F \in \mathcal{S}tack$ we define the subsets $F^{\uparrow} := \{ f \in F \mid f \text{ is increasing} \}$ and $F^{\downarrow} := \{ f \in F \mid f \text{ is decreasing} \}$.

We set the *shortcut* operation $\mathtt{short}(F) := F + F^{\downarrow}F^{\uparrow}$ and its least fixpoint

$$\mathtt{sat}(F) := \sum_{n\in\mathbb{N}} \mathtt{short}^{n}(F) \qquad \textit{(where } \mathtt{short}^n \textit{ denotes the } n^{th} \textit{ iteration of } \mathtt{short}\textit{)}$$

which we call the *saturation* of F.

It is a direct consequence of lemma II.54 that if F is finite the fixpoint is reached in a finite number of steps.

Proposition II.57

Let $F \in \mathcal{S}tack$ be a finite wiring and S the number of distinct function symbols appearing in F. For any n we have that $\mathtt{h}\left(\mathtt{short}^{n}(F)\right) = \mathtt{h}(F)$.

Moreover if $n \geq (S^{\mathtt{h}(F)} + S^{\mathtt{h}(F)-1} + \cdots + 1)^2$ then $\mathtt{short}^{n}(F) = \mathtt{sat}(F)$.

Proof ▸ By lemma II.54 we have that $\mathtt{h}(F^{\downarrow}F^{\uparrow}) \leq \mathtt{max}\{\mathtt{h}(F^{\downarrow}), \mathtt{h}(F^{\uparrow})\} = \mathtt{h}(F)$ therefore $\mathtt{h}\left(\mathtt{short}(F)\right) = \mathtt{h}(F)$ and we get the first property by induction.

For any n the elements of $\mathtt{short}^{n}(F)$ are stack operations of height at most $\mathtt{h}(F)$ built with the function symbols from F. As this set of stack operations is of cardinal $k := (S^{\mathtt{h}(F)} + S^{\mathtt{h}(F)-1} + \cdots + 1)^2$ and $G \subseteq \mathtt{short}(G)$ for all G, the iteration of $\mathtt{short}(\cdot)$ on F is stable after at most k steps. ◂

Remark II.58. The idea of the $\mathtt{short}(\cdot)$ operation is very close in spirit with the idea of "exponentiation by squaring". Indeed $F^{\downarrow}F^{\uparrow}$ is a subset of F^2 that contains only terms of height at most $\mathtt{h}(F)$. This allows to reach much faster some elements that belong to exponential iterations of F.

Even if the general situation might be slightly more complex, we can see for instance that in the case of example II.53, $\mathtt{short}^{n}(F)$ already contains a cycle that would normally be found in F^{2^n}.

The saturation operation allows for a further reduction of the nilpotency problem to the acyclicity of wirings that contains operations that are all increasing or all decreasing.

Lemma II.59 (rotation)

Let f and g be stack operations, we have that fg is a cycle *iff.* gf is a cycle.

Proof ▸ If fg is a cycle, then $(fg)^n \neq 0$ for any n by corollary II.49. In particular $(fg)^3 \neq 0$ and as $(fg)^3 = f(gf)(gf)g$ we get $(gf)^2 \neq 0$, *i.e.* gf is a cycle. ◂

Theorem II.60

Any finite $F \in \mathcal{S}tack$ is cyclic *iff.* either $\mathtt{sat}(F)^{\uparrow}$ or $\mathtt{sat}(F)^{\downarrow}$ is.

Proof ▸ The cyclicity of $\mathtt{sat}(F)^{\uparrow}$ or $\mathtt{sat}(F)^{\downarrow}$ obviously implies that of F because $\mathtt{short}(F) \subseteq F + F^2$, hence $\mathtt{sat}(F) \subseteq \sum_{n\in\mathbb{N}} F^n$.

Conversely, suppose F is cyclic and let $\vec{s} = f_1, \dots, f_n \in F$ such that the product $p(\vec{s}) \in F^n$ is a cycle.

We are going to produce from $\vec{s}$ a sequence of elements of $\mathtt{sat}(F)^{\uparrow}$ or $\mathtt{sat}(F)^{\downarrow}$ which product is a cycle. For this we apply to the sequence the following rewriting procedure:

1. If there are f_i and f_{i+1} such that f_i is decreasing and f_{i+1} is increasing, then rewrite $\vec{s}$ as $f_1, \dots, f_i f_{i+1}, \dots, f_n$.
2. If step 1 does not apply and $\vec{s} = \vec{s}_1\vec{s}_2$ ($\vec{s}_1$ and $\vec{s}_2$ both non-empty) with all elements of $\vec{s}_1$ increasing and all elements of $\vec{s}_2$ decreasing, then rewrite $\vec{s}$ as $\vec{s}_2\vec{s}_1$.

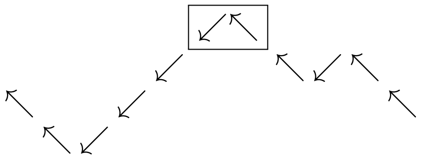
This rewriting procedure preserves the following invariants:

- All elements of the sequence are in $\mathtt{sat}(F)$: step 2 does not affect the elements of the sequence (only their order) and step 1 replaces the flows $f_i \in \mathtt{sat}(F)^{\downarrow}$ and $f_{i+1} \in \mathtt{sat}(F)^{\uparrow}$ by $f_i f_{i+1} \in \mathtt{sat}(F)$.
- The product $p(\vec{s})$ of the sequence is a cycle: step 1 does not alter $p(\vec{s})$ and step 2 does not alter the fact that $p(\vec{s})$ is a cycle by lemma II.59.

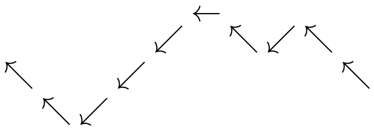
The rewriting terminates as step 1 strictly reduces the length of the sequence and step 2 can never be applied twice in a row (it can be applied only when step 1 is impossible and its application makes step 1 possible). Let $g_1, \dots, g_n$ be the resulting sequence. As $g_1, \dots, g_n$ cannot be reduced, the g_i must be either all increasing or all decreasing.

Therefore, by the invariants above $g_1, \dots, g_n$ is either a sequence of elements of $\mathtt{sat}(F)^{\downarrow}$ or $\mathtt{sat}(F)^{\uparrow}$ such that the product $g_1 \cdots g_n$ is a cycle. ◂

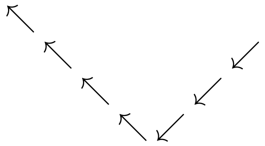
We can give a graphical account of the above rewriting procedure: if we depict an increasing stack operation as an arrow going up ↖ and a decreasing one as an arrow going down $\swarrow$, we can then depict a sequence of stack operation a sort of mountainous landscape



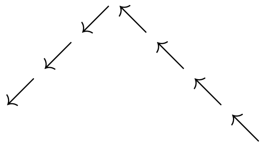
and the step 1 of the reduction the appears as "pruning the peaks" of the landscape. For instance, suppose we prune the highlighted peak above (drawing the result as a $\leftarrow$ as it can be either increasing or decreasing) the landscape would become



if step 1 cannot be applied, it must be that there are no peaks, and therefore the landscape looks like



and applying step 2 would yield



thus creating a new peak and allowing to perform step 1 anew.

Finally, we study the special case of wirings that contain operations that are either all increasing or all decreasing, showing that this case can be reduced to the balanced wirings of the previous section.

Notation

Given a set of unary function symbols E and an integer h, we define the associated *truncation* wiring as

$$\mathtt{TR}_{[E,h]} := \sum_{\tau=\mathtt{f}_1,\dots,\mathtt{f}_h\in E} \tau(\star) \leftharpoonup \tau(x)$$

The action of $\tau(\star) \leftharpoonup \tau(x)$ can be intuitively understood as follows: take a stack that start by τ and remove its bottom, replacing it with the constant symbol $\star$.

Theorem II.61

Let $F \in \mathcal{S}tack$ be a wiring containing only increasing stack operations, E the set of function symbols used in F and $h \geq \hbar(F)$ an integer.

The wiring $\texttt{TR}_{[E,h]}F$ is balanced (definition II.29) and $\hbar(\texttt{TR}_{[E,h]}F) = h$.

Moreover, F is nilpotent *iff.* $\texttt{TR}_{[E,h]}F$ is nilpotent.

Proof ▸ It is clear that $\texttt{TR}_{[E,h]}F$ is balanced as it contains only flows of the form $\tau(\star) \leftharpoonup \sigma(x)$, with only one variable.

An easy computation shows that $(\texttt{TR}_{[E,h]}F)(\texttt{TR}_{[E,h]}F) = \texttt{TR}_{[E,h]}F^2$ and therefore $(\texttt{TR}_{[E,h]}F)^n = \texttt{TR}_{[E,h]}F^n$; moreover if τ and σ contain only symbols from E then $\texttt{TR}_{[E,h]}\texttt{OP}_{\tau,\sigma} \neq 0$ therefore $\texttt{TR}_{[E,h]}F^n = 0$ *iff.* $F^n = 0$.

Finally we get $(\texttt{TR}_{[E,h]}F)^n = 0$ *iff.* $F^n = 0$, that is $(\texttt{TR}_{[E,h]}F)$ is nilpotent *iff.* F is nilpotent. ◂

Remark II.62. The decreasing case can be treated by symmetry: if F contains only decreasing stack operations, then its adjoint $F^\dagger$ contains only increasing stack operations and is nilpotent *iff.* F is nilpotent, because $(F^\dagger)^n = (F^n)^\dagger$.

Chapter III

Geometry of Interaction in $\mathcal{R}$

The GoI interpretation of linear logic in the resolution semiring was first studied as a direct construction by J.-Y. Girard [Gir89a, Gir95a] and its fundamental properties were proven directly. For instance the associativity axiom of the trace (definition I.18, item 5), which corresponds to the Church-Rosser property, might be delicate to prove directly while it holds as an automatically in a UDC.

In this chapter we carry on a more abstract proof of the possibility of interpreting linear logic (and even pure λ-CALCULUS), by means of the categorical framework introduced in section I.3: we will show that a unique decomposition category $\mathfrak{R}$ can be built within the resolution semiring, and that this category yields a GoI situation. This automatically implies that $\mathfrak{R}$ has all the required structures to interpret the dynamics of the pure λ-CALCULUS.

We provide the reader with a reminder of how the interpretation goes in a category with a GoI situation in the appendix.

The third section of this chapter discusses briefly the connection with logic programming this interpretation entails. This point was already evoked in Girard's work [Gir89b, Gir95a] but has not been explored much further since then. We will see in particular that the notion of boundedness of logic programs is related to the algebraic notion of nilpotency and that the ability to model the dynamics of λ-CALCULUS yields an undecidability result for the nilpotency problem of the resolution semiring.

Contents

III.1 A traced category of logic programs

We begin by showing that a concrete traced category can be built within the resolution semiring.

We start by restricting to a specific class of wirings, that correspond more closely to the notion of logic programs, as we reintroduce a distinction between predicate symbols and usual function symbols.

Definition III.1 (clause wiring)

We assume a denumerable family of function symbols (which we call *predicate symbols*) $\mathbf{u}_{n[i]}$ for $n \in \mathbb{Z}$ and $i \in \mathbb{N}$, with $\mathbf{u}_{n[i]}$ of arity i.

A *clause* is then a flow of the form

$$l = \mathbf{u}_{n[i]}(t_1, \dots, t_i) \leftharpoonup \mathbf{u}_{m[j]}(u_1, \dots, u_j)$$

The *input symbol* $\mathtt{in}(l)$ of l is the predicate symbol $\mathbf{u}_{m[j]}$, its *output symbol* $\mathtt{out}(l)$ is the predicate symbol $\mathbf{u}_{n[i]}$.

A *clause wiring* is a wiring F containing only clauses, its set of input $\mathtt{in}(F)$ (resp. output, $\mathtt{out}(F)$) symbols is the set of input (resp. output) symbols of the clauses in it.

Notation

We will write the clause $\mathbf{u}_{n[i]}(x_1, \dots, x_i) \leftharpoonup \mathbf{u}_{n[i]}(x_1, \dots, x_i)$ simply as $\mathbf{u}_{n[i]} \leftharpoonup \mathbf{u}_{n[i]}$.

Then, our category will have interface specifications as objects and clause wirings satisfying the specifications as morphisms.

Definition III.2 (resolution category)

The *resolution category* $\mathfrak{R}$ is defined as:

- *Objects:* finite (possibly empty) sequences of integers $\langle i_1, \dots, i_n \rangle$. We write $\mathtt{l}(A)$ the length of the sequence A.
- *Morphisms:* a morphism from $\langle j_1, \dots, j_m \rangle$ to $\langle i_1, \dots, i_n \rangle$ is a clause wiring F with

$$\mathtt{in}(F) \subseteq \{ \mathbf{u}_{1[j_1]}, \dots, \mathbf{u}_{m[j_n]} \} \quad \text{and} \quad \mathtt{out}(F) \subseteq \{ \mathbf{u}_{1[i_1]}, \dots, \mathbf{u}_{n[i_n]} \}$$

The *composition* in $\mathfrak{R}$ is the product of wirings, the *identities* are defined as

$$\mathrm{Id}_{\langle i_1, \dots, i_n \rangle} := \mathbf{u}_{1[i_1]} \leftharpoonup \mathbf{u}_{1[i_1]} + \cdots + \mathbf{u}_{n[i_n]} \leftharpoonup \mathbf{u}_{n[i_n]}$$

The associativity of composition comes directly from the associativity of product of flows and the fact that identities behave as expected follows from an easy computation.

