Lambda-calculus and formal language theory

Sylvain Salvati

▶ To cite this version:

Sylvain Salvati. Lambda-calculus and formal language theory. Computer Science [cs]. Université de Bordeaux, 2015. tel-01253426

HAL Id: tel-01253426
https://hal.science/tel-01253426v1

Submitted on 10 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUE ET INFORMATIQUE DE BORDEAUX

Habilitation à diriger les recherches

Soutenue publiquement le 10 décembre 2015

Sylvain Salvati

Lambda-calculus and formal language theory

Jury:

Rapporteurs:

Thomas Ehrhard	- Directeur de recherche CNRS / PPS
Giorgio Satta	- Professeur Università Padua, Italie
Sophie Tison	- Professeur Université de Lille / CRIStAL

Examinateurs:

Bruno Courcelle	- Professeur émérite Université de Bordeaux / LaBRI
Philippe de Groote	- Directeur de recherche INRIA / Loria
Jérôme Leroux	- Directeur de recherche CNRS / LaBRI

Contents

Contents iii

1 Introduction **1**
1.1 Research context . . . 1
1.2 Research orientation and main contributions . . . 2
1.3 Organization of the document . . . 5

2 Preliminaries and recognizability **7**
2.1 Simply typed λ-calculus . . . 8
2.2 Special constants . . . 10
2.3 Some syntactic restrictions on simply typed λ-calculus . . . 12
2.4 Models of λ-calculi . . . 14
2.5 Recognizability in simply typed λ-calculus . . . 21
2.6 Conclusion and perspective . . . 26

3 Abstract Categorial Grammars **29**
3.1 Abstract Categorial Grammars . . . 30
3.2 Expressiveness . . . 34
3.3 Parsing Algorithms . . . 39
3.4 OI grammars . . . 42
3.5 Conclusion and perspectives . . . 47

4 Mildly Context Sensitive Languages **51**
4.1 On mild-context sensitivity and its limitations . . . 53
4.2 Multiple Context Free Grammars . . . 59
4.3 The language MIX . . . 62
4.4 Iteration properties for MCFGs . . . 70
4.5 Classifying Mildly Context Sensitive Formalisms . . . 72
4.6 Conclusion and perspectives . . . 73

5 Logic and Verification **77**
5.1 Schematology . . . 79
5.2 Parity automata . . . 81
5.3 Wreath product and weak parity automata . . . 87
5.4 λY-calculus and abstract machines . . . 90

5.5 A λY-model for parity automata 96
5.6 Adequacy of the model via parity games 102
5.7 Conclusion and perspectives 109

6 Conclusion **113**
Bibliography 116
Personal bibliography 134

Abstract

Formal and symbolic approaches have offered computer science many application fields. The rich and fruitful connection between logic, automata and algebra is one such approach. It has been used to model natural languages as well as in program verification. In the mathematics of language it is able to model phenomena ranging from syntax to phonology while in verification it gives model checking algorithms to a wide family of programs.

This thesis extends this approach to simply typed lambda-calculus by providing a natural extension of recognizability to programs that are representable by simply typed terms. This notion is then applied to both the mathematics of language and program verification. In the case of the mathematics of language, it is used to generalize parsing algorithms and to propose high-level methods to describe languages. Concerning program verification, it is used to describe methods for verifying the behavioral properties of higher-order programs. In both cases, the link that is drawn between finite state methods and denotational semantics provide the means to mix powerful tools coming from the two worlds.

Acknowledgments

First of all, I would like to thank the members of the jury for the attention the have paid to my work.

I am grateful to Burno Courcelle for having accepted to be the president of the jury. When I was writing this document, I realized how much his work has influenced mine in many respects.

I thank the reviewers for having read this document in details. Moreover, as this document reports on work that I did in two different fields, I must thank them for the patience they had to go through chapters which had little to do with their daily research. I thank Thomas Ehrhard for having done this with computational linguistics and also for the discussions I could have with him during the past years on denotational semantics. Giorgio Satta had the difficult task to accommodate the peculiarities of the French *habilitation* and I am very happy that he accepted to do this. The third reviewer is Sophie Tison who has a high profile in automata theory. I have remarked that the community working on automata does not like λ-calculus so much, I am thankful to her for having accepted to have overcome this reluctance.

I am very pleased that Jérôme Leroux accepted to be part of the jury. First because, I really like his work and also, because, now, as the head of the team *Méthodes formelles*, he represents the stimulating environment in which my work has grown.

I am glad that Philippe de Groote is in this jury. I started research with him and, still, in the rare occasions that we find to discuss research topics or technical problems, I am always impressed by the elegance and clarity of his ideas.

I want to thank the PoSET team, David Janin and Myriam de Sainte-Catherine with whom I am trying to find the best tune.

When I arrived at LaBRI, I was immediately welcome by the team *Méthodes formelles*. Not only did it provide me with a strong scientific environment, but also could I have very stimulating discussions with its members. Among these people, I address special thanks to Géraud Sénizergues whose vivid enthusiasm and joyful pleasure in research is strongly contagious; to Igor Walukiewicz who introduced me to the fascinating world of infinite objects; and to Anca Muscholl for her support and encouragement.

Research is a matter of ideas and inspiration. It seems impossible to do anything of interest without the insights and the ideas of other researchers.

This is true for me and I must say that I have had the chance to collaborate with very talented researchers to whom my work owes most. I want to thank Makoto Kanazawa who has provided me with a postdoc when I needed one, for the common work and his patient listening to my so often wrong intuitions. I thank also Greg Kobele for the discussions on linguistics and the formal work. He is the only person I know who is combining at such a high degree an expertise in linguistic and in mathematics. Ryo Yoshinaka did his PhD nearly at the same time as I did and I remember cheerfully the moment when I could work with him. I want to thank my PhD students Pierre and Jérôme and my master students for having trusted me as a superviser. We still have not produced research together, but I hope this will happen before he retires, so I thank Pierre Casteran for teaching me CoQ and sharing with me his smart ideas about coding in the calculus of constructions.

Finally I would like to thank my wife for her constant support and for taking care of everything at home when I am away. A final word for my sons, Arthur and Timothée, to whom I want to say again how much I care for them.

Chapter 1

Introduction

1.1 Research context

This document is a synthesis of the research that I carried out in the past ten years and is part of my file to obtain the *habilitation à diriger les recherches*. I review the work I have done since I obtained my PhD in 2005. I did this work from the end of 2005 until the beginning of 2007 as a postdoc at the National Institute of Informatics in Tokyo under the supervision of Makoto Kanazawa. I then joined INRIA Bordeaux Sud-Ouest as a researcher in the project Signes. My work has taken place in LaBRI and I have been strongly influenced by its scientific environment.

While I have been in the team Signes, I have been working on formalisms for natural language, aiming at understanding their algorithmic properties, their limitations and their relations.

In 2011, when the team Signes was ended, I was part of the ANR project FREC, on *Frontiers of RECognizability*. This project allowed me to apply methods I had developed for grammatical formalisms to verification problems and infinitary systems.

In this period of time I have had the opportunity to supervise two PhD students:

- Pierre Bourreau who defended his thesis in 2012. He worked on generalizing parsing methods for abstract categorial grammars based on datalog programs.

- Jérôme Kirman who is expected to defend his thesis by the end of the year. He worked on high-level modeling of natural language based on logics related to finite state automata.