The semiring structure of $\mathcal{R}$ transfers to $\mathfrak{R}$ as a Σ-monoid structure for the sets of morphisms, with a distributivity property of the product over the sum, two requirements of definition I.29.

Remark III.3. Here as we chose to consider plain sets of flows with no topology, any infinite sum makes sense, which is a delicate point in operator algebraic approaches.

There are two options when facing this situation: one can either choose to work with partially defined infinite sums [MSS12] or look for a specific structure where a total trace can still be defined [Gir90, Gir06].

The resolution category can be endowed with a monoidal structure in a very straightforward way, by shifting interfaces to avoid undesired interferences between clauses.

Notation

The *shift* wirings are defined for all $k \in \mathbb{Z}$ as $\mathbf{S}_k := \sum_{n\in\mathbb{Z}, i\in\mathbb{N}} \mathbf{u}_{(n+k)[i]} \leftharpoonup \mathbf{u}_{n[i]}$.

Definition III.4 (symmetric monoidal structure)

If we have $A = \langle i_1, \dots, i_n\rangle$ and $B = \langle j_1, \dots, j_m\rangle$ two objects of $\mathfrak{R}$, we define $A \oplus B := \langle i_1, \dots, i_n, j_1, \dots, j_m\rangle$ (*i.e.* the concatenation of the two sequences).
If we have $F : B_1 \leftarrow A_1$ and $G : B_2 \leftarrow A_2$, then $F \oplus G : B_1 \oplus B_2 \leftarrow A_1 \oplus A_2$ is defined as

$$F \oplus G := F + \mathbf{S}_{\iota(B_1)}\, G\, \mathbf{S}_{-\iota(A_1)}$$

The *unit object* is defined as the empty sequence $\mathbf{0} := \langle\,\rangle$.

The symmetries are defined as $\sigma_{A,B} := \mathbf{S}_{\iota(B)}\mathrm{Id}_A + \mathrm{Id}_B\mathbf{S}_{-\iota(A)}$.

Note that any morphism from or to the unit object is necessarily 0.

Example III.5. Let us do a little example to fix the ideas: consider the objects $A := \langle 2, 1\rangle$ and $B := \langle 3\rangle$. We have $A \oplus B = \langle 2, 1, 3\rangle$ and

$$\begin{aligned}
\mathrm{Id}_A \oplus \mathrm{Id}_B &= \mathrm{Id}_A + \mathbf{S}_2\,\mathrm{Id}_B\,\mathbf{S}_{-2} \\
&= \mathbf{u}_{1[2]} \leftharpoonup \mathbf{u}_{1[2]} + \mathbf{u}_{2[1]} \leftharpoonup \mathbf{u}_{2[1]} + \mathbf{S}_2(\mathbf{u}_{1[3]} \leftharpoonup \mathbf{u}_{1[3]})\mathbf{S}_{-2} \\
&= \mathbf{u}_{1[2]} \leftharpoonup \mathbf{u}_{1[2]} + \mathbf{u}_{2[1]} \leftharpoonup \mathbf{u}_{2[1]} + \mathbf{u}_{3[3]} \leftharpoonup \mathbf{u}_{3[3]} \\
&= \mathrm{Id}_{A\oplus B}
\end{aligned}$$

Moreover we can compute

$$\begin{aligned}
\sigma_{A,B} &= \mathbf{S}_1\,\mathrm{Id}_A + \mathrm{Id}_B\mathbf{S}_{-2} \\
&= \mathbf{S}_1(\mathbf{u}_{1[2]} \leftharpoonup \mathbf{u}_{1[2]} + \mathbf{u}_{2[1]} \leftharpoonup \mathbf{u}_{2[1]}) + (\mathbf{u}_{1[3]} \leftharpoonup \mathbf{u}_{1[3]})\mathbf{S}_{-2} \\
&= \mathbf{u}_{2[2]} \leftharpoonup \mathbf{u}_{1[2]} + \mathbf{u}_{3[1]} \leftharpoonup \mathbf{u}_{2[1]} + \mathbf{u}_{1[3]} \leftharpoonup \mathbf{u}_{3[3]}
\end{aligned}$$

Also, we can see that in general we have

$$\begin{aligned}\sigma_{B,A}\sigma_{A,B} &= (\mathbf{S}_{\imath(A)}\mathrm{Id}_B + \mathrm{Id}_A\mathbf{S}_{-\imath(A)})(\mathbf{S}_{\imath(B)}\mathrm{Id}_A + \mathrm{Id}_B\mathbf{S}_{-\imath(A)})\\ &= \mathbf{S}_{\imath(A)}\mathrm{Id}_B\mathrm{Id}_B\mathbf{S}_{-\imath(A)} + \mathrm{Id}_A\mathbf{S}_{-\imath(A)}\mathbf{S}_{\imath(B)}\mathrm{Id}_A\\ &= \mathrm{Id}_A + \mathbf{S}_{\imath(A)}\mathrm{Id}_B\mathbf{S}_{-\imath(A)}\\ &= \mathrm{Id}_A \oplus \mathrm{Id}_B = \mathrm{Id}_{A\oplus B}\end{aligned}$$

To show that $\mathfrak{R}$ has a unique decomposition structure (as we already noted that the $\sum$-monoid structure and the distributivity property hold) we still need to give injections and projections (definition I.29) for any finite monoidal product.

Proposition III.6

Given a finite monoidal product $A_1 \oplus \cdots \oplus A_n$ and writing (with $l_1 := 0$) $l_i = \imath(A_1) + \cdots + \imath(A_{i-1})$, we have that the morphisms

- $\pi_i := \mathrm{Id}_{A_i}\,\mathbf{S}_{-l_i} \;:\; A_i \leftarrow A_1 \oplus \cdots \oplus A_n$
- $\iota_i := \mathbf{S}_{l_i}\,\mathrm{Id}_{A_i} \;:\; A_1 \oplus \cdots \oplus A_n \leftarrow A_i$

satisfy the following equations: $\begin{cases}\pi_i\iota_i = \mathrm{Id}_{A_i}\\ \pi_i\iota_j = 0 \text{ if } i \neq j\\ \sum_i \iota_i\pi_i = \mathrm{Id}_{A_1\oplus\cdots\oplus A_n}\end{cases}$

Example III.7. Carrying on the previous example, the unique decomposition structure of $A \oplus B$ would be given by the projections

$$\pi_A = \mathrm{Id}_A\mathbf{S}_0 = \mathrm{Id}_A \quad\text{and}\quad \pi_B = \mathrm{Id}_B\mathbf{S}_{-2} = (\mathbf{u}_{1[3]} \leftharpoonup \mathbf{u}_{1[3]})\mathbf{S}_{-2} = \mathbf{u}_{1[3]} \leftharpoonup \mathbf{u}_{3[3]}$$

and injections

$$\iota_A = \mathbf{S}_0\,\mathrm{Id}_A = \mathrm{Id}_A \quad\text{and}\quad \iota_B = \mathbf{S}_2\,\mathrm{Id}_B = \mathbf{S}_2(\mathbf{u}_{1[3]} \leftharpoonup \mathbf{u}_{1[3]}) = \mathbf{u}_{3[3]} \leftharpoonup \mathbf{u}_{1[3]}$$

We can check that indeed that $\pi_A\iota_A = \mathrm{Id}_A\mathrm{Id}_A = \mathrm{Id}_A$, $\pi_B\iota_B = \mathrm{Id}_B$ *etc.*

Corollary III.8

The resolution category $\mathfrak{R}$ has a unique decomposition structure.

Now, we saw in section I.3 that a unique decomposition category where all sums are defined automatically yields a traced category. As we are in such a configuration, we finally obtain the first ingredient of the GoI interpretation.

Corollary III.9

The resolution category $\mathfrak{R}$ has a traced structure, with $\mathbf{EX}(\cdot)$ (definition I.31) as a trace.

III.2 A GoI situation

We complete the proof that an interpretation of the dynamics of λ-CALCULUS can be built within $\mathcal{R}$ by showing that we can set up a GoI *situation* [HS06] in the category described above.

Remember from the first chapter that we need to find a traced functor $\mathbf{!}$ with a number of associated monoidal retractions and an object U, called a *reflexive object*, with its specific retraction pairs.

Let us first describe the traced functor. Informally, its action is to add a variable which allows for an interaction with different flows using the same interface.

Definition III.10 (promotion functor)

Let $l = \mathbf{u}_{n[i]}(t_1, \dots, t_i) \leftharpoonup \mathbf{u}_{m[j]}(u_1, \dots, u_j)$ be a clause. The *promotion* of l is defined as

$$\mathbf{!}l := \mathbf{u}_{n[i+1]}(t_1, \dots, t_i, y) \leftharpoonup \mathbf{u}_{m[j+1]}(u_1, \dots, u_j, y) \qquad \textit{(where y is a fresh variable)}$$

On objects, the functor $\mathbf{!}$ is defined as:

$$\mathbf{!}\langle i_1, \dots, i_n\rangle := \langle i_1 + 1, \dots, i_n + 1\rangle$$

Finally, if $F = \sum_i l_i$ is a morphism, then

$$\mathbf{!}F := \sum_i \mathbf{!}l_i$$

It is plain that this defines a functor. Its traced nature derives (by lemma I.34) from the fact that $\mathbf{!}\sum_i F_i = \sum_i \mathbf{!}F_i$ (obvious by the definition of $\mathbf{!}$), *i.e.* the functor is *additive*. This functor is also compatible with the adjoint operation induced by $\mathcal{R}$: for any F we have that $\mathbf{!}(F^\dagger) = (\mathbf{!}F)^\dagger$ whenever $F^\dagger$ is defined.

Definition III.11 (retraction pairs)

The promotion functor enjoys the following monoidal retraction pairs, parametrized by an object $A = \langle i_1, \dots, i_n\rangle$:

- The *digging* retraction $(\mathbf{T}_A, \mathbf{T}_A^\dagger) : \mathbf{!!}A \lhd \mathbf{!}A$ is given by

$$\begin{aligned}\mathbf{T}_A := \quad & \mathbf{u}_{1[i_1+2]}(x_1, \dots, x_{i_1+1}, x_{i_1+2}) \leftharpoonup \mathbf{u}_{1[i_1+1]}(x_1, \dots, x_{i_1+1} \bullet x_{i_1+2}) \\ + \quad & \cdots \\ + \quad & \mathbf{u}_{n[i_n+2]}(x_1, \dots, x_{i_n+1}, x_{i_n+2}) \leftharpoonup \mathbf{u}_{n[i_n+1]}(x_1, \dots, x_{i_n+1} \bullet x_{i_n+2})\end{aligned}$$

- The *dereliction* retraction $(\mathbf{D}_A, \mathbf{D}_A^\dagger) : A \lhd \mathbf{!}A$ is given by

$$\begin{aligned}\mathbf{D}_A := \quad & \mathbf{u}_{1[i_1]}(x_1, \dots, x_{i_1}) \leftharpoonup \mathbf{u}_{1[i_1+1]}(x_1, \dots, x_{i_1}, \star) \\ + \quad & \cdots \\ + \quad & \mathbf{u}_{n[i_n]}(x_1, \dots, x_{i_n}) \leftharpoonup \mathbf{u}_{n[i_n+1]}(x_1, \dots, x_{i_n}, \star)\end{aligned}$$

- The *contraction* retraction $(\mathbf{C}_A, \mathbf{C}_A^\dagger) : !A \oplus !A \lhd !A$ is given by the sum $\mathbf{C}_A := \mathbf{L}_A + \mathbf{S}_{\imath(A)}\mathbf{R}_A$ ($\mathbf{S}$ being the shift morphism from the previous section), where

$$\begin{array}{rll} \mathbf{L}_A := & & \mathbf{u}_{1[i_1]}(x_1, \dots, x_{i_1}) \leftharpoonup \mathbf{u}_{1[i_1]}(x_1, \dots, \mathtt{f}(x_{i_1})) \\ & + & \cdots \\ & + & \mathbf{u}_{n[i_n]}(x_1, \dots, x_{i_n}) \leftharpoonup \mathbf{u}_{n[i_n]}(x_1, \dots, \mathtt{f}(x_{i_n})) \end{array}$$

$$\begin{array}{rll} \mathbf{R}_A := & & \mathbf{u}_{1[i_1]}(x_1, \dots, x_{i_1}) \leftharpoonup \mathbf{u}_{1[i_1]}(x_1, \dots, \mathtt{g}(x_{i_1})) \\ & + & \cdots \\ & + & \mathbf{u}_{n[i_n]}(x_1, \dots, x_{i_n}) \leftharpoonup \mathbf{u}_{n[i_n]}(x_1, \dots, \mathtt{g}(x_{i_n})) \end{array}$$

- The *weakening* retraction $(\mathbf{W}_A, \mathbf{W}_A^\dagger) : \mathbf{0} \lhd !A$ is given by $\mathbf{W}_A := 0$.