In the context of FREC, I have supervised the postdoctoral studies of Laurent Braud on higher-order schemes.

I also supervised a number of master students on various topics related to formal language theory.

1.2 Research orientation and main contributions

My research activities find applications in two different fields: mathematics of language and verification. In my work I do not really make a distinction between those two seemingly different fields. Indeed, ideas or tools I use for solving a problem in one field may then find some applications in the other. This is mainly due to the technical apparatus that I use: λ-calculus and formal language theory. They provide a theoretical framework which is favorable to the generalization of techniques and ideas. Once technical ideas are made simple, they naturally find applications in other fields. A good example of this situation is the extension of the notion of recognizability, or of finite state machines, to simply typed λ-calculus that I have introduced. This notion is simple and provides ways of proving the decidability of problems in both fields with simplicity. It also gives general guidelines for future developments of my work. I will now present the general approaches I have followed in both domains and my main contributions.

Mathematics of language

The aim of mathematics of language is to understand natural language from a mathematical point of view. Not only is the goal to model formally natural language, but also to question the very nature of language. The properties of language this field tries to address are, among others, those of its learnability, its syntactic properties, how utterances are related to meaning representations. All the general properties of natural language conspire to delineate a particular class of languages that is limited in its expressive power and that is able to model every human language. Clearly, setting the border of this class is in the tradition initiated by Chomsky [76] and the generative approach to linguistics. Many mathematical models have been proposed in the literature which address various aspects of natural languages. I have been interested in the grammatical ones and I addressed questions related to the mathematical models of syntax, their algorithmic properties with and relation to semantics.

I have worked in the formal setting of Abstract Categorial Grammars (ACGs), a formalism based on λ-calculus. A first line of work has been to compare ACGs to other formalisms and I could prove that most of them could be faithfully represented in this setting [S13, S18, S10]. Moreover, I could prove [S17] that a particular class of (second order ones) ACGs have the same kind of limitations as formalisms that were considered as *mildly context sensitive*, a notion proposed by Joshi [161] so as to give a formal description of the limitations grammars modeling natural language should have. This led me to study this notion from a formal point of view. Surprisingly, I could prove that the formalisms that were considered as mildly context sensitive were having unexpected properties. In particular, I proved [S19] that the language MIX which presents no structure and is generally considered as a language that should be out of the scope of grammars for natural languages was captured by classes of grammars considered so far as mildly context sensitive. Related to this result,

I have been able to solve a long standing conjecture by Joshi that MIX was not in the class of Tree Adjoining Languages [S8]. I could also show, contrary to what was assumed so far, that, for these grammars, iterations in derivations did not translate into simple iterations in strings [S6]. All these results point to the fact that the class of languages that is widely considered as able to capture natural languages was not very well understood.

A second line of work has consisted in studying the algorithmic properties of ACGs, mainly the parsing algorithms related to these grammars. This work is a sequel of my PhD work [S22] in which I proposed a parsing algorithm under the hypothesis that the grammar is not using copying mechanisms. I generalized this approach [S21] to the case of copying. As my initial work proposed a parsing algorithm for languages of λ-terms, so is that generalization. Following Montague's approach to natural language semantics [219], this result gives an algorithm so as to generate texts from their meaning representations. The interest of this work is that it naturally led to the notion of recognizability in the simply typed λ-calculus [S23] that has proved useful in verification. This result was further generalized to cope with the parsing problem of higher-order OI grammars [S12]. The results of my PhD have been revisited by Makoto Kanazawa who proposed to use datalog programs so as to implement parsers for ACGs and also for other formalisms [168, 167]. These results together with those I obtained for the non-linear cases were giving an interesting opportunity to explore further parsing methods for grammars based on λ-calculus and datalog. This has become the PhD topic of Pierre Bourreau. He proposed parsing methods for almost affine λ-calculus based on datalog [S3, S2] and also some datalog methods to cope with grammars with copying mechanisms [S1].

Finally, this work on grammars and on their algorithms has shaped a general architecture for linguistic modeling that takes into account the constraints on expressiveness and algorithmic complexity. The work related to recognizability pointed towards the use of trees as syntactic structures so as to obtain efficient and simple parsing algorithms. Moreover, the relationship between finite state tree automata and Monadic Second Order Logic (MSOL) appeared as a strong leverage in modeling syntactic constraints concisely. With Lionel Clément and Bruno Courcelle, I supervised the PhD thesis of Jérôme Kirman whose work has consisted in the design of a formalism based on logic that would be able to concisely model natural language. This work resulted in a proposal [S4] that could handle elegantly several complex linguistic phenomena in several languages (English, Dutch, German) and also in semantics. Kirman's work proposed also some direction to treat free word order phenomena in that context [175].

Verification

My work on verification started as part of the ANR project FREC where I started a collaboration with Igor Walukiewicz. This work follows the idea of seeing executions of programs from an abstract point of view. An example of this approach is the modelization of first order programs with the transition

systems generated by pushdown automata. In this context, verifying some of the program properties can be reduced to verifying properties of the transition system. The transition system forms a sort of window on the actual semantics of the program.

In my work, I take a similar point of view in leaving the low level operations performed by a program as uninterpreted and model its control flow with higher-order constructions. Then, programs are abstractly modeled as λY-terms whose executions result in compositions of low level operations that form a possibly infinite tree, its Böhm tree. Some properties of the program can be reduced to syntactic properties of Böhm trees. Among these properties, we can mention resource usage, liveness and fairness properties. My work uses Monadic Second Order Logic as the class of syntactic properties of Böhm trees we want to verify. This logic is standard when it comes to verification problems. It is at the same time highly expressive and also often yields decision procedures.

Historically, setting MSOL verification problems in the context of λY-calculus took a rather long time (several decades). Indeed, this problem is at the cross-road of several communities. On the one hand, it is based on the idea of Scott and Elgot [113] that programs interpretation could be performed via the syntactic intermediate step that I explained above. On the other hand, it follows a series of results related to MSOL which started with Rabin's seminal result [247] which was gradually applied to richer and richer classes of machines. Finally the verification of MSOL specifications against higher-order programs only started in the beginning of 2000's with the work of Knapik *et al.* [180]. This work was mostly related to formal language theory through its use of generalizations of pushdown automata: Higher-Order Pushdown Automata (HOPDA). This work has been generalized by Ong [233] who showed that the MSOL theory of Böhm trees generated by higher-order schemes, or said differently of λY-terms, is decidable. This result opened the possibility of the behavioral verification of higher-order programs.

My starting point in that field has consisted in trying to simplify Ong's result and clarify the tools on which it was based. This led me to give a new proof of Ong's results based on finite models of the λ-calculus and on Krivine machines [S29]. The simplicity of the proof allowed to generalize Ong's result and restate it in terms of logical transfers [S27]. This result shows in a very general setting that the MSOL theory of the Böhm tree of a λY-term M can be reduced to the MSOL theory of M itself. Moreover, the Krivine machine happened to be a very good tool to understanding abstract machines that had been proposed by the other approaches [S30].