These families of morphism are all *natural* in A, a tedious but straightforward verification which amounts to checking that for all $F : B \leftarrow A$ we have

- $!!F\,\mathbf{T}_A = \mathbf{T}_B\,!F$
- $F\,\mathbf{D}_A = \mathbf{D}_B\,!F$ (and the corresponding equations for the adjoints)
- $(!F \oplus !F)\,\mathbf{C}_A = \mathbf{C}_B\,!F$
- $0\,\mathbf{W}_A = \mathbf{W}_B\,!F$

which is reminiscent of the equations of dynamic algebras (definition I.14). Note that the naturality of the second members of each pairs (when the morphism F has an adjoint, otherwise some further computation is needed) derives from their being adjoints and the compatibility of $!$ and $(\cdot)^\dagger$.

For instance, $\mathbf{T}_A^\dagger !!F = (!!F^\dagger \mathbf{T}_A)^\dagger = (\mathbf{T}_B\,!F^\dagger)^\dagger = !F\,\mathbf{T}_B^\dagger$. Here we take advantage of the extra *dagger structure* [Sel07] we have in our category.

The fact that they form retraction pairs is immediate. The monoidality property simply amounts at the equations $\mathbf{T}_{A\oplus B} = \mathbf{T}_A \oplus \mathbf{T}_B$, $\mathbf{D}_{A\oplus B} = \mathbf{D}_A \oplus \mathbf{D}_B$, $\mathbf{C}_{A\oplus B} = \mathbf{C}_A \oplus \mathbf{C}_B$ and $\mathbf{W}_{A\oplus B} = \mathbf{W}_A \oplus \mathbf{W}_B$, and their adjoint version.

Finally, we take as our reflexive object the image of some object by the functor $!$, following remark I.24. We avoid the object $\mathbf{0}$, as we have equations $!\mathbf{0} = \mathbf{0}$ and $\mathbf{0} \oplus \mathbf{0} = \mathbf{0}$ and the fact that a morphism from or to the object $\mathbf{0}$ is necessarily 0, this choice would lead to a trivial interpretation.

Proposition III.12 (reflexive object)

The object $\langle 1 \rangle$ ($= !\langle 0 \rangle$) is a reflexive object, with retractions:

- $(\mathbf{C}_{\langle 0 \rangle}, \mathbf{C}_{\langle 0 \rangle}^\dagger) : \langle 1 \rangle \oplus \langle 1 \rangle \lhd \langle 1 \rangle$
- $(\mathbf{W}_{\langle 0 \rangle}, \mathbf{W}_{\langle 0 \rangle}^\dagger) : \mathbf{0} \lhd \langle 1 \rangle$
- $(\mathbf{T}_{\langle 0 \rangle}, \mathbf{T}_{\langle 0 \rangle}^\dagger) : !\langle 1 \rangle \lhd \langle 1 \rangle$

We obtain therefore our GoI situation.

Theorem III.13

The triple $(\mathfrak{R}, \mathbf{!}, \langle 1 \rangle)$ is a GoI situation.

Remark III.14. Pursuing on remark I.25, let us say a word about the type of wirings we obtain interpreting λ-CALCULUS. Indeed, the only part of $\mathfrak{R}$ actually used in the interpretation comes from combinations of identity, symmetry and retraction morphisms and their images through the trace operation and the $\mathbf{!}$ and $\oplus$ functors.

It is easy to see that this yields only wirings that are isometries (definition II.19) and use only the constant symbol $\star$, the two unary symbols $\mathtt{f}, \mathtt{g}$ and the binary symbol $\bullet$. Note that this relies on the fact that isometries do compose.

Moreover, the interpretation of linear λ-CALCULUS makes no use of the $\mathbf{!}$ functor and relies only on $\left(\mathbf{C}_{\langle 0 \rangle}, \mathbf{C}^{\dagger}_{\langle 0 \rangle}\right) : \langle 1 \rangle \oplus \langle 1 \rangle \lhd \langle 1 \rangle$ among the retractions. It is not hard to see that this would yield only elements of the $\mathcal{S}tack$ semiring.

III.3 GoI and logic programming

We now discuss briefly the relation between the GoI construction in the resolution semiring and logic programming that follows from the categorical structure we exposed.

Functional programs and logic programs

The morphisms of the category we defined in the above sections can be looked at as a particular type of logic programs, where clauses are required to be safe (this is the condition on variables from definition II.1) and have exactly one atom in the body. Moreover, from that perspective the product of wirings corresponds to the resolution rule as we already remarked in section II.1.

Let us now consider the fixpoint semantics [DEGV01] of logic programs, of which we can give a simplified definition in our restricted case.

Definition III.15 (fixpoint)

Let F be a clause wiring and $\mathbf{U}$ a set of facts. The *consequence operator* $\mathbf{C}_F$ of F acts on sets of facts the following way

$$\mathbf{C}_F(\mathbf{U}) := \{ \mathbf{u} \mid \mathbf{u} \in F\mathbf{U} \text{ or } \mathbf{u} \in \mathbf{U} \}$$

and the *fixpoint operator* $\mathbf{C}^{\infty}_F$ of F acts on facts the following way:

$$\mathbf{C}^{\infty}_F(\mathbf{U}) := \bigcup_{n \in \mathbb{N}} \mathbf{C}^{n}_F(\mathbf{U}) \qquad \textit{(where } \mathbf{C}^{n}_F \textit{ denotes the } n^{th} \textit{ iteration of } \mathbf{C}_F\textit{)}$$

We formulated it this way to keep close to the more usual definition of this notion in logic programming, but it should be quite clear that we already encountered them:

- The consequence operator is simply a multiplication: $\mathbf{C}_F(\mathbf{U}) = (I + F)\mathbf{U}$.
- The fixpoint operator is multiplication by the iteration: $\mathbf{C}_F^\infty(\mathbf{U}) = \mathbf{IT}(F)\mathbf{U}$ (remember definition I.31).

This establishes a link between iteration/execution of wirings and the fixpoint semantics of logic programs. The results the previous chapter yield therefore a translation from λ-CALCULUS to logic programs where the β-reduction corresponds to the fixpoint semantics of the program: a sort of Curry-Howard correspondence involving logic programming.

This connexion was already considered in early work on the subject [Gir89b, Gir95a] but still needs to be explored, especially in view of complexity results which will be the subject of the next chapter.

Nilpotency

The boundedness property for logic programs states that the iteration of the consequence operator described above eventually reaches its fixpoint after a finite number of iterations, independently of the set of facts it is acting on.

Definition III.16 (boundedness)

A wiring is *bounded* (of *rank* k) if there is an integer k such that for any set of facts $\mathbf{U}$, we have $\mathbf{C}_F^\infty(\mathbf{U}) = \mathbf{C}_F^k\mathbf{U}$.

Nilpotency obviously implies boundedness. Note that the converse is not true: the unit is not nilpotent since $I^n = I$ for all n, but it is bounded.

Anyway, we can use these remarks to transport results between proof theory and logic programming. For instance, consider the nilpotency result for the interpretation of SYSTEM F, the polymorphic λ-CALCULUS [GLT89]:

Theorem III.17 (nilpotency theorem[Gir89a, theorem 1])

If t is a λ-term typable in SYSTEM F, then its GoI interpretation has a finite execution (definition I.31).

This applies to any unique decomposition category with a GoI situation, and therefore we can read this result as a way to translate λ-terms of SYSTEM F as nilpotent (thus bounded) logic programs of the form described in remark III.14, while boundedness is a property that is usually difficult to guarantee for logic programs using function symbols.

More generally we have the following result by V. Danos and L. Regnier:

Theorem III.18 (nilpotency and strong normalization [DR95, theorem 3])

Let t be a λ-term, the GoI interpretation of t has a finite execution *iff.* t is strongly normalizing.

This also applies to any unique decomposition category with a GoI situation and leads to an undecidability result for the nilpotency problem in $\mathcal{R}$, and more precisely for the aforementioned restricted class of logic programs.

Corollary III.19 (undecidability of nilpotency)

The nilpotency problem for wirings is undecidable.

It remains undecidable if we restrict to clause wirings (definition III.1) that are isometries (definition II.19) and use only a constant symbol, two unary and one binary function symbols.

Indeed, the GoI interpretation yields only wirings of the required form and Theorem III.18 tells us a term t is strongly normalizing *iff.* some wiring is nilpotent. Therefore a procedure deciding the nilpotency problem would yield a procedure deciding the strong normalization of λ-terms while this problem is well known to be undecidable [Urz03].

This result may be compared to other undecidability results for the boundedness problem in logic programming [DEGV01, Bla82, Fit87].

In the next chapter we are going to build a setting for capturing complexity classes, using nilpotency as an acceptance condition. In order to make this problem tractable, we see already that we will need to restrict to specific classes of wirings, and that for instance the linearity constraint is certainly not enough in that perspective.

Chapter IV

Complexity

The aim of implicit computational complexity theory [DL12] is to give characterizations of complexity classes without any reference to cost bounds, for instance by a type system of a restricted recursion scheme.

The last two decades have seen various works relating proof theory and implicit computational complexity [Bai08], the basic idea being to look for restricted fragments of linear logic with an expressiveness that corresponds exactly to some complexity class. Various syntactic restrictions, most often concerning the rules of the exponential modalities of linear logic, produced systems with a less complex cut-elimination procedure. Also, a study of elementary complexity in terms of the resolution algebra had already been pursued [BP01].

More specifically, let us mention classical work on implicit characterizations of the complexity classes we study in this thesis:

As for Ptime, characterizations have been obtained as restrictions on recursion [BC92, Lei93] and *via* a simply typed λ-calculus manipulating words [LM93]. On the proof-theoretic side, the original article by J.-Y. Girard on light linear logic [Gir95b] initiated the investigation of the expressive power of fragments of linear logic and λ-calculus with linear types, leading to characterizations of polynomial time [Laf04, BT04].

Concerning Logspace, a characterization in terms of restricted recursion [Nee04] has been given, while the proof-theoretic approach of the question relies on a bidirectional view of computation [Sch07, LS10] to represent the composition of logarithmic space programs, which is related to the geometry of interaction view of the dynamics of logic.

The complexity of other problems concerning linear logic has also been investigated, including the cut-elimination problem [MT03]: "given two proofnets, are they equivalent *modulo* the cut-elimination procedure?" In the case of multiplicative linear logic (or equivalently linear λ-calculus) this problem has been proven to be Ptime-complete [Mai04]. This result was actually the intuitive starting point of our characterization of Ptime in section IV.4: by remark III.14, we know the GoI interpretation of linear λ-calculus can be done in the semiring $\mathcal{S}tack$ and it appears therefore natural to wonder if this semiring can be related to polynomial time in our approach.

In this last chapter, we characterize complexity classes in terms of restricted semirings of $\mathcal{R}$. This approach relates naturally to the work on implicit computational complexity mentioned above *via* the GoI interpretation of linear logic, although this connexion remains mostly to be explored.

We will set up in the first two sections the general framework in which our characterizations take place: we expose first the representation of data we use, which is inspired by the representation of words in linear logic, then the notion of acceptance of an input by an observation, the counterpart of a program in our construction.

The last two sections are devoted to the characterization of the complexity classes Logspace, NLogspace, and Ptime, making use of the results of section II.3 and section II.4 on nilpotency in the semirings $\mathcal{R}_b$ and $\mathcal{S}tack$. The completeness part of these result rely on an encoding of devices we call *pointer machines* and *stack machines* which are classical characterizations of the complexity classes we consider, characterizing them by the type of memory involved.

Contents

IV.1 Representation of words

We begin by setting up the framework we will use to capture complexity classes in the resolution semiring.

We will define the notion of representation of words as wirings which is inspired from the representation of words in linear logic and their image through the GoI interpretation. The notion of *observation* will be given in the next section. It constitutes the counterpart of programs accepting and rejecting words in the construction.

Note however that this distinction data *vs.* program is different than the one usually assumed in logic programming: the representation of words is itself a program which *interacts* with another program, the observation. This computation model will prove particularly suitable for capturing logarithmic space complexity in section IV.3 as the amount of space needed to perform the computation will correspond to the size of the "messages" (indeed closed terms) exchanged between the two programs.

The construction relies on two semirings: one for word representations, one for observations. The choice of these parameters sets the operations that can be performed by the programs, and they will be required to satisfy some disjointness property (see section IV.2).

The following construction will be used to give a very basic common formatting to these objects.

Notation

We fix from now on two constant symbols $\mathtt{l}, \mathtt{r}$, the set $\mathtt{lr} := \{\mathtt{l}, \mathtt{r}\}$ and an infinite set $\mathbf{P}$ of constant symbols which we call *position constants* and a unary function symbol $\textsc{head}$. We write $\textsc{head}(\mathbf{P})$ the set of terms of the form $\textsc{head}(\mathtt{p})$ with $\mathtt{p} \in \mathbf{P}$.