The result inclined me to believe that all the theory could be recast in term of recognizability. This line of research was appealing as it is connected to many interesting problems related to the algebraic understanding of finite state properties of trees. I started by investigating the class of properties that were captured by finite Scott models [S16]. I showed in particular that they could only capture safety and reachability properties but could not take into account whether a computation was unproductive. I modified Scott models so as to

make them sensitive to unproductive computations. I could then generalize the construction so as to obtain models that capture the properties of the weak MSOL (a fragment of MSOL that is a bit less expressive) [S15]. Finally, the construction of a model that captures the entire logic requires intuitions coming from linear logic so as to summarize some control flow information in a finite way. These intuitions come from the work of Grellois and Melliès [141, 142]. I could recently give a construction for such models [S26].

1.3 Organization of the document

As this document is a synthesis of already produced research and a presentation of the perspectives this research opens, all the parts of the work I report are not given with the same level of detail. First of all no proof is fully detailed. Proofs can be found in the published papers. I have decided to present the general lines of certain proofs and constructions because they are representative of a general method or because they present some interesting originality. As much as I could, I have tried to present my work from a general perspective, but sometimes at the cost of only giving a very brief presentation of certain results. This is the price to pay for making this document relatively concise.

The document is organized in four Chapters. The first chapter introduces notations and basic notions related to simply type λ-calculus. Though simple these notions happen not to be well-known outside the community of its specialists. This chapter also presents the notion of recognizability cast in the setting of simply typed λ-calculus. It contains also the basic properties related to that notion. The second chapter presents my work on abstract categorial grammars. It presents result about its expressiveness and then explains how recognizability allows to construct parsing algorithms for them. This approach is then generalize to higher-order OI grammars for which we explain how to devise a parsing algorithm only using some semantic methods. The third chapter presents my work on mildly context sensitive languages. I present here a personal point of view on this notion that motivated the PhD subject of Jérôme Kirman. I then present some technical results: MIX being a 2-MCFL and that MCFLs are not iterable in general. The fourth chapter presents my contribution to program verification. It gives the motivations of studying programs from the syntactic traces they generate. It then present some construction of models of λ-calculus for certain kinds of specification. Interestingly for proving the correctness of the model for the most complex class of specifications I present, the standard denotational methods of λ-calculus could not be used successfully and I present a method based on the compression of parity games. The document ends with a conclusion that draws the lines for future work.

Chapter 2

Preliminaries and recognizability

This chapter is essentially meant to provide the main definitions and notations that are used throughout the document. As they are mostly standard, the knowledgeable reader should feel free to quickly skim trough it. It mostly presents λ-calculus in its simply typed version. λ-calculus has been introduced by Church [78] as a way of modeling the notion of functions. Church then designed simply typed λ-calculus as a mean of representing higher-order logic [79] which serves, as we will see in the next chapter, as a basis of the formal semantics of natural language. Simply typed λ-calculus has subsequently been used to define the idealized functional programming language PCF proposed by Scott, Milner [216, 215] and Plotkin [243]. PCF has attracted a lot of attention with the question of giving a mathematical description of sequentiality through the *full abstraction problem* (see [246] for a quick presentation of this problem).

In this chapter we introduce the notion of higher-order signatures, and some special constants such as the fixpoint operators, the non-deterministic choice operator and the undefined constants. Whether we authorize the use of those special constants, or only part of them give rise to various calculi. This explains why we may use "simply typed λ-calculi" to the plural when we wish to refer to those calculi without distinction. After we have introduced simply typed λ-calculi, we introduce their denotational semantics in the guise of Henkin models, standard models and monotone models. In Chapter 5, we introduce some other kinds of models, that will be related to monotone models. Finally, the chapter ends with a presentation of the notion of recognizability in the simply typed λ-calculi. It extends the ideas related to finite state automata for strings or trees. In the context of simply typed λ-calculi, this notion relates syntactic problems to semantic ones and makes the tools of semantics available to their studies. This document will present some applications of this notion to decision problems in Chapters 3 and 5.

2.1 Simply typed λ-calculus

Simple types are built from finite set of atoms. For such a set $\mathcal{A}$, the set of types $\mathcal{T}(\mathcal{A})$ is inductively constructed as follows: every A in $\mathcal{A}$ is in $\mathcal{T}(\mathcal{A})$, and for every A, B in $\mathcal{T}(\mathcal{A})$, $(A \rightarrow B)$ is in $\mathcal{T}(\mathcal{A})$. The arrow is associating to the right and we write $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$ for $(A_1 \rightarrow (\ldots(A_n \rightarrow B)\ldots))$. Intuitively, the type $A \rightarrow B$ is meant to denote the functions from A to B. The order of types is a measure of their functional complexity, whether the type is representing some data, some functions over data, some functionals parametrized by functions and so on... It is inductively defined by $\mathit{ord}(A) = 1$ when A is an atom and $\mathit{ord}(A \rightarrow B) = \max(\mathit{ord}(A) + 1, \mathit{ord}(B))$. For this definition, we follow the convention adopted by syntactic traditions of λ-calculus related to the problems of higher-order unification or of higher-order matching and that has been initiated by Huet [155]. The semantic community also uses a notion of order for similar purpose which is equal to $\mathit{ord}(A) - 1$.

We use typed constants either to model operations of a programming language, but we will also see that they can be used to model sets of letters for building languages of strings or ranked alphabets for building languages of trees. Typed constants are declared in higher-order signatures. A higher-order signature Σ is a triple $(\mathcal{A}, \mathcal{C}, \tau)$ where $\mathcal{A}$ is a finite set of atomic types, $\mathcal{C}$ is a finite set of higher-order constants and τ is a function that assigns types in $\mathcal{T}(\mathcal{A})$ to constants. So as to simplify the notation, for a higher-order signature $\Sigma = (\mathcal{A}, \mathcal{C}, \tau)$, we may write $\mathcal{T}(\Sigma)$ instead of $\mathcal{T}(\mathcal{A})$.

Fixing a higher-order signature Σ, we assume that for each type we have an infinite countable set of λ-variables; we also assume that those sets are pairwise disjoint. In general, we write x^A to emphasize the fact that the variable x is of type A. Under those assumptions, a higher-order signature defines a family $(\Lambda_A(\Sigma))_{A \in \mathcal{T}(\Sigma)}$ where $\Lambda_A(\Sigma)$ is the set of λ-terms of type A. When Σ is clear from the context, we may simply write Λ_A. These sets are the smallest sets so that:

- x^A is in Λ_A,
- if c is a constant of type A (i.e. $\tau(c) = A$), then c is in Λ_A,
- if M is in $\Lambda_{A \rightarrow B}$ and N is in Λ_A, then (MN) is in Λ_B,
- if M is in Λ_B, then $(\lambda x^A.M)$ is in $\Lambda_{A \rightarrow B}$.

Simply typed λ-calculus maybe seen from two perspectives, a first one is that of a toy programming language, a second is that of a theory of binding that sets the ground for higher-order logic [79] which is instrumental in Montague's approach to the semantics of natural language [219]. The main use of simply typed λ-calculus we will have in this document is that of a small programming language. It can be either a way of defining some syntactic operations on various objects, or a way of assembling certain operations so as to perform a computation.

We take for granted the notions of free and bound variables, and we write $FV(M)$ to denote the set of variables that have a free occurrence in M. We freely use the standard notational conventions that allow us to remove useless parentheses. As it is usual, we always consider λ-terms up to α-conversion, that is up to the renaming of bound variables. We also assume the notion of capture-avoiding substitution, and we write $[x_1 \leftarrow N_1, \ldots, x_n \leftarrow N_n]$ for the simultaneous capture-avoiding substitution of N_1 for x_1, ..., and of N_n for x_n. Even though these conventions are well-understood and fairly intuitive, they are technically difficult and require careful definitions which are not the focus of this document. So we shall remain about these problems at an informal level.

The operational semantics of λ-calculus is β-contraction that is defined as:

$$(\lambda x.M)N \rightarrow_\beta M[N/x] \tag{2.1}$$

The reflexive transitive closure of β-contraction is called β-reduction ($\xrightarrow{*}_\beta$), and the symmetric closure of β-reduction is called β-conversion $=_\beta$. An important result concerning simply typed λ-calculus is that it is strongly normalizing modulo β-contraction. This means that the relation of β-contraction is well-founded: no matter how a term is reduced, it always gets closer to a normal form. This result due to Tait [280], introduces some fundamental ideas which were subsequently used to prove the consistency of type theories, most notably that of system F which has been proved by Girard [135].

Another important and older result that concerns λ-calculus and that also holds in the untyped case is the Church-Rosser property of β-contraction, i.e. the relation is confluent [80]. In conjunction with the strong normalization Theorem, this property implies that each simply typed λ-term has a unique normal form.

We also consider λ-terms up to η-conversion that is defined by:

$$(\lambda x.Mx =_\eta M) \text{ when } x \notin FV(M) \tag{2.2}$$

As η-conversion is a *non-local* rule in the sense that before being able to η-contract a term one needs to verify the side condition that the variable is not free in the term. It is usual, in the context of simply typed λ-calculus, to work with terms in η-long form. Indeed, terms in η-long form are closed modulo β-conversion and moreover two terms are equal modulo $\beta\eta$-conversion iff they have the same η-long, β-normal form. A term M is in η-long form whenever for every context $C[]$ and term N if $M = C[N]$ and N has type $A \rightarrow B$, then either N is a λ-abstraction and $N = \lambda x^A.N'$, or $C[]$ is an applicative context which provides N with an argument, i.e. for some N' and $C'[]$, $C[] = C'[[]N']$. In other words a term is in η-long form whenever each of its subterm that has an functional type is either a λ-abstraction or is provided an argument by its context. Every term can be put in η-long form: for every term M there is a term M' that is in η-long form so that M' is η-reducible to M. When one mentions *the η-long form* of a term M, one implictely refers to a term in η-long form that is also minimal for η-reduction; this defines a unique term (see [155]).

String and tree signatures

Higher-order signatures can define standard structures such as strings or trees. A signature is called a *string signature* when:

- it has a unique atomic type o,
- every constant is of type $o \to o$.

Given a string $a_1 \dots a_n$, it can be represented by the closed λ-term of type $o \to o$, $\lambda x.a_1(\dots(a_n\, x)\dots)$. The function composition, $\lambda s_1 s_2 x.s_1(s_2\, x)$ encodes the concatenation of strings, while the identity function represents the empty word. The associativity of concatenation and the fact that the empty string is its neutral element can be verified simply by using β-reduction.

A signature is called a *tree signature* when:

- it has a unique atomic type o,
- every constant has a type with order at most 2, i.e. is of the form $\underbrace{o \to \cdots \to o}_{n\times} \to o$ which we may write $o^n \to o$.

Closed terms of type o on a tree signature are just the ranked trees that we are used to. If Θ is a tree signature, we may write $\Theta^{(n)}$ to denote the set of constants of type $o^n \to o$ in Θ.

The interest of using these representations of usual data structures inside the simply typed λ-calculus, is that they are immediately immersed into a small programming language in which standard operations can be represented.

2.2 Special constants

Depending on the context, we may add to λ-calculus constants that are not to be declared in signatures. We may refer to these constants as *special constants*. These constants come with a computational meaning and have thus a particular status. These constants are indexed by types and thus there are infinitely many of them:

Fixpoint operators for every A, Y^A is a constant of type $(A \to A) \to A$, and the δ-contraction is defined by:

$$Y M \to_\delta M(Y M) \tag{2.3}$$

Undefined constants for every Ω^A is a constant of type A, the ω-contraction is defined by,

$$\Omega^{A\to B} N \to_\Omega \Omega^B \tag{2.4}$$

Non-deterministic operator for every A, $+_A$ is a constant of type $A \to A \to A$ (it will be written with an infix notation), their reduction rules are given by:

$$M_1 + M_2 \to_+ M_1 \qquad M_1 + M_2 \to_+ M_2 \tag{2.5}$$

When we find it necessary to specify with which constants we wish to work, we add them to the name of the calculus. With this convention, the $\lambda Y\Omega$-calculus is the λ-calculus where we may use the fixpoint operators and the undefined constants, other calculi will designated this way.

Remark 1 When we work with the Y combinator, it is sometimes convenient to use it as a binder. In that case we write $Yx.M$ for $Y(\lambda x.M)$.

When working with a calculus, we may simply write $\rightarrow$ for the union of the contraction rules of its special constants and β-contraction. Similarly we may use $\stackrel{*}{\rightarrow}$ and $=$ respectively for the reduction and the conversion relations. We may also restrict our attention to a subset of the contraction rules, and in that case, we will write the contraction, reduction and conversion relations with the adequate indices. For example when dealing with $\lambda Y+$-calculus and only working with β and δ-contraction rules, we will write the relations $\rightarrow_{\beta\delta}$, $\stackrel{*}{\rightarrow}_{\beta\delta}$ and $=_{\beta\delta}$.

Except for the rules concerning the non-deterministic operator, all those reduction rules preserve the Church-Rosser property. And except for δ-reduction, all those rules preserve the strong normalization property of simply typed λ-calculus. There are terms that do not have a normal form. Böhm trees [47] provide a way of defining a notion of infinitary normal forms for terms in calculi that involve the fixpoint operators. The Böhm tree $BT(M)$ of a term M is the (possibly infinite) tree defined as follows:

- if $M \stackrel{*}{\rightarrow}_{\beta\delta} \lambda x_1 \dots x_n.hM_1 \dots M_n$, with h being either a constant (not a special one) or a variable, then $BT(M)$ is the tree whose root node is labeled $\lambda x_1 \dots x_n.h$ and which has $BT(M_1)$ as first daughter, ..., and $BT(M_n)$ as n^{th} daughters,

- otherwise $BT(M)$ is a one node tree whose root is labeled Ω_A (A is the type of M).

The construction of Böhm trees is based on $\beta\delta$-reduction; this ensures its uniqueness by the Church-Rosser property of this relation. In case the we consider a calculus with the undefined constants, adding its contraction rule does not modify the definition of Böhm trees.

We have given a syntactic definition of Böhm trees that has the merit of being rather intuitive. Nevertheless, we should mention that they can be defined using Cauchy series under a suitable metric on terms [283]; probably the most elegant way of defining Böhm trees is by ideal completion of syntactically order terms using Ω as the least term (see [246]).

Homomorphism

A homomorphism h from a signature Σ_1 to a signature Σ_2 maps atomic types of Σ_1 to types of Σ_2, for functional types we have $h(A \rightarrow B) = h(A) \rightarrow h(B)$. Moreover, it then maps terms in $\Lambda(\Sigma_1)_A$ to terms in $\Lambda(\Sigma_2)_{h(A)}$ as follows:

- $h(x^A) = x^{h(A)}$,
- $h(MN) = h(M)h(N)$,
- $h(\lambda x^A.M) = \lambda x^{h(A)}.M$,
- $h(c)$ is a closed term of Σ_2 of type $h(\tau_1(c))$.