Definition IV.1

Given an alphabet Σ and semirings $\mathcal{A}$ and $\mathcal{B}$ we define, using definition II.23 and definition II.26, the semiring

$$\mathcal{M}_\Sigma(\mathcal{A}, \mathcal{B}) := (\Sigma \cup \{\star\})^{\leftrightharpoons} \bullet \mathtt{lr}^{\leftrightharpoons} \bullet \mathcal{A} \bullet \mathcal{B}$$

We can then fix a notion of representation of words that will serve for the characterizations of Logspace, NLogspace and Ptime in the next sections.

These representations are elements of the semiring

$$\mathcal{M}_\Sigma\left(\mathcal{I}, \mathcal{I} \bullet \textsc{head}(\mathbf{P})^{\leftrightharpoons}\right)$$

so that the interaction between the observation and the word representation will really take place in the second parameter of $\mathcal{M}_\Sigma(\cdot, \cdot)$, the first parameter being for internal operations of the observation on which the word representation acts trivially.

Notation

We write $u \leftrightharpoons v$ for $u \leftharpoonup v + v \leftharpoonup u$.

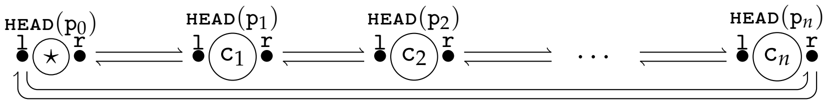
Definition IV.2 (word representation)

Let $W = \mathrm{c}_1 \dots \mathrm{c}_n$ be a word over an alphabet Σ and $\vec{\mathrm{p}} = \mathrm{p}_0, \mathrm{p}_1, \dots, \mathrm{p}_n$ be pairwise distinct elements of $\mathbf{P}$.

The *representation* of W associated with $\vec{\mathrm{p}}$ is the following element of the semiring $\mathcal{M}_\Sigma(\mathcal{I}, \mathcal{I}\bullet\texttt{HEAD}(\mathbf{P})^{\leftrightharpoons})$

$$\begin{array}{rrl} W[\vec{\mathrm{p}}] := & & \star\bullet\texttt{r}\bullet x\bullet y\bullet\texttt{HEAD}(\mathrm{p}_0) \leftrightharpoons \mathrm{c}_1\bullet\texttt{l}\bullet x\bullet y\bullet\texttt{HEAD}(\mathrm{p}_1) \\ & + & \mathrm{c}_1\bullet\texttt{r}\bullet x\bullet y\bullet\texttt{HEAD}(\mathrm{p}_1) \leftrightharpoons \mathrm{c}_2\bullet\texttt{l}\bullet x\bullet y\bullet\texttt{HEAD}(\mathrm{p}_2) \\ & + & \dots \\ & + & \mathrm{c}_n\bullet\texttt{r}\bullet x\bullet y\bullet\texttt{HEAD}(\mathrm{p}_n) \leftrightharpoons \star\bullet\texttt{l}\bullet x\bullet y\bullet\texttt{HEAD}(\mathrm{p}_0) \end{array}$$

This can be summarized in the following picture, which is reminiscent of a representation of words as proofnets.



Another intuition, that will be at use in the completeness proofs of section IV.3 and section IV.4, is to see the flows of the sum in definition IV.2 as the description of transitions from a configuration to another in a computation made by some kind of automaton. From this point of view, the term $\mathrm{c}\bullet\texttt{l}\bullet s\bullet m\bullet\texttt{HEAD}(\mathbf{P})$ is to be understood as:

- c is the symbol that is read by the reading head of the machine.
- $\texttt{l}$ (resp. $\texttt{r}$) tells the direction of the next move of the reading head.
- s describes the internal state the machine is in.
- m describes the memory of the machine.
- $\texttt{HEAD}(\mathrm{p})$ gives the position of the reading head of the machine.

Remember we said that the word representation is itself a program *interacting* with the observation. The role it has in this interaction, seen as the computation an automaton, can be understood as follows: the word representation is *moving the head*, providing a new symbol and a new position according to the previous symbol and the direction in which it is asked to move the head. In a flow

$$\mathrm{c}\bullet\texttt{l}\bullet x\bullet y\bullet\texttt{HEAD}(\mathrm{p}) \leftharpoonup \mathrm{c}'\bullet\texttt{r}\bullet x\bullet y\bullet\texttt{HEAD}(\mathrm{p}')$$

we start from a situation where the reading head was at position p', reading the symbol c' and we go to (moving the reading head to the right) to a situation where the reading head is at position $\mathtt{p}$, reading the symbol $\mathtt{c}$. The variables x and y account for the fact that the word representation does neither modify the internal state nor the memory of the abstract machine.

Note that word representations are balanced (definition II.29) wirings, with fixed height, fixed arity and use a number of symbols proportional to the length of the word they represent. We get, as a consequence of lemma II.17, that they are also deterministic. Let us summarize this in a lemma:

Lemma IV.3

Let W be a word of length n and $\vec{\mathtt{p}}$ be n distinct position constants. Let $W[\vec{\mathtt{p}}]$ be the associated representation. We have:

- $W[\vec{\mathtt{p}}]$ is deterministic (definition II.15).
- $\mathcal{M}_\Sigma(\mathcal{I}, \mathcal{I} \bullet \mathtt{HEAD}(\mathbf{P})^{\leftrightharpoons}) \subseteq \mathcal{R}_{\mathbf{b}}$, therefore $W[\vec{\mathtt{p}}]$ is balanced.
- Its height (definition II.12) is $h(W[\vec{\mathtt{p}}]) = 5$.
- It uses $n+5$ distinct symbols ($n+1$ position constants, together with the $\bullet, \mathtt{l}, \mathtt{r}, \mathtt{HEAD}$ symbols) with maximal arity $A = 2$.

IV.2 Acceptance and normativity

The goal of this section is to define the acceptance condition for observations. The basic idea is to say that an observation (definition IV.4 below) O accepts a word W if $OW[\vec{\mathtt{p}}]$ is nilpotent for some representation of the word.

However, we need to make sure that the nilpotency of $OW[\vec{\mathtt{p}}]$ does not depend on $\vec{\mathtt{p}}$ if we want the notion to be well defined: acceptance should not depend on the specific choice of a representation of W. This leads to the notion of *normativity* introduced by J.-Y. Girard [Gir12].

In short, one should ensure that the choice of semirings in which representations and observations live are sufficiently “disjoint” to ensure the aforementioned independence property. In order to state it, let us introduce the two observation semirings we will consider in the next sections.

Definition IV.4 (observation semirings)

We fix an infinite set of *state constants* $\mathbf{S}$ that we suppose disjoint from the set of position constants.

A *balanced observation* is a finite element of the semiring

$$\mathcal{O}_\Sigma^{\mathbf{b}} := \mathcal{M}_\Sigma\left(\mathbf{S}^{\leftrightharpoons}, {\mathcal{R}_{\mathbf{b}}}^{\setminus \mathbf{P}}\right)$$

An *observation with stack* is a finite element of the semiring

$$\mathcal{O}_\Sigma^{\mathbf{s}} := \mathcal{M}_\Sigma\left(\mathbf{S}^{\leftrightharpoons} \bullet \mathcal{S}tack, {\mathcal{R}_{\mathbf{b}}}^{\setminus \mathbf{P}}\right)$$

As we already said, the first parameters of $\mathcal{M}_\Sigma(\cdot, \cdot)$ concerns internal operations of the observation, while the second corresponds to the part that interacts with the input.

The fact that second parameters of these two constructions are equal can be related to the automaton intuition we will use in the completeness proofs of the next sections: the interaction with the input can be understood as a reading head that can move on the input together with a number of read-only pointers that can store positions. This is algebraically captured by the balanced semiring as already evoked in remark II.31.

On the other hand, the first parameters differs and this corresponds to the fact that the internal operations allowed by the semirings differ: $\mathcal{O}^{\mathbf{b}}_{\Sigma}$ allows only a (finite) set of states while $\mathcal{O}^{\mathbf{s}}_{\Sigma}$ adds the possibility to store information in a pushdown store. This will result into a different level of expressivity, as we shall see in the next sections.

Anyway, before we turn to this, we need a well defined notion of acceptance which is provided by the following theorem.

Theorem IV.5 (normativity)

Let W be a word on an alphabet Σ, O a balanced observation. Then if $OW[\vec{\mathrm{p}}]$ (definition IV.2) is nilpotent for some $\vec{\mathrm{p}}$, it is nilpotent for any $\vec{\mathrm{p}}$.

The same holds when O is an observation with stack.

This is due to the fact that observations cannot use the position constants, which avoids any interference of the choice of constants into the computation.

Before we prove this theorem, let us use it to define a notion of acceptance and rejection.

Definition IV.6 (language of an observation)

Let O be a balanced observation (resp. observation with stack), the *language recognized* by O is the set of words

$$\mathcal{L}(O) := \{\, W \mid OW[\vec{\mathrm{p}}] \text{ is nilpotent for any } \vec{\mathrm{p}} \,\}$$

For instance, the observation 0 will accept any word, while the observation I will refuse any word because $W[\vec{\mathrm{p}}]$ is never nilpotent and $IW[\vec{\mathrm{p}}] = W[\vec{\mathrm{p}}]$. However, the notion of acceptance is undecidable *a priori* as we saw in the end of section III.3.

With these definitions at hand, we will be able to state our complexity results, relating type of observations and complexity classes.

Proof (of theorem IV.5) ▸ Let O be an observation in either $\mathcal{O}^{\mathbf{b}}_{\Sigma}$ or $\mathcal{O}^{\mathbf{s}}_{\Sigma}$, $W[\vec{\mathrm{p}}]$ and $W[\vec{\mathrm{q}}]$ be two representations of the same word. We define φ the function from terms to terms that replaces any occurrence of a p_i by its corresponding q_i, and conversely. It extends naturally to flows by $\varphi(t \leftharpoondown u) := \varphi(t) \leftharpoondown \varphi(u)$ and then to wirings by linearity.

This function φ is such that $\varphi(FG) = \varphi(F)\varphi(G)$ for all F, G because replacing constant symbols in a term before or after performing their unification yields the same result. It is also a bijective function, and this means in particular that $\varphi(F) = 0$ *iff.* $F = 0$.

Now, note that on elements of $\mathcal{O}^{\mathbf{b}}_{\Sigma}$ and $\mathcal{O}^{\mathbf{s}}_{\Sigma}$, φ acts as the identity because these cannot use any of the symbols in $\mathbf{P}$, in particular we have $\varphi(O) = O$. Moreover φ was defined so that $\varphi(W[\vec{\mathrm{p}}]) = W[\vec{\mathrm{q}}]$.

Let us consider the product $OW[\vec{\mathrm{q}}]$, we have

$$(OW[\vec{\mathrm{q}}])^n = \big(\varphi(O)\varphi(W[\vec{\mathrm{p}}])\big)^n = \big(\varphi(OW[\vec{\mathrm{p}}])\big)^n = \varphi\big((OW[\vec{\mathrm{p}}])^n\big)$$

by the properties of φ we saw above.

Then, as φ is bijective, we have that $(OW[\vec{\mathrm{q}}])^n = 0$ *iff.* $(OW[\vec{\mathrm{p}}])^n = 0$, that is to say $OW[\vec{\mathrm{q}}]$ is nilpotent *iff.* $OW[\vec{\mathrm{p}}]$ is. ◀

IV.3 Logarithmic space

This section is devoted to the proof that balanced observations correspond to logarithmic space computation, either deterministic or not depending on the determinism (definition II.15) of the observation.

Completeness

We begin by expanding the remarks on automata of section IV.1 to the point where it will give us a lower bound for the expressivity of balanced wirings. Indeed, by borrowing some classical results on *two-ways multihead finite automata* (we will consider a variant which we will call *pointer machines*) we will see that balanced observations can decide any logarithmic space problem:

Theorem IV.7

If $L \in$ coNLogspace, then there is a balanced observation $O \in \mathcal{O}^{\mathbf{b}}_{\Sigma}$ such that $\mathcal{L}(O) = L$. Moreover, if $L \in$ Logspace then O can be chosen deterministic.

Pointer machines. It is a classical result in automata theory that two-ways multihead finite automata characterize logarithmic space computation [Har72, WW86]. This model of computation is tolerant to a lot of modifications [Pig13] that do not affect the class of languages it captures (though simulations may cost an explosion of the number of heads and states) and thus many variant of it have been defined.

We will call a *pointer machine* an automaton with:

- A reading head that can move both ways and comes back to the beginning of the input when reaching its end.
- A finite number of states.
- A finite number of auxiliary pointers that can store positions on the input.

which is just another variation of the same model, therefore characterizing logarithmic space computation, (non-)deterministic automata corresponding to (non-)deterministic logarithmic space. Such a machine is said to be *deterministic* if its transition relation turns out to be the graph of a partial function.

Theorem IV.8 (pointer machines [WW86, theorem 13.2])

If $L \in$ NLogspace then there is a pointer machine M that recognizes L.

If $L \in$ Logspace then M can be chosen deterministic.