We mainly deal with homomorphisms in contexts where there is no special constants. Nevertheless, we can extend the definition to those cases simply by letting $h(Y^A) = Y^{h(A)}$, $h(\Omega^A) = \Omega^{h(A)}$ and $h(+_A) = +_{h(A)}$.

A homomorphism h is called *relabeling* when:

- every atomic type is mapped to an atomic type,
- every constant of Σ_1 is mapped to a constant of Σ_2.

2.3 Some syntactic restrictions on simply typed λ-calculus

In certain cases, we are going to work with calculi that satify some syntactic restrictions. These restrictions mainly concern the possible number of occurrences of bound variables. They are a combination of the two following conditions:

relevance every bound variable has at least one free occurrence in the term in which it is bound. More formally, for every term of the form $\lambda x^A.M$, this condition requires that x^A is in $FV(M)$,

non-copying every variable has at most one free occurrence in every term. This condition can be ensured by restricting the application construction as follows: if M is in $\Lambda_{A\to B}(\Sigma)$, and N is in $\Lambda_A(\Sigma)$, then MN is non-copying when M and N are non-copying and $FV(M) \cap FV(N) = \emptyset$.

Terms that satisfy both conditions are called *linear*. Those that satisfy only the relevance condition are called *relevant* or λI-terms. Those that satisfy only the non-copying property are called *affine*. Finally those which satisfy a variant of the non-copying property that allows only first order variable to have more that one free occurrence in a term are called *almost affine* and *almost linear* when they also satisfy the relevance condition. We extend these restrictions to homomorphisms: a homomorphism is said *linear*, *affine*, *relevant*, *almost linear*, or *almost affine* when it respectively maps constants to *linear*, *affine*, *relevant*, *almost linear* or *almost affine* λ-terms.

Another interesting restriction called *safety* has been described by Knapik *et al.* [180] and fully formalized in the setting of λ-calculus by Blum and Ong [57]. While the other restrictions (except almost affinity/linearity) make sense also for untyped λ-calculus, the safety restriction is based on properties related to simple types. This notion is rather technical and we do not wish to give

the most general definition that captures it. Instead, we follow a particular convention about types. We consider that the types involved in the construction of safe λ-terms are *homogeneous types*, i.e. types of the form $A_1 \to \cdots \to A_n \to o$ where o is atomic, $A_1, \ldots, A_n$ are homogeneous, and for every i in $[1, n-1]$, $ord(A_i) \geq ord(A_{i+1})$. This restriction to homogeneous types is the original restriction adopted by Damm [101] and every safe λ-terms defined with the more general notion of safety rephrased by Blum and Ong [57] can be represented (modulo σ-equivalence, i.e. modulo the permutation of arguments) to a safe λ-term with homogeneous type [218].

Definition 2 Given a homogeneous type A, the set of safe λ-terms of type A are inductively defined by:

- x^A is a safe λ-term of type A,
- if $\tau(c) = A$, c is a safe λ-term of type A,
- if $A = A_1 \to \cdots \to A_n \to B$ with $ord(A_i) = ord(A_j) > ord(B)$ for every i, j in $[1, n]$, and M is a safe λ-term of type B, then $M' = \lambda x_1^{A_1} \ldots x_n^{A_n}.M$ is a safe λ-term of type A when for every y^C in $FV(M')$, $ord(C) \geq ord A$,
- if M is a safe λ-term of type $A_1 \to \cdots \to A_n \to A$ with $ord(A_i) = ord(A_j) > ord(A)$ for every i, j in $[1, n]$, and $N_1, \ldots, N_n$ are safe λ-terms respectively of type $A_1, \ldots, A_n$, then $MN_1 \ldots N_n$ is a safe λ-term of type A.

A consequence of this definition is that every safe λ-term M of type A verifies the property that: if x^B is free in M, then $ord(B) \geq ord(A)$.

An important remark made by Blum and Ong [57] is that for safe λ-terms the traditional capture-avoiding substitution may be faithfully replaced by the syntactic substitution. A reason why capture-avoiding substitution is used in λ-calculus is to prevent the capture of free variables by λ-abstraction: for example when computing the result of the substitution $(\lambda x^A.M^B)[y^C \leftarrow N]$ we wish that if x^A is free in N then it remains free after the substitution. For this we need to rename the bound occurrences of x^A in M^B and in the λ-binder. Now if we consider that the terms are safe, then it means that if y^C is free in $\lambda x^A.M^B$, then $ord(C) \geq ord(A \to B) = \max(ord(A)+1, ord(B)) > ord(A)$. Moreover if z^D is free in N, then we must have $ord(D) \geq ord(C)$ and so $ord(D) > ord(A)$ and it cannot be the case that x^A is free in N which explains why simple substitution can be *safely* used. A nice property of safety, is that it is preserved by substitution and by some variant of β-reduction that reduces simultaneously extended redices, i.e. reduces terms of the form $(\lambda x_1^{A_1} \ldots x_n^{A_n}.M^A)N_1 \ldots N_n$ where $ord(A_1) = \cdots = ord(A_n) > ord(A)$ to $M[x_1 \leftarrow N_1, \ldots, x_n \leftarrow N_n]$.

In the context of λY-calculus, the notion of safety needs to be refined a bit. So as to coincide with the notions that are captured by machines such as Higher-Order PushDown Automata (HOPDA), we need to consider Y-combinators as binders and distinguish two kinds of variables: the *recursive variables* that

can be bound by Y-combinators and the λ-*variables* that can be bound by λ-abstraction. While recursive variables can be used with no restrictions, λ-variables are required to verify the safety restriction. Now, substitutions need to be capture avoiding at least for recursive variables. As it is the case for safe λ-terms, the set of safe λY-terms is closed under $\beta\delta$-reduction.

The work of Werner Damm on the IO and OI hierarchy [101] gives us some details about the nature of the safe λ-calculus. In universal algebra, there is a natural construction that extends an algebra $\mathbb{A}$ into an algebra that is able to describe the polynomials over that algebra. The new algebra, $\mathrm{cl}(\mathbb{A})$, is called the *clone* of the orginal algebra [81]. Other definitions of the same operations have been described in the literature such as Lawvere Theories [201] or the magmoid structure [44, 45]. One may then consider polynomials over the clone of $\mathbb{A}$, i.e. the clone of the clone of $\mathbb{A}$: $\mathrm{cl}^2(\mathbb{A}) = \mathrm{cl}(\mathrm{cl}(\mathbb{A}))$. Now by iterating the construction we obtain a higher-order algebra $\mathrm{HO}(\mathbb{A}) = \bigcup_{i=0} \mathrm{cl}^i(\mathbb{A})$ which is functionally complete in the sense that every polynomial over that algebra can be represented as an element of the algebra. As λ-calculus is functionally complete, this new algebra can be faithfully represented in the simply typed λ-calculus that is freely generated by $\mathbb{A}$. Werner Damm's work shows that the natural embedding of $\mathrm{HO}(\mathbb{A})$ into that λ-calculus coincides with the safe λ-calculus that would be constructed from the operations of $\mathbb{A}$ (see also [218]).

It has been showed that safe λ-calculus is less expressive than λ-calculus in several ways. First of all, Blum and Ong [57] show that the safe λ-calculus cannot define all the extended polynomials on Church numerals contrary to λ-calculus [264]. In particular, they prove that only multivariate polynomials are definable showing that the conditional is not. Another, more difficult, result has been showed by Parys [239]: the class Böhm trees of closed safe λY-terms of atomic type over a tree signature is strictly smaller than the class of Böhm trees of closed λY-terms of atomic types.

2.4 Models of λ-calculi

When working with a programming language, understanding the structures of the properties that are invariant under computation (i.e. the conversion relations) is interesting as it gives methodologies to prove programs correct. It also gives some ways of proving that certain algorithm cannot be represented by a given programming language. An early example of this kind of result is given by Plotkin [243] who showed that the *Kleene or* (also called *parallel or* in the programming language community) could not be programmed in PCF. *Kleene or* is a program with two arguments that returns true whenever one of its argument is true, the other can take any value and possibly be undefined. The research on denotational semantics is focused on the description of those structures. Simply typed λ-calculus because of its close connection with computable functions and its rather simple syntax constitutes a privileged framework in which to study semantics of programming languages.

The kinds of semantics that we will use in this document are rather simple.

We nevertheless have a need for a general definition which is an adaptation to our context of the notion of model developed by Henkin [151]. We will then introduce standard models and Scott models. In each case, we will do it in a finitary setting. A reason is that it simplifies the matter; another one, is that, as we shall see, finite models can be seen as playing the same role for simply typed λ-calculi as finite monoid in formal language theory.

Rather than taking a category theoretic approach to present the models of simply typed λ-calculi, we take a set theoretic approach. This approach is less elegant than the category theoretic one, but it makes it easier to explain models of λ-calculi to a non-expert audience.

For the whole section, we assume that we have fixed a higher-order signature Σ on which terms are built.

Definition 3 (Applicative structures) An applicative structure on Σ, an applicative structure is a pair $(\mathcal{F}, \bullet)$ where $\mathcal{F} = (\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}$ is a family of structures[1] indexed by the simple types of Σ and $\bullet$ is a binary operation so that for every type A and B, given f in $\mathcal{M}_{A\to B}$ and g in $\mathcal{M}_A$, $f \bullet g$ is in $\mathcal{M}_B$.

An applicative structure is said *finite* when for every type $\mathcal{M}_A$ is finite. It is said *effective* when the structure $\mathcal{M}_A$ is can be constructed for every type A and that the operation $\bullet$ is computable.

All the applicative structures that we are to see in this document will be finite and effective.

Applicative structures are the building block of models of λ-calculi. Because, the aim of models is to characterize computational invariants of λ-terms and that λ-terms may contain free variables, it is a necessity to be able to compute the semantics of open terms. For this we need to parametrize semantics of terms with a functions that assigns their meaning to free variables: *valuations*.

Definition 4 (Valuations) Given an applicative structure over a signature Σ $((\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}, \bullet)$, a valuation ν is a partially defined function with a finite domain so that when $\nu(x^A)$ is defined it is in $\mathcal{M}_A$.

Given $f \in \mathcal{M}_A$, we write $\nu[x^A \leftarrow f]$ for the valuation that is equal to ν but that maps x^A to f.

We write $\emptyset$ for the nowhere defined valuation.

We are now in position of giving the definition of what models are.

Definition 5 (λ-models) A model of Σ is a pair $\mathbb{M} = (\mathcal{F}, [\![\cdot,\cdot]\!])$ where $\mathcal{F} = ((\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}, \bullet)$ is an applicative structure over Σ and $[\![\cdot,\cdot]\!]$ is a function that map a λ-term M of type A and a valuation ν whose domain of definition contains $FV(M)$ to an element of $\mathcal{M}_A$ and that satisfies the following identities:

[1] We mean here algebraic structures of the same kind. As usual, we confuse the structure with its carrier. So if we consider a lattice, L, we allow ourselves to write $a \in L$ (and the likes) to mean that a is an element of the carrier of L.

1. $[\![x^A, \nu]\!] = \nu(x^A)$,
2. $[\![MN, \nu]\!] = [\![M, \nu]\!] \bullet [\![N, \nu]\!]$,
3. $[\![\lambda x^A.M, \nu]\!] \bullet f = [\![M, \nu[x^A \leftarrow f]]\!]$ for every f in $\mathcal{M}_A$,
4. $[\![Y^A, \nu]\!] \bullet f = f \bullet ([\![Y^A, \nu]\!] \bullet f)$ for every f in $\mathcal{M}_{A\to A}$,
5. $[\![\Omega_{A\to B}, \nu]\!] \bullet f = [\![\Omega_B, \nu]\!]$ for every f in $\mathcal{M}_A$,
6. if ν_1 and ν_2 are valuation that take the same values on $FV(M)$, then $[\![M, \nu_1]\!] = [\![M, \nu_2]\!]$.

A model is finite when its applicative structure is finite. It is effective when its applicative structure is effective and the semantics of every constants (including special ones, fixpoints,...) is computable.

A model is extensional when for every types A, B and every f_1, f_2 in $\mathcal{M}_{A\to B}$ if for every g in $\mathcal{M}_A$, $f_1 \bullet g = f_2 \bullet g$, then $f_1 = f_2$. Otherwise, it is called intentional. All the model we are going to see are extensional. It will even be the case that $\mathcal{M}_{A\to B}$ will always be a subset of the functions from $\mathcal{M}_A$ to $\mathcal{M}_B$. Thus when we work with models and applicative structure we will often omit the operation $\bullet$ when it is simply function application.

For a given calculus, an element f in $\mathcal{M}_A$, is said *definable* when there is a closed term M in that calculus so that $[\![M, \emptyset]\!] = f$.

We have given a definition that covers $\lambda Y{+}\Omega$-terms. When we use models for a λ-calculus that contains less constants, we will implicitly assume that only the relevant identities are to be verified. The very definition of Henkin models has the consequence that the interpretation of terms in Henkin models is inveriant under conversion.

Theorem 6 *If M and N are two terms of the same type and $M = N$, then for every Henkin model $\mathbb{M} = (\mathcal{M}, [\![\cdot,\cdot]\!])$, and every valuation ν, $[\![M, \nu]\!] = [\![N, \nu]\!]$.*

Before we turn our attention to particular kinds of models, we are going to introduce an important construction that allows to build models in a convenient way: logical relations. Though one could understand Tait's [280] proof of strong normalization of simply typed λ-calculus as being based on logical relations, they have been explicitly introduced by Plotkin [242] so as to study the definable elements of models.