Encoding as balanced observations. We already explained in section IV.1 that once we read a term

$$\mathtt{c} \bullet \mathtt{l}/\mathtt{r} \bullet s \bullet m \bullet \mathtt{HEAD}(\mathtt{p})$$

as the description of a configuration of an automaton, the action of $W[\vec{\mathtt{p}}]$ can be understood as "moving the reading head in the asked direction". The s and m part of this term can be used by the observation to perform various operations.

If in place of m we have terms of the form $\mathtt{AUX}_n(\mathtt{p_1}, \dots, \mathtt{p_n})$ that are understood as the positions of n *auxiliary pointers*. Then, the following flow:

$$\cdots \bullet \mathtt{AUX}_n(x_1, \dots, x_n) \bullet \mathtt{HEAD}(x) \leftharpoonup \cdots \bullet \mathtt{AUX}_n(y_1, \dots, y_n) \bullet \mathtt{HEAD}(y)$$

(with $\{x_1, \dots, x_n, x\} \subseteq \{y_1, \dots, y_n, y\}$ to respect the safety condition) implements a transition where the position of the auxiliary pointers and the reading head may be rearranged. In particular $y_i = y_j$ would require that the two pointers are at the same position to perform the transition, while $x_i = x_j$ would equate the positions of the two pointers after the transition.

For instance, the flow

$$\cdots \bullet \mathtt{AUX}_n(x, \dots, x) \bullet \mathtt{HEAD}(x) \leftharpoonup \cdots \bullet \mathtt{AUX}_n(y_1, \dots, y_n) \bullet \mathtt{HEAD}(x)$$

corresponds to a transition where all the pointers are moved to the position of the reading head, no matter what their position was before.

If moreover in place of s we have state constants from the set $\mathbf{S}$ we can implement state change as

$$\cdots \bullet \mathtt{s}' \bullet \cdots \leftharpoonup \cdots \bullet \mathtt{s} \bullet \cdots$$

The control over the direction of the reading head can be implemented as

$$\cdots \bullet \mathtt{l} \bullet \cdots \leftharpoonup \cdots \bullet \mathtt{r} \bullet \cdots$$

$$\cdots \bullet \mathtt{r} \bullet \cdots \leftharpoonup \cdots \bullet \mathtt{l} \bullet \cdots$$

With these ideas at hand, it is easy to encode the transitions of a pointer machine M as a balanced observation O_M which is deterministic when M is.

Acceptance. The nilpotency of $O_M W[\vec{\mathtt{p}}]$ is then equivalent to the absence of non-terminating sequence of transitions when starting from any configuration. This quite odd acceptance condition turns out to be equivalent to more usual ones (with initial, accepting and rejecting states for instance) by translating rejection as reinitialization, and acceptance as stopping computation [AS14, Aub13, Sei12]. Moreover, the fact that the result on pointer machines is formulated with the class NLogspace is not problematic in view of the classical result NLogspace =coNLogspace [Imm88, Sze88].

Theorem IV.9 (encoding)

Any pointer machine M can be encoded as a balanced observation $O_M \in \mathcal{O}^{\mathbf{b}}_{\Sigma}$ in a way that $\mathcal{L}(M) = \mathcal{L}(O_M)$ and O_M is deterministic *iff.* M is deterministic.

Combining theorem IV.9 and theorem IV.8, we get the expected result.

Soundness

We now use the results of section II.3 to design a procedure that decides whether a word belongs to the language recognized by a balanced observation within logarithmic space.[1] This procedure relies on a *simulation* principle, reducing the problem to the acyclicity of a graph using theorem II.40: we are not going to compute the iterations of $OW[\vec{\mathtt{p}}]$ until we eventually reach 0, which would require too much space.

We first show that the computation graph (definition II.39) of the product $OW[\vec{\mathtt{p}}]$ can be constructed by a deterministic procedure using only logarithmic space.

Then, we show that testing the acyclicity of such a graph can be done within the same bounds. Here the procedure will be deterministic or not depending on the shape of the graph (which is itself affected by the determinism of the wiring, recall lemma II.41).

Since logarithmic space algorithms do compose,[2] we will obtain the expected result:

Theorem IV.10

If $O \in \mathcal{O}^{\mathbf{b}}_{\Sigma}$ is a balanced observation, then $\mathcal{L}(O) \in$ coNLogspace.

If moreover O is deterministic, then $\mathcal{L}(O) \in$ Logspace.

[1] When not stated explicitly, when we write “logarithmic space” it should read “logarithmic space in the length of the input”.

[2] Which is a classical, though quite non-trivial, result [Sav98, Fig. 8.10].

We know by lemma IV.3 that any $W[\vec{\mathrm{p}}]$ is balanced and that its height and maximal arity of function symbol do not depend on W. Therefore the product $OW[\vec{\mathrm{p}}]$ with a balanced observation is still balanced (lemma II.32) and its height and maximal arity do not vary when W does.

Building the computation graph. Given a word W of length n, building a representation $W[\vec{\mathrm{p}}]$ is doable within logarithmic space: after the discussion of section IV.2, we know that the choice of $\vec{\mathrm{p}}$ is irrelevant to the outcome of the computation, and we can safely choose $\mathrm{p}_1, \dots, \mathrm{p}_n$, identified by their index that can be stored within logarithmic space with a binary encoding; then, each flow in $W[\vec{\mathrm{p}}]$ depends only on two consecutive symbols of W, so we can build them scanning W locally. We have therefore a function $\textsc{Rep}(\cdot) \in \mathrm{F}\textsc{Logspace}$ that inputs a word W and outputs a representation $W[\vec{\mathrm{p}}]$.

Now, to build the computation graph (definition II.39) of $OW[\vec{\mathrm{p}}]$ we need to: first, enumerate the edges (*i.e.* the elements of the computation space $\mathbf{Comp}(OW[\vec{\mathrm{p}}])$, Definition II.36). Second, determine whether there is an edge between two vertices.

The elements of $\mathbf{Comp}(OW[\vec{\mathrm{p}}])$ are trees of height and maximal arity bounded by integers that do not depend on W, so there is a fixed number of tree shapes they can take. These trees are in turn labeled by symbols stored as integers ranging from 1 to S and S is linearly growing with the length of W.

This sets the stage for a enumeration within logarithmic space, so that we have a function $\textsc{Comp}_O(\cdot) \in \mathrm{F}\textsc{Logspace}$ that inputs a word W and outputs the list of the elements of $\mathbf{Comp}(OW[\vec{\mathrm{p}}])$. Finally, remember that the matching problem can be solved within logarithmic space (theorem I.13), which yields a function $\textsc{Match}(\cdot,\cdot) \in \mathrm{F}\textsc{Logspace}$ that inputs a flow f and a fact $\mathbf{u}$ and outputs the fact $f\mathbf{u}$.

Combining all these elements, we get a procedure that builds the computation graph within logarithmic space.

Lemma IV.11 (computation graph in $\mathrm{F}\textsc{Logspace}$)

Given a balanced observation O, there is a function $\textsc{Graph}_O(\cdot) \in \mathrm{F}\textsc{Logspace}$ that inputs a word W and outputs $\mathbf{G}(OW[\vec{\mathrm{p}}])$.

We can then end the proof of the main theorem of this section.

Proof (of theorem IV.10) ▸ Let O be a balanced observation. It is a classic result of complexity theory [Jon75, p. 83] that the cycle search on directed graphs is in $\mathrm{N}\textsc{Logspace}$, that is: there is a non-deterministic logarithmic space procedure $\textsc{Cycl}(\cdot)$ that inputs a directed graph and accepts *iff.* it has a cycle.

We can briefly describe it as follows: to check whether there is a cycle starting at some vertex v, one can non-deterministically explore the graph (starting by v) remembering only a current vertex v'; at each step choosing an edge that has source v', going through this edge and checking whether we reached v, otherwise updating v'. A counter remembering the number of steps is also needed, to detect a cycle if we were able to do more steps than the number of vertices in the graph. This procedure has then to be run starting at any v, in case the graph is not connected.

Therefore, since $\textsc{Cycl} \circ \textsc{Graph}_O(\cdot)$ accepts a word W whenever O rejects W (theorem II.40), we have that $\mathcal{L}(O) \in \textsc{coNLogspace}$.

Moreover, it is quite clear that with the additional assumption that the graph has an out-degree bounded by 1, the $\textsc{Cycl}(\cdot)$ procedure becomes deterministic because there is no longer any choice to be made when deciding which is the next edge to follow. In case O is deterministic, $OW[\vec{\mathrm{p}}]$ is also deterministic (by lemma IV.3 and remark II.16) and therefore $\mathbf{Comp}(OW[\vec{\mathrm{p}}])$ has an out-degree bounded by 1 (lemma II.41). Then, $\textsc{Cycl} \circ \textsc{Graph}_O(\cdot)$ is a deterministic procedure and $\mathcal{L}(O) \in \textsc{Logspace}$. ◂

The same argumentation as above in the case of a plain balanced wiring without considering word representations and observations would give a complexity result for the nilpotency problem of these wirings which we state independently.

Theorem IV.12 (nilpotency in $\mathcal{R}_\mathbf{b}$)

Given two integers A, h, there is a procedure $\text{BN}\textsc{ilp}_{A,h}(\cdot) \in \textsc{coNLogspace}$ that inputs a balanced wiring F built with function symbol of arity at most A and such that $\hbar(F) \leq h$ and accepts *iff.* F is nilpotent.

If we restrict to deterministic elements of $\mathcal{R}_\mathbf{b}$, then $\text{BN}\textsc{ilp}_{A,h}(\cdot) \in \textsc{Logspace}$.

In this case, we mean logarithmic space in the *size* of F, defined as the total number of symbols in it.

IV.4 Polynomial Time

We turn now to the study of polynomial time computation and show that it is captured by observations with stack.

Here also

Completeness

We follow the pattern of the previous section and use a well-known type of automata that capture polynomial time computation: *auxiliary pushdown automata*, which we will call *stack machines*.

We will see in this section that observations with stack can decide any $\textsc{Ptime}$ language:

Theorem IV.13

If $L \in$ PTIME, then there exist an observation with stack $O \in \mathcal{O}^{\mathbf{s}}_{\Sigma}$ such that $\mathcal{L}(O) = L$.

Stack machines. S. Cook [Coo71] was one of the first to explore the expressivity of automata equipped with a stack (or "pushdown store") together with either a logarithmically bounded tape or a fixed number of pointers.

We will call a *stack machine* an automaton with

- A reading head that can move both ways and comes back to the beginning of the input when reaching its end.
- A finite number of states.
- A finite number of auxiliary pointers that can store positions on the input.
- A pushdown stack.

These machines work the same way as pointer machines to which a stack would have been added, together with the usual "push, pop" operations: they can manipulate the stack to add or remove symbols at the top of it.

Cook's results and its various later reformulations have as a consequence that these machines characterize polynomial time, which is now part of the classical theorems in complexity theory.

Theorem IV.14 ([WW86, theorem 13.20])

If $L \in$ PTIME, then there is a stack machine M that recognizes L.

Extending the encoding. We only need to show how we can extend the encoding of the previous section to handle the addition of a stack to our model. The flows $\texttt{PUSH}_{\texttt{f}}, \texttt{POP}_{\texttt{f}}$ we saw in section II.4 turn out to be exactly what we need to do this. Compared to the above section, the only thing that changes is that in a term

$$\texttt{c} \bullet \texttt{l}/\texttt{r} \bullet s \bullet m \bullet \texttt{HEAD}(\texttt{p})$$

the state s of the machine will no longer be represented simply as a state constant $\texttt{s}$, but as a pair of a state constant and a stack $\texttt{s} \bullet \tau(x)$.

We can then use the elements of the semiring $\mathcal{S}tack$ to encode stack-related transitions. For instance a "push $\texttt{f}$" (without changing the state $\texttt{s}$) operation would be implemented as

$$\cdots \bullet (\texttt{s} \bullet \texttt{f}(x)) \bullet \cdots \leftharpoonup \cdots \bullet (\texttt{s} \bullet x) \bullet \cdots$$

while a "pop $\texttt{f}$" (again, without changing the state $\texttt{s}$) would be implemented as

$$\cdots \bullet (\texttt{s} \bullet x) \bullet \cdots \leftharpoonup \cdots \bullet (\texttt{s} \bullet \texttt{f}(x)) \bullet \cdots$$

Empty stack symbol. The stack operations we have in $\mathcal{S}tack$ are not designed to have a specific treatment of an empty stack, but it clearly can be simulated by adding a specific unary function symbol when encoding a stack machine: a stack with the special symbol on top is the same thing as an empty stack.

Then, *modulo* the same remarks as in the previous section, we get an encoding of stack machines that implies theorem IV.13 *via* theorem IV.14.

Soundness

The actually delicate part is rather to give a polynomial time decision procedure for balanced observations with stacks.

As in the above section, we are not going to compute directly the iterations of $OW[\vec{\mathrm{p}}]$ (O is now an observation with stack) and see if they eventually reach 0, as the order of nilpotency of an element of $\mathcal{S}tack$ may be exponential in its size, as we saw in example II.53.