Definition 7 (Logical relations) Given two applicative structures of Σ, $\mathcal{F}_1 = ((\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}, \bullet)$ and $\mathcal{F}_2 = ((\mathcal{N}_A)_{A\in\mathcal{T}(\Sigma)}, \diamond)$, a family of relations $\mathcal{R} = (R_A)_{A\in\mathcal{T}(\Sigma)}$ is called a logical relation between $\mathcal{F}_1$ and $\mathcal{F}_2$ when it verifies:

- for every $A \in \mathcal{T}(\Sigma)$, R_A is included in $\mathcal{M}_A \times \mathcal{N}_A$,
- for every $A, B \in \mathcal{T}(\Sigma)$,

$$R_{A\to B} = \{(f, g) \in \mathcal{M}_{A\to B} \times \mathcal{N}_{A\to B} \mid \forall (a, b) \in R_A, (f \bullet a, g \diamond b) \in R_B\}$$

Two models $\mathbb{M}_1 = (\mathcal{F}_1, [\![\cdot,\cdot]\!]_1)$ and $\mathbb{M}_2 = (\mathcal{F}_2, [\![\cdot,\cdot]\!]_2)$ whose respective applicative structures are $\mathcal{F}_1$ and $\mathcal{F}_2$ are said logically related by $\mathcal{R}$ when for every constant c (including special ones), $([\![c,\emptyset]\!]_1, [\![c,\emptyset]\!]_2)$ is in $\mathcal{R}$.

When we say that two models are in logical relation, we will implicitly assume that their applicative structures are in logical relation.

Lemma 8 (Fundamental lemma) Given $\mathbb{M}_1 = (((\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}, \bullet), [\![\cdot,\cdot]\!]_1)$ and $\mathbb{M}_2 = (((\mathcal{N}_A)_{A\in\mathcal{T}(\Sigma)}, \diamond), [\![\cdot,\cdot]\!]_2)$ that are in logically related by the relation $\mathcal{R} = (\mathcal{R}_A)_{A\in\mathcal{T}(\Sigma)}$, the pair $\mathbb{R} = ((\mathcal{R}_A)_{A\in\mathcal{T}(\Sigma)}, \star), [\![\cdot,\cdot]\!])$ where

- $(f,g) \star (a,b) = (f \bullet a, g \diamond b)$ and
- $[\![M,\nu]\!] = ([\![M, \pi_1 \circ \nu)]\!]_1, [\![M, \pi_2 \circ \nu]\!]_2)$

is a model[2].

We can see logical relations as a way of combining models so as to define new models. It is also a good tool so as to prove properties about models.

We are now going to see simple examples of models of λ-calculi.

Standard models of λ-calculus

Standard models are a set theoretic presentation of models of λ-calculus (without any special constant). They are constructed on top of standard applicative structures.

Definition 9 (Standard applicative structures/Standard model) An applicative structure $((\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}, \bullet)$ is standard when:

- $\mathcal{M}_{A\to B}$ is the set of functions from $\mathcal{M}_A$ to $\mathcal{M}_B$,
- and $\bullet$ is function application, i.e. $f \bullet g = f(g)$.

A model of λ-calculus is said standard when its applicative structure is standard.

Standard models are a direct representation of the intuition that λ-calculus is a theory of function. Actually Henkin [151] introduced the notion of models we have given in Definition 5 as an alternative to standard models. Indeed, in Church simple theory of types [79], if terms are interpreted in standard models, then Gödel's incompleteness theorem [137] implies that there are terms that are valid in any standard models but which cannot be equated to truth using the axioms of higher-order logic. The situation of higher-order logic is then different from the one of first order logic where validity in standard models coincides with provability [136]. Henkin introduced a new notion of models so as to make the interpretation in those models coincide with deduction in higher-order logic.

[2] Above π_1 and π_2 denote the first and second projections: $\pi_1(a,b) = a$ and $\pi_2(a,b) = b$.

A natural question arises about the properties of λ-calculus that standard models can capture. In particular, does standard model capture $\beta\eta$-convertibility? Friedman [129] answered positively to this question.

Theorem 10 (Friedman [129]) *For every λ-terms M and N with the same type, the following properties are equivalent:*

- $M =_{\beta\eta} N$*,*
- *for every standard model $\mathbb{M} = (\mathcal{F}, [\![\cdot,\cdot]\!])$ and every valuation ν whose domain of definition contains $FV(M) \cup FV(N)$, $[\![M,\nu]\!] = [\![N,\nu]\!]$.*

Statman [274, 275] showed that this result can be refined and that finite standard models are enough to characterize $\beta\eta$-convertibility. Actually Statman proves a slightly stronger theorem. He proves that for a given term M there is a *characteristic* model, a model in which any term that has the same interpretation as M is $\beta\eta$-convertible to M.

Theorem 11 (Statman's finite completeness [274, 275]) *For every term M of type A, there is a finite standard model $\mathbb{M} = (\mathcal{F}, [\![\cdot,\cdot]\!])$ a valuation ν on $\mathbb{M}$ whose domain is $FV(M)$, so that for every term N of type A, the following are equivalent:*

- $M =_{\beta\eta} N$*,*
- *for every valuation ν' that is equal to ν on its domain and whose domain contains $FV(N)$, $[\![M,\nu']\!] = [\![N,\nu']\!]$.*

An important property of this Theorem is that the construction of the characteristic model of a term is effective.

It has been conjectured by Plotkin and Statman [242, 274] that definability in standard models was decidable. This conjecture has been proved false by Loader [207].

Theorem 12 (Loader [207]) *The problem whether, given a finite standard model $\mathbb{M} = (\mathcal{F}, [\![\cdot,\cdot]\!])$ and given an element f of $\mathbb{M}$, there is a closed λ-term M with the same type as f so that $[\![M,\emptyset]\!] = f$ is undecidable.*

Monotone models of $\lambda Y{+}\Omega$-calculus

We are now going to introduce lattice-based semantics of $\lambda Y{+}\Omega$-calculus. As we will always be working with finite models, the presentation slightly departs from the standard notion of Scott models one can find in the literature. Indeed, semantics based on partially ordered set are mainly used to account for full-fledged programming languages which require the modeling of infinitary objects such as partial functions from natural numbers to natural numbers. Then semantics is mainly concerned about relating the finiteness of programs and of programs stepwise execution to their possibly infinitary semantics. This

is in general done by considering directed complete partial order or lattices. Here as we work with finite models, things are drastically simplified. This is in particular due to the fact that fixpoint computations always converge in finitely many steps in such models.

Definition 13 (Monotone Applicative structures and models) An applicative structure $((\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}, \bullet)$ is called monotone when:

- for every A, $\mathcal{M}_A$ is a finite lattice,
- $\mathcal{M}_{A\to B}$ is the set of monotone functions from $\mathcal{M}_A$ to $\mathcal{M}_B$, i.e. the set of functions f so that for every $a, b \in \mathcal{M}_A$, if $a \leq b$, then $f(a) \leq f(b)$. This set is ordered pointwise, i.e. for every $f, g \in \mathcal{M}_{A\to B}$, $f \leq g$ iff for every $a \in \mathcal{M}_A$, $f(a) \leq g(a)$,
- and $\bullet$ is function application, i.e. $f \bullet g = f(g)$.

A model of $\lambda Y{+}\Omega$-calculus is said monotone when its applicative structure is monotone and when it satisfies the following properties:

- $[\![+_A, \emptyset]\!](f_1)(f_2) = f_1 \vee f_2$, and
- either $[\![\Omega_A, \emptyset]\!]$ is the greatest element of $\mathcal{M}^A$ and $[\![Y^A, \emptyset]\!](f)$ is the greatest fixpoint of f,
- or $[\![\Omega_A, \emptyset]\!]$ is the least element of $\mathcal{M}^A$ and $[\![Y^A, \emptyset]\!](f)$ is the least fixpoint of f.

In the former case, the model is called a greatest fixpoint model (GFM) and latter case, it a least fixpoint model (LFM).