In that respect, we are in a situation that is quite similar to what happens with stack machines: Cook proved that it is possible to decide whether such a machine accepts a word in polynomial time, while the actual run of the machine may be of exponential length. We therefore need to speed up computation in a similar way Cook managed to do with his *memoization* [Glü13] technique.

Most of the technical work has already been carried out in section II.4 and we mainly need now to re-read it with complexity issues in mind. Before we get started, let us state the main theorem we aim at proving in this section:

Theorem IV.15
If $O \in \mathcal{O}^{\mathrm{s}}_{\Sigma}$ is an observation with stack, then $\mathcal{L}(O) \in \textsc{Ptime}$.

The proof goes in two steps: first we show that the nilpotency of an element of $\mathcal{S}tack$ can be decided in polynomial time, then we show that given the product $OW[\vec{\mathrm{p}}]$ of an observation and the representation of an integer, we can build in polynomial time a wiring $F \in \mathcal{S}tack$ that is nilpotent *iff.* $OW[\vec{\mathrm{p}}]$ is nilpotent.

The size of wirings. The procedure will have wirings as data for its intermediate steps, we must therefore state how the size of wirings is measured: we will call the *size* of a wiring (notation $|F|$) the total number of occurrences symbols in it.

Nilpotency in $\mathcal{S}tack$. We already know from theorem II.60 that given a finite $F \in \mathcal{S}tack$, its nilpotency is equivalent to the acyclicity of $\mathtt{sat}\,(F)^{\downarrow}$ and $\mathtt{sat}\,(F)^{\uparrow}$. Moreover, we know from theorem II.61 that given a sum F of stack operations which are all increasing (or all decreasing) we can associate to it a balanced wiring which is nilpotent *iff.* F is nilpotent.

First let us show that the saturation is computable in polynomial time.

Proposition IV.16

Given any integer h, there is a function $\textsc{Sat}_h(\cdot) \in \textsc{FPtime}$ that inputs a finite $F \in \mathcal{S}tack$ with $\mathtt{h}(F) \leq h$ and outputs $\mathtt{sat}(F)^{\downarrow}, \mathtt{sat}(F)^{\uparrow}$.

Proof ▸ We write S the number of different function symbols in F. Consider the following algorithm:

```
1: $H := F$
2: while $H^{\downarrow}H^{\uparrow} \not\subseteq H$ do
3:     $H := \mathtt{short}(H)$        (definition II.56)
4: end while
5: return $H^{\downarrow}, H^{\uparrow}$
```

It is clear from proposition II.57 that this algorithm terminates in at most $(S^{\mathtt{h}(F)} + S^{\mathtt{h}(F)-1} + \cdots + 1)^2 \leq (|F|^h + |F|^{h-1} + \cdots + 1)^2$ steps and outputs $\mathtt{sat}(F)^{\downarrow}, \mathtt{sat}(F)^{\uparrow}$.

Moreover, the time cost of each step is that of the computation of $H^{\downarrow}H^{\uparrow}$. We know by proposition II.57 that at any point the total number of elements in H (hence in $H^{\downarrow}$ and $H^{\uparrow}$) is bounded by $(|F|^h + |F|^{h-1} + \cdots + 1)^2$ and that the terms involved are built with unary function symbols and of height at most $\mathtt{h}(F)$, hence their size is at most $\mathtt{h}(F)$ and each unification can be performed in linear time in $|F|$. Therefore the time needed to compute the product $H^{\downarrow}H^{\uparrow}$ is polynomial in $|F|$. ◂

Remark IV.17. In remark II.58, we explained that the $\mathtt{short}(\cdot)$ operation provides an acceleration in the iterations of F we can reach.

This part of the procedure corresponds indeed to the memoization part of the simulation of stack machines we evoked in the beginning of this section: the program considered is augmented with transitions that are stored in $\mathtt{sat}(F)$ and can then be used with the same time cost as any other, while they may hide the composition of an exponential number of the original transitions.

Proposition IV.18

Given any integer h, there is procedure $\textsc{Incr}_h(\cdot) \in \textsc{Ptime}$ that inputs a finite sum F of increasing stack operations such that $\mathtt{h}(F) \leq h$ and accepts *iff.* F is nilpotent.

Proof ▸ By theorem II.61 we know that F is nilpotent *iff.* $F' := \textsc{tr}_{[E,h]}F$ is.

We can compute F' from F in polynomial time, as it just means to compute the product of F with a wiring which size is polynomial in $|F|$ ($\textsc{tr}_{[E,h]}$ contains $\mathtt{card}(E)^{2h}$ flows of size at most $2h$).

Moreover, as F' is balanced and $\mathtt{h}(F') = h$ (theorem II.61), we deduce from the previous section (theorem IV.12) that we can decide the nilpotency of F' in logarithmic space (hence polynomial time) in the size of F' which is polynomial in $|F|$. ◀

Combining these two results, we get:

Theorem IV.19

Given any integer h, there is procedure $\textsc{SNilp}_h(\cdot) \in \textsc{Ptime}$ that inputs a finite $F \in \mathcal{S}tack$ of stack operations such that $\boldsymbol{h}(F) \leq h$ and accepts *iff.* F is nilpotent.

Acceptance of observations with stack. Now, to complete the proof of theorem IV.15, we need to show that it is possible to transform a product $OW[\vec{\mathrm{p}}]$ into an element of $\mathcal{S}tack$ in polynomial time, preserving its eventual nilpotency.

For this we will rely on the fact that, apart for its stack part, $OW[\vec{\mathrm{p}}]$ is a balanced flow. We will use the fact that balanced flows have a finite separating space as we already used in the previous section, but as the stack part does not, we need the following lemma to handle this mixed situation.

Lemma IV.20

Let $\mathbf{U}$ be a separating space (definition II.34) for a wiring F and P a projection (definition II.21) such that $P\mathbf{u} = \mathbf{u}$ for any $\mathbf{u} \in \mathbf{U}$. We have that F is nilpotent *iff.* PF is nilpotent.

Proof ▶ The wiring P being a projection, F nilpotent implies PF nilpotent by proposition II.22. Conversely, as $P\mathbf{u} = \mathbf{u}$ for any $\mathbf{u} \in \mathbf{U}$ and $F\mathbf{U} \subseteq \mathbf{U}$ we have that $F^n\mathbf{U} = (PF)^n\mathbf{U}$ and therefore $(PF)^n = 0$ implies $F^n\mathbf{U} = 0$ which implies $F^n = 0$ because $\mathbf{U}$ is separating for F. ◀

Proposition IV.21

Let O be an observation with stack.

There is a function $\textsc{Red}_O(\cdot) \in \textsc{FPtime}$ that inputs a word W and outputs a wiring $F \in \mathcal{S}tack$ with $\boldsymbol{h}(F) \leq \boldsymbol{h}(O)$ such that F is nilpotent *iff.* $OW[\vec{\mathrm{p}}]$ is for any choice of $\vec{\mathrm{p}}$.

Proof ▶ First we can easily (by associativity/commutativity rearrangements around $\bullet$) turn $OW[\vec{\mathrm{p}}]$ into an element G of $\mathcal{R}_{\mathbf{b}} \bullet \mathcal{S}tack$ of the same size and height (eventually with a constant overhead), preserving its nilpotency.

Writing $G = \sum_i B_i \bullet S_i$, we can consider $B := \sum_i B_i \in \mathcal{R}_{\mathbf{b}}$ and $S := \sum_i S_i$. We can then apply lemma II.37 to get that $\mathbf{Comp}(B)$ is separating for B. Let us write U the set of terms u such that $u \leftharpoonup \star \in \mathbf{Comp}(B)$. We have that $\mathtt{card}(U)$ is polynomial in $|G|$ *via* proposition II.38.

As $G \subseteq B \bullet S$ we have that (following remark II.35) the set of facts of the form $u \bullet t \leftharpoonup \star$, with $u \in U$ and t any term, is separating for G. This means by the previous lemma that if we set $P := \sum_{u \in U} u \leftharpoonup u$ and $P' := P \bullet I$, we have G nilpotent *iff.* $P'G$ nilpotent.

Now we have $P'G = \sum_i u_i \bullet \tau(x) \leftharpoonup v_i \bullet \sigma(x)$ with $u_i, v_i \in U$. If we associate to each element u of U a distinct unary function symbol $\mathtt{f}_u$, we can consider the wiring $F := \sum_i \mathtt{f}_{u_i}(\tau(x)) \leftharpoonup f_{v_i}(\sigma(x)) \in \mathcal{S}tack$ which is nilpotent *iff.* G is.

As for complexity of the transformation of $OW[\vec{\mathrm{p}}]$ into F, it is clear that first G can be computed in linear time from $OW[\vec{\mathrm{p}}]$. Then P' is a wiring of polynomial size (a consequence of proposition II.38 and the bounded height) so that $P'G$ is computed from G in polynomial time. Finally, going from $P'G$ to F is just a matter of associating symbol to closed terms and can be performed in polynomial time because the cardinal of U is polynomial $|G|$. ◀

It only remains now to compose the results above to prove the main theorem of this section.

Proof (of theorem IV.15) ▶ Given an observation with stack O, the composition of $\textsc{Red}_O(\cdot)$ and $\mathrm{S}\textsc{Nilp}_{h(O)}(\cdot)$ yields a polynomial decision procedure for the language $\mathcal{L}(O)$. ◀

Perspectives

Logic programming

We saw in chapter III that it is possible to build a traced category within the resolution semiring and that this category has all the structures required to interpret λ-CALCULUS. It seems reasonable that the comments of section III.3 could be extended into a wider approach linking recent developments in proof theory and logic programming, two domains that have been growing apart for some time.

The complexity results of chapter IV should be related with the large amount of work that has already been done on the complexity of logic programming. In particular a study of the specific case of logic programs with unary functions symbols with results of section II.4 and section IV.4 in mind might turn out interesting.

In this perspective, we would need to further extend the framework and the results, considering other problems than just nilpotency: although it is a very natural notion from the algebraic point of view, logic programming is usually more concerned with the reachability of a certain goal assuming a number of facts.

Moreover, the restriction to flows (safe clauses with exactly one atom in the body) is a strong restriction from the point of view of logic programming. Its great interest is to allow us to work with algebraic intuitions and tools, but the case of general logic programs may also be investigated. Some blueprinting has already been done on a relaxed version of the framework with multiple atoms in the body and the head of clauses [Gir13]. The extension of the results on logarithmic space to this case should be straightforward, but the case of polynomial time might be more delicate: one would need to extend the notion of increasing and decreasing flow (page 45) which is a crucial element of the soundness proof.

Relaxing the safety condition (the fact that in a flow $t \leftharpoonup u$ one must have $\texttt{var}(t) \subseteq \texttt{var}(u)$) is also a possibility: there do not seem to be any technical difficulty in defining a semiring with the relaxed definition (though we may loose certain nice properties like the ideal structure of the set of facts, remember remark II.6). Note that this would introduce a form of non-determinism, as for instance the flow $x \leftharpoonup \star$ can be understood (following remark II.11) as the sum of all the $t \leftharpoonup \star$ with t closed, a flow that is non-deterministic.

Implicit computational complexity

We also aim at extending the correspondence between complexity classes and restricted semirings of $\mathcal{R}$. A first candidate would be the class Pspace, that could benefit from the work already done on logarithmic space computation: indeed the cardinality of $\mathbf{Comp}(F)$ (proposition II.38) indicates how much space will be needed to solve the nilpotency problem. The fact that this number grows polynomially with the number of different symbols involved was necessary in order to characterize logarithmic space computation, but remark that it grows exponentially with the maximal arity, indicating a polynomial space bound. The NC hierarchy might be considered too, as some subcases of the unification problem are known to lie within it.

Another direction is to deepen the relation with proof theory. Indeed our approach comes from the geometry of interaction program and the work relating complexity theory and linear logic we evoked in the introduction of chapter IV. Thanks to the GoI interpretation of linear logic, any restricted proof system can be translated in the resolution semiring, and this allows for a study of its complexity in the framework we developed.

The question of the complexity of decision problems *vs.* the complexity of computing functions is related to this. We have been concerned in this thesis only with decision problems, but it should be possible to also obtain characterizations of complexity classes of functions with the same methods. An application of this would be to build an abstract proof of the compositionality of FLogspace.

This idea may also work in the other direction: for instance one could look for a proof system corresponding the balanced semiring of section II.3 and therefore to logarithmic space computation.

Complexity and abstract algebra

Finally, we want to think of this thesis as a first step towards a possibly fruitful relation between complexity theory and abstract algebra.

The decision procedure for the $\mathcal{S}tack$ semiring is a good illustration of this: to solve the algebraic question of nilpotency, we imported and adapted ideas used to handle pushdown automata in order to design our decision procedure. This also sheds an original light on the memoization technique, which becomes in our context a sort of exponentiation by squaring.

We believe an important next step would be to understand better what are the algebraic principle at work in the characterizations we obtained so far. Indeed, balance and the use of unary function symbols are syntactic restrictions that do not make sense from an algebraic point of view, while for instance in the case of $\mathcal{S}tack$ the fact that $f^2 \neq 0$ implies $f^n \neq 0$ for all n (corollary II.49) is a key ingredient of the procedure and does not rely on syntax.

Among other possibilities, we could try to adapt the notion of finiteness of the theory of von Neumann algebras [Tak01] in our setting: without working out the details, imagine we say two projections (definition II.21) P, Q of a semiring are equivalent if there is an isometry W of the semiring such that $P = WW^{\dagger}$ and $Q = W^{\dagger}W$, that P is (strictly) included in Q if the domain of P is (strictly) included in the domain of Q. Then we can say that a projection is finite if it is not equivalent to any projection strictly included in it. This adaptation of Dedekind-finiteness is interesting as it is relative to the semiring we consider: the projection (indeed its domain) may be infinite from a cardinality point of view, but not from the algebraic point of view of the semiring. For instance, consider a semiring containing $f = \mathtt{f}(x) \leftharpoonup x$ and $f^{\dagger} = x \leftharpoonup \mathtt{f}(x)$. It must also contain the projections $P = f^{\dagger}f$ and $Q = ff^{\dagger}$, with Q strictly included in P, and P would hence be considered as infinite in this semiring.

This notion might be a tool to understand more abstractly logarithmic space computation: in earlier works, instead of the notion of balance, the characterization of logarithmic space relied on a semiring of permutations [AB14]. A common characteristics of these two semiring is that they would be finite in the sense sketched above.

Appendix: GoI interpretation

We expose in this appendix the GoI interpretation of multiplicative linear logic (MELL) and λ-CALCULUS. Our objective is to provide an reference that is easy to read. We will give the interpretation in the graphical language we introduced in section I.3 and we won't refrain from slight abuses of notation in order to draw more readable pictures.

For further details, the reader should consult the literature [Reg92, HS06].

Linear logic and λ-CALCULUS

Let us first recall the encoding of pure λ-CALCULUS in MELL.[3] Composing this encoding with the GoI interpretation of MELL, one obtains the GoI interpretation of pureλ-CALCULUS. Note that the encoding we present is the so-called "call by name" encoding, but there exist other encodings with different features [Acc12].

We consider a variant of intuitionistic MELL with a specific formula U and the equation $!U \multimap U = U$ (dually $!U \otimes U^{\perp} = U^{\perp}$). A λ-term t, with free variables among $x_1, \dots, x_n$ will be translated as a proof of the sequent $!U, \dots, !U \vdash U$. We abbreviate this as

$$x_1 : !U, \dots, x_n : !U \vdash t : U$$

The term x where we consider that there is no other free variable than x corresponds to an axiom rule followed by a dereliction

$$\dfrac{\dfrac{}{U \vdash U}\,\texttt{Ax}}{x : !U \vdash x : U}\,\texttt{d}$$

The abstraction of a variable corresponds to a $\vdash\multimap$ rule: the term $\lambda x.t$ is encoded as

$$\dfrac{x_1 : !U, \dots, x_n : !U, x : !U \vdash t : U}{x_1 : !U, \dots, x_n : !U \vdash \lambda x.t : U}\,\vdash\multimap$$

[3] A more detailed account of this encoding, using the terminology of proofnets can be found in L. Regnier's thesis [Reg92]

The application is encoded using a combination of various rules, including promotion (allowing the duplication of the argument) and `cut` rule: the term $(t)u$ (where we assume that the free variable of t and u are distinct, as we will have a specific rule to handle this later) is encoded as

$$\dfrac{x_1 : !U, \dots, x_n : !U \vdash t : U \qquad \dfrac{\dfrac{\dfrac{y_1 : !U, \dots, y_m : !U \vdash u : U}{!!U, \dots, !!U \vdash : !U}\,! }{!U, \dots, !U \vdash : !U}\,!! \qquad \dfrac{}{U \vdash U}\,\texttt{Ax}}{!U, \dots, !U, !U \multimap U \vdash U}\,\multimap\vdash}{x_1 : !U, \dots, x_n : !U, y_1 : !U, \dots, y_m : !U \vdash (t)u : U}\,\texttt{cut}$$

We chose to decompose the usual promotion rule (**!**) of linear logic into a functorial promotion followed by a series of digging (**??**) rule to be more in line with the interpretation of MELL we give below. Note also that in the `cut` rule, we use the equation $U = !U \multimap U$.

Then we have an encoding of manipulation of variables by structural rules: adding an unused free variable x to the term t is encoded as a weakening

$$\dfrac{x_1 : !U, \dots, x_n : !U \vdash t : U}{x_1 : !U, \dots, x_n : !U, x : !U \vdash t : U}\,\texttt{W}$$

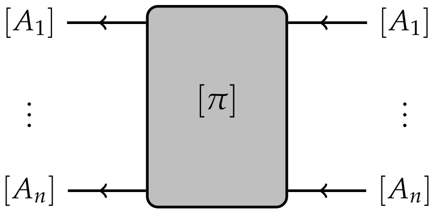
and the merging of two distinct variable into one is encoded as a contraction

$$\dfrac{x_1 : !U, \dots, x_n : !U, x : !U, y : !U \vdash t : U}{x_1 : !U, \dots, x_n : !U, z : !U \vdash \{x \mapsto z, y \mapsto z\} t : U}\,\texttt{C}$$

Interpretation of MELL

Let us suppose now that we dispose of a GoI situation (definition I.23) we denote $(\mathfrak{C}, !, U)$.

The interpretation of a proof π of the sequent $\vdash A_1, \dots, A_n$ will be a morphism $[\pi] : [A_1] \oplus \cdots \oplus [A_n] \leftarrow [A_1] \oplus \cdots \oplus [A_n]$, where $[A]$ is defined inductively by $[\alpha] = [\alpha^\perp] := U$ for any atom α, $[!A] = [?A] := ![A]$ and $[A \parr B] = [A \otimes B] := U$.

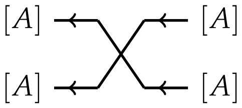


Combining the retractions $(t, t') : !! \lhd !$ and $(a, a') : !U \lhd U$, we can obtain retractions $: !^n U \lhd U$ for any n. In what follows we will draw simply as (a, a') the retraction $: [A] \lhd U$ leaving implicit the actual objects involved.

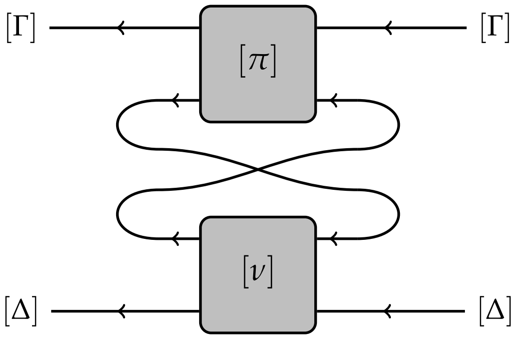
The application of the functor **!** will be depicted as a box surrounding its argument.

Note that the diagram we draw for the interpretation of the `cut` rule is an abuse of notation: it is easy to imagine how to obtain something that is equivalent and correct, but this would result in a quite unreadable diagram with a lot of wires crossing, defeating the idea of an intuitive graphical presentation.

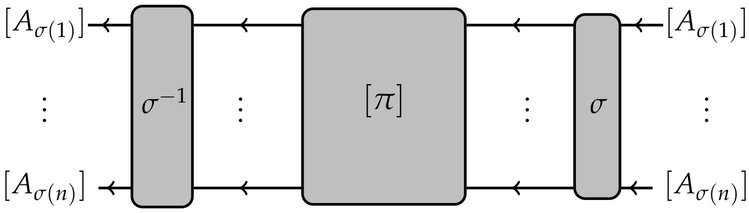
Axiom rule. $\dfrac{}{\vdash A^{\perp}, A}\,\mathtt{Ax}$ is interpreted as a symmetry morphism $\sigma_{[A],[A]}$



Cut rule. $\dfrac{\overset{\vdots\;\pi}{\vdash \Gamma, A} \qquad \overset{\vdots\;\nu}{\vdash A^{\perp}, \Delta}}{\vdash \Gamma, \Delta}\,\mathtt{cut}$ is interpreted using the trace structure

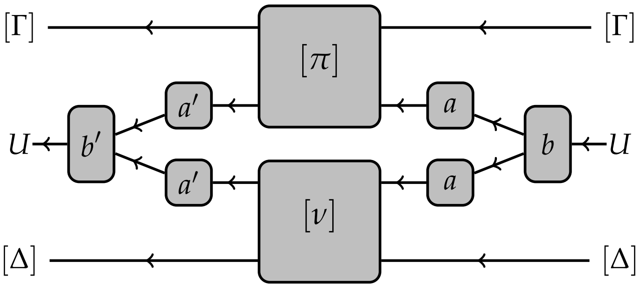


Exchange rule. $\dfrac{\overset{\vdots\;\pi}{\vdash A_1, \ldots, A_n}}{\vdash A_{\sigma(1)}, \ldots, A_{\sigma(n)}}\,\sigma$ is interpreted using the permutation morphisms σ and σ^{-1}, built by combining symmetry morphisms

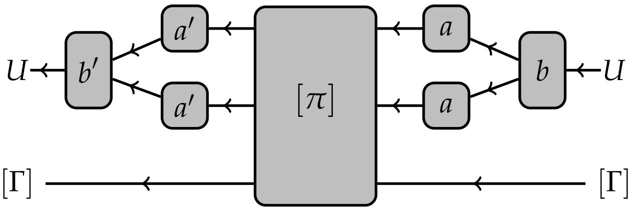


Multiplicative rules

Tensor rule. $\dfrac{\begin{array}{cc}\vdots\;\pi & \vdots\;\nu\\ \vdash \Gamma, A & \vdash B, \Delta\end{array}}{\vdash \Gamma, A \otimes B, \Delta}\otimes$ is interpreted as

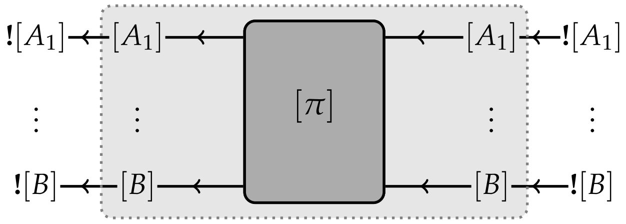


Par rule. $\dfrac{\begin{array}{c}\vdots\;\pi\\ \vdash A, B, \Delta\end{array}}{\vdash A \parr B, \Gamma}\parr$ is interpreted as

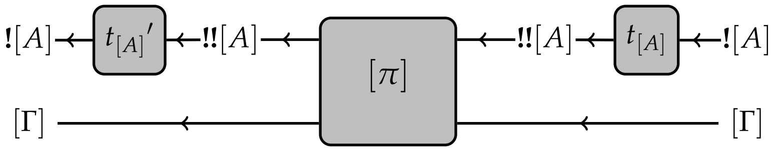


Exponential rules

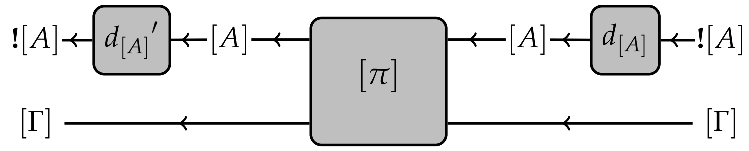
Promotion rule. $\dfrac{\begin{array}{c}\vdots\;\pi\\ \vdash A_1, \ldots, A_n, B\end{array}}{\vdash \mathbf{?}A_1, \ldots, \mathbf{?}A_n, \mathbf{!}B}\,\mathbf{!}$ is interpreted as



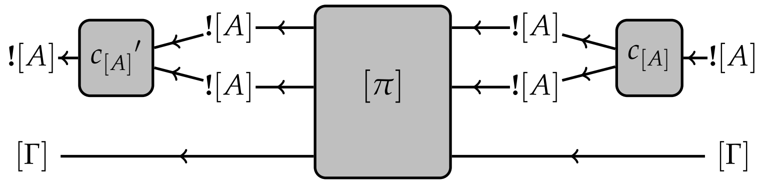
Digging rule. $\dfrac{\begin{array}{c}\vdots\;\pi\\ \vdash \mathbf{??}A, \Gamma\end{array}}{\vdash \mathbf{?}A, \Gamma}\,\mathbf{??}$ is interpreted as



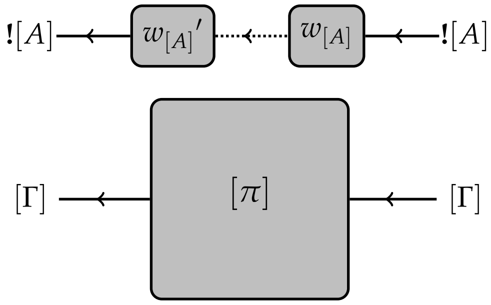
Dereliction rule. $\dfrac{\begin{array}{c}\vdots\ \pi\\ \vdash A,\Gamma\end{array}}{\vdash \mathbf{?}A,\Gamma}\ \mathtt{d}$ is interpreted as



Contraction rule. $\dfrac{\begin{array}{c}\vdots\ \pi\\ \vdash \mathbf{?}A,\mathbf{?}A,\Gamma\end{array}}{\vdash \mathbf{?}A,\Gamma}\ \mathtt{c}$ is interpreted as



Weakening rule. $\dfrac{\begin{array}{c}\vdots\ \pi\\ \Gamma\end{array}}{\vdash \mathbf{?}A,\Gamma}\ \mathtt{w}$ is interpreted as



Bibliography

[AB14] Clément Aubert and Marc Bagnol. Unification and logarithmic space. In Gilles Dowek, editor, *RTA-TLCA 2014*, volume 8650 of *LNCS*, pages 77–92. Springer, 2014. *(cited on pp. 8 and 77)*

[ABPS14] Clément Aubert, Marc Bagnol, Paolo Pistone, and Thomas Seiller. Logic programming and logarithmic space. In Jacques Guarrigue, editor, *APLAS 2014*, volume 8858 of *LNCS*, pages 39–57,. Springer, 2014. *(cited on p. 8)*

[Acc12] Beniamino Accattoli. Proof nets and the call-by-value lambda-calculus. In *Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2012, Rio de Janeiro, Brazil, September 29-30, 2012.*, pages 11–26, 2012. *(cited on p. 78)*

[AHS02] Samson Abramsky, Esfandiar Haghverdi, and Philip J. Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002. *(cited on p. 20)*

[AJ94] Samson Abramsky and Radha Jagadeesan. New foundations for the geometry of interaction. *Inf. Comput.*, 111(1):53–119, May 1994. *(cited on p. 20)*

[AS14] Clément Aubert and Thomas Seiller. Logarithmic space and permutations. *CoRR*, abs/1301.3189, 2014. *(cited on p. 67)*

[Aub13] Clément Aubert. *Linear Logic and Sub-polynomial Classes of Complexity*. PhD thesis, Université Paris 13–Sorbonne Paris Cité, November 2013. *(cited on p. 67)*

[Bai08] Patrick Baillot. Linear logic, types and implicit computational complexity, March 2008. *(cited on pp. 7 and 59)*

[BC92] Stephen J. Bellantoni and Stephen Arthur Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *STOC*, pages 283–93. ACM, 1992. *(cited on p. 59)*

[Bla82] Howard A. Blair. The recursion-theoretical complexity of the semantics of predicate logic as a programming language. *Information and Control*, 54(1/2):25–47, 1982. *(cited on p. 58)*

[BO03] Marco Bellia and Maria Eugenia Occhiuto. N-axioms parallel unification. *Fund. Inform.*, 55(2):115–128, 2003. *(cited on p. 18)*

[BP01] Patrick Baillot and Marco Pedicini. Elementary complexity and geometry of interaction. *Fund. Inform.*, 45(1–2):1–31, 2001. *(cited on pp. 14, 42, 43, and 59)*

[BT04] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE Computer Society, 2004. *(cited on p. 59)*

[CCIL08] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *LNCS*, pages 407–424. Springer, 2008. *(cited on p. 37)*

[Coo71] Stephen A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM*, 18(1):4–18, 1971. *(cited on p. 70)*

[Dan90] 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 p. 19)*

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001. *(cited on pp. 19, 33, 39, 56, and 58)*

[DKM84] Cynthia Dwork, Paris C. Kanellakis, and John C. Mitchell. On the sequential nature of unification. *J. Log. Program.*, 1(1):35–50, 1984. *(cited on pp. 18 and 19)*

[DKS88] Cynthia Dwork, Paris C. Kanellakis, and Larry J. Stockmeyer. Parallel algorithms for term matching. *SIAM J. Comput.*, 17(4):711–731, 1988. *(cited on p. 18)*

[DL12] Ugo Dal Lago. A short introduction to implicit computational complexity. In Nick Bezhanishvili and Valentin Goranko, editors, *Lectures on Logic and Computation*, volume 7388 of *Lecture Notes in Computer Science*, pages 89–109. Springer Berlin Heidelberg, 2012. *(cited on pp. 7 and 59)*

[DR95] Vincent Danos and Laurent Regnier. Proof-nets and the hilbert space. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, number 222 in London Math. Soc. Lecture Note Ser., pages 307–328. CUP, June 1995. *(cited on p. 58)*

[Fit87] Melvin Fitting. *Computability Theory, Semantics, and Logic Programming*. Oxford University Press, Inc., New York, NY, USA, 1987. *(cited on p. 58)*

[Gal95] Jean Gallier. On the correspondence between proofs and & λ-Terms. In Philippe de Groote, editor, *The Curry-Howard isomorphism*, Cahiers du Centre de Logique, pages 55–138. Academia, 1995. *(cited on p. 11)*

[Gen34a] Gerhard Gentzen. The consistency of elementary number theory. In M.E. Szabo, editor, *The Collected Works of Gerhard Gentzen*, pages 132–213. North Holland, Amsterdam, 1934. 1968. *(cited on p. 10)*

[Gen34b] Gerhard Gentzen. Investigations into logical deduction. In M.E. Szabo, editor, *The Collected Works of Gerhard Gentzen*, pages 68–129. North Holland, Amsterdam, 1934. 1968. *(cited on pp. 10 and 11)*

[Gir87a] Jean-Yves Girard. Linear logic. *Theoret. Comput. Sci.*, 50(1):1–101, 1987. *(cited on pp. 11 and 13)*

[Gir87b] Jean-Yves Girard. Multiplicatives. In G. Lolli, editor, *Logic and Computer Science: New Trends and Applications*, pages 11–34. Rosenberg & Sellier, 1987. *(cited on p. 13)*

[Gir89a] Jean-Yves Girard. Geometry of interaction 1: Interpretation of system F. *Studies in Logic and the Foundations of Mathematics*, 127:221–260, 1989. *(cited on pp. 7, 13, 20, 50, and 57)*

[Gir89b] Jean-Yves Girard. Towards a geometry of interaction. In John W. Gray and Andre Ščedrov, editors, *Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference held June 14-20, 1987*, volume 92 of *Categories in Computer Science and Logic*, pages 69–108. AMS, 1989. *(cited on pp. 7, 9, 13, 50, and 57)*

[Gir90] Jean-Yves Girard. Geometry of interaction 2: Deadlock-free algorithms. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 76–93. Springer Berlin Heidelberg, 1990. *(cited on pp. 7 and 52)*

[Gir95a] Jean-Yves Girard. Geometry of interaction III: accommodating the additives. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, number 222 in London Math. Soc. Lecture Note Ser., pages 329–389. CUP, June 1995. *(cited on pp. 7, 13, 14, 30, 50, and 57)*

[Gir95b] Jean-Yves Girard. Light linear logic. In Daniel Leivant, editor, *LCC*, volume 960 of *LNCS*, pages 145–176. Springer, 1995. *(cited on p. 59)*

[Gir96] Jean-Yves Girard. Proof-nets: The parallel syntax for proof-theory. *Logic and Algebra*, 180:97–124, May 1996. *(cited on p. 13)*

[Gir06] Jean-Yves Girard. Geometry of interaction IV: the feedback equation. In Stoltenberg-Hansen and Väänänen, editors, *Logic Colloquium 2003*, pages 76–117. The Association for Symbolic Logic, 2006. *(cited on pp. 13 and 52)*

[Gir11] Jean-Yves Girard. Geometry of interaction V: logic in the hyperfinite factor. *Theoret. Comput. Sci.*, 412(20):1860–1883, April 2011. *(cited on p. 13)*

[Gir12] Jean-Yves Girard. Normativity in logic. In Peter Dybjer, Sten Lindström, Erik Palmgren, and Göran Sundholm, editors, *Epistemology versus Ontology*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 243–263. Springer, 2012. *(cited on p. 63)*

[Gir13] Jean-Yves Girard. Three lightings of logic. In Simona Ronchi Della Rocca, editor, *CSL*, volume 23 of *LIPIcs*, pages 11–23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. *(cited on p. 75)*

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. CUP, 1989. *(cited on p. 57)*

[Glü13] Robert Glück. Simulation of two-way pushdown automata revisited. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013.*, pages 250–258, 2013. *(cited on p. 71)*

[Hag00a] Esfandiar Haghverdi. *A Categorical Approach to Linear Logic, Geometry of Proofs and Full Completeness*. PhD thesis, University of Ottawa, 2000. *(cited on pp. 20, 28, and 29)*

[Hag00b] Esfandiar Haghverdi. Unique decomposition categories, geometry of interaction and combinatory logic. *Mathematical Structures in Computer Science*, 10(2):205–230, 2000. *(cited on p. 20)*

[Har72] Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Inform.*, 1(4):336–344, 1972. *(cited on p. 65)*

[HS06] Esfandiar Haghverdi and Philip Scott. A categorical model for the geometry of interaction. *Theoretical Computer Science*, 350(2–3):252 – 274, 2006. Automata, Languages and Programming: Logic and Semantics (ICALP-B 2004) Automata, Languages and Programming: Logic and Semantics 2004. *(cited on pp. 9, 20, 25, 28, 29, 54, and 78)*

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. In *CoCo*, pages 112–115. IEEE Computer Society, 1988. *(cited on p. 67)*

[Jon75] Neil D. Jones. Space-bounded reducibility among combinatorial problems. *J. Comput. Syst. Sci.*, 11(1):68–85, 1975. (cited on p. 68)

[JSV96] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119:447–468, 4 1996. (cited on p. 20)

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124, 1989. (cited on p. 9)

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theoret. Comput. Sci.*, 318(1):163–180, 2004. (cited on p. 59)

[Lei93] Daniel Leivant. Stratified functional programs and computational complexity. In Mary S. Van Deusen and Bernard Lang, editors, *POPL*, pages 325–333. ACM Press, January 1993. (cited on p. 59)

[LL09] Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In Patricia M. Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 489–493. Springer, 2009. (cited on p. 37)

[LM93] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. In Marc Bezem and JanFriso Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 274–288. Springer Berlin Heidelberg, 1993. (cited on p. 59)

[LS10] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 205–225. Springer, 2010. (cited on p. 59)

[MA86] Ernest G. Manes and Michael A. Arbib, editors. *Algebraic Approaches to Program Semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1986. (cited on p. 27)

[Mai04] Harry Mairson. Linear lambda calculus and ptime-completeness. *J. Funct. Program.*, 14(6):623–633, November 2004. (cited on p. 59)

[Mel09] Paul-André Mellies. Categorical semantics of linear logic. In *Interactive models of computation and program behaviour*. Société Mathématique de France, 2009. (cited on p. 20)

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971. (cited on p. 20)

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982. *(cited on pp. 16 and 18)*

[MSS12] Octavio Malherbe, Philip J. Scott, and Peter Selinger. Partially traced categories. *Journal of Pure and Applied Algebra*, 216(12):2563 – 2585, 2012. *(cited on pp. 27 and 52)*

[MT03] Harry Mairson and Kazushige Terui. On the computational complexity of cut-elimination in linear logic. *Theoret. Comput. Sci.*, pages 23–36, 2003. *(cited on p. 59)*

[Nee04] Peter Møller Neergaard. A functional language for logarithmic space. In *In APLAS*, pages 311–326, 2004. *(cited on p. 59)*

[OYY87] Masaaki Ohkubo, Hiroto Yasuura, and Shuzo Yajima. On parallel computation time of unification for restricted terms. Technical report, Kyoto University, May 1987. *(cited on p. 18)*

[Pig13] Giovanni Pighizzini. Two-way finite automata: Old and recent results. *Fund. Inform.*, 126(2–3):225–246, 2013. *(cited on p. 65)*

[PW78] Mike Paterson and Mark N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978. *(cited on p. 18)*

[Reg92] Laurent Regnier. *Lambda-calcul et réseaux*. PhD thesis, Université Paris 7, 1992. *(cited on p. 78)*

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. *(cited on pp. 7 and 14)*

[Sav98] John E. Savage. *Models of computation - exploring the power of computing*. Addison-Wesley, 1998. *(cited on p. 67)*

[Sch07] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS*, pages 411–420. IEEE Computer Society, 2007. *(cited on p. 59)*

[Sei12] Thomas Seiller. *Logique dans le Facteur Hyperfini : Géometrie de l'Interaction et Complexité*. PhD thesis, Université de la Méditerranée, 2012. *(cited on p. 67)*

[Sel07] Peter Selinger. Dagger compact closed categories and completely positive maps: (extended abstract). *Electronic Notes in Theoretical Computer Science*, 170(0):139 – 163, 2007. Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005). *(cited on p. 55)*

[Sel11] Peter Selinger. A survey of graphical languages for monoidal categories. In Bob Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 289–355. Springer Berlin Heidelberg, 2011. *(cited on p. 21)*

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Inform.*, 26(3):279—–284, 1988. *(cited on p. 67)*

[Tak01] Masamichi Takesaki. *Theory of Operator Algebras 1*, volume 124 of *Encyclopedia of Mathematical Sciences*. Springer, 2001. *(cited on p. 77)*

[Urz03] PawełUrzyczyn. A simple proof of the undecidability of strong normalisation. *Mathematical. Structures in Comp. Sci.*, 13(1):5–13, February 2003. *(cited on p. 58)*

[WW86] Klaus W. Wagner and Gerd Wechsung. *Computational Complexity*, volume 21 of *Mathematics and its Applications*. Springer, 1986. *(cited on pp. 65, 66, and 70)*