If we are concerned with $\lambda Y\Omega$-terms, then Statman theorem extends nicely. In an untyped context this extension has been proved by means of intersection types by Dezani *et al.* [105]. To the best of our knowledge, this theorem has not been stated yet in the literature but some minor adaptation of Statman's argument and of Dezani *et al.* is proving this statement.

Theorem 14 *For every two λY-terms of the same type, M and N, the following properties are equivalent:*

- $BT(M) = BT(N)$,
- *for every LFM, $\mathbb{M} = (\mathcal{F}, [\![\cdot,\cdot]\!])$, and every valuation ν, $[\![M,\nu]\!] = [\![N,\nu]\!]$.*

By duality between LFMs and GFMs, this theorem may also be stated by replacing LFMs with GFMs.

In monotone models, we will be interested by a particular kind of functions: *step functions.*

Definition 15 Given a monotone applicative structure, $(\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}$, $f \in \mathcal{M}_A$ and $g \in \mathcal{M}_B$, we define the step function $f \mapsto g$ in $\mathcal{M}_{A\to B}$ as the function so that:

$$(f \mapsto g)(h) = \begin{cases} g \text{ if } h \geq f \\ \bot \text{ otherwise} \end{cases}$$

With monotone models, we cannot go beyond Böhm tree equality as any term that is non-convergent has Ω as Böhm tree and that there are non-convergent terms that are not $\beta\delta$-convertible. It has even been showed by Statman that $\beta\delta$-convertibility is an undecidable property [276].

The theorem may be also refined by ordering Böhm trees in the usual way. A Böhm tree t is smaller from another Böhm tree u, which we write $t \leq u$, when t can be obtained from u by replacing some subtrees of u by Ω.

Theorem 16 (Finite Böhm completeness) *For every two λY-terms of the same type, M and N, the following properties are equivalent:*

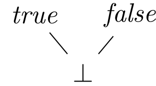
- $BT(M) \leq BT(N)$,
- *for every LFM, $\mathbb{M} = (\mathcal{F}, [\![\cdot,\cdot]\!])$, and every valuation ν, $[\![M,\nu]\!] \leq [\![N,\nu]\!]$.*

Moreover, monotone models are strong enough to decide the convergence of $\lambda Y\Omega$-terms, i.e., whether the Böhm tree of a term is Ω. For this it is sufficient to take an LFP model whose monotone applicative structure is associating the two elements lattice $\mathbf{2}$ to atomic types and which interprets every constant c as the maximal element of the lattice associated to the type $\tau(c)$. We write $\mathbb{2}$ for this model.

Theorem 17 *For every term M, the following properties are equivalent:*

- *$BT(M)$ is the one node tree labeled Ω,*
- *$[\![M,\nu]\!] = \bot$ in $\mathbb{2}$.*

Loader has showed [206] that the definability problem in the case of λY-calculus is not decidable. Actually he proved that the observable equivalence problem for finitary PCF is undecidable. Finitary PCF is a programming language where the only data are booleans and the control structures are the conditionals and the fixpoint operators. So the only atomic type is *bool* and a natural semantics for this language is the LFP model where the lattice associated to *bool* is depicted by:



Two closed terms of type A, M and N are observably equivalent with for every term P of type $A \to bool$, $PM = PN$. This result entails the undecidability of the definability problem in monotone models as if the definability

problem were decidable, one could solve the observability problem simply by checking whether there is a definable element f of the model at type $A \to bool$ which takes different values when applied to the semantics of M and to the semantics of N.

Theorem 18 *The definability problem in the λY-calculus for monotone models is undecidable.*

2.5 Recognizability in simply typed λ-calculus

Finite state automata have a wealth applications in computer science, they have thus undergone thorough research investigations. One of the outcome of this activity is the connection of finite state machine with algebra and logic. It has been showed that the class of languages definable by finite state automata is the same as the class of languages that are recognized by finite monoids and also the same as the class of languages definable with Monadic Second Order Logic (MSOL). The first equivalence is due to Myhill-Nerode Theorem [226, 229], the second equivalence was independently proved by Büchi [70], Elgot [114] and Trakhtenbrot [286]. Another equivalent presentation of finite state automata is by means of regular expressions was proposed by Kleene [178, 177]. Kleene's presentation can be rephrased in terms of finite set of regular equations (left (or right) linear ones) that define languages which are the smallest (for the inclusion) languages verifying those equations.

When dealing with objects other than strings, one may try to adapt these definitions. In certain cases, they coincide, while in other they don't. If one is interested in languages of finite trees, then, again, finite state tree automata [108, 284], finite algebras [62, 63, 42, 112], MSOL [108, 107, 285] and regular expressions all define the same class of languages. In many cases, these notions define different classes of languages as it is the case for arbitrary monoids [112]. In other cases then, only certain of those notions make sense. A typical example is the case of graphs that has been studied at length by Courcelle [91]. It is very unclear here how to define a finite state machine that would recognize a language of graphs in a sense that would arguably be a canonical generalization of what finite state automata for string or tree languages. It is also rather difficult to come up with a notion of finite algebraic interpretation of graphs that would generalize finite monoids and finite algebras. So Courcelle's proposal has been to define a notion of recognizability for graphs that is based on MSOL[3]. Thus a *regular language of graphs* is a set of graphs that satisfies a given MSOL formula. Here this notion of regularity departs from that of string and trees in that the satisfiability of of MSOL formulae over graphs is undecidable, or said in a different way, the emptiness problem for this notion of regular sets of graphs is undecidable. Then Courcelle, based on Robertson and

[3]Actually it is possible to define two kinds of MSOL theory of graphs, one for which one may quantify only on vertices and the other where it is also possible to quantify over edges. We do not enter those details so as to make the discussion simpler.

Seymour's graph minors theorem, showed that if one restricted his attention to particular classes of graphs, then the emptiness problem becomes decidable. These classes of graphs are the classes for which a parameter called treewidth is bounded. A very interesting side-effect of considering graphs with bounded treewidth is that they can be defined with an algebra with finitely many generators and that moreover, the MSOL theory of graphs represented by terms of that algebra can be reduced to the MSOL theory of those terms, i.e. to finite states automata on trees. Courcelle and Durand [89] are now trying to take advantage of this connection to algorithmically solve combinatorial problems about graphs.

In our case, we are interested by extending the notion of recognizability to λ-calculi. For this, logic is not so convenient as it should be on normal forms of terms and that it has to deal with the bindings that are present in the terms. It thus remains the algebraic and the automata theoretic approaches. Finite models provide a natural extension of the finite algebraic approaches for strings and trees. For this it suffices to define recognizable languages of λ-terms as languages of terms whose interpretations are certain elements in a finite model. Of course, as we work in a typed setting, we only consider languages of terms that all have the same type.

Definition 19 (Recognizability) A set L of closed terms of types A, is said recognizable whenever there is a finite model $\mathbb{M} = ((\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}, \bullet), [\![\cdot,\cdot]\!])$, and $R \subseteq \mathcal{M}_A$ so that:

$$L = \{M \mid [\![M, \emptyset]\!] \in R\}$$

In the original definition given in [S23] the notion of recognizability was restricted to finite standard models. Afterwards, it has become clear that standard models were not so convenient to work with. Nevertheless, for the simply typed λ-calculus (without special constant) using logical relations one easily establishes that recognizability with standard models is equivalent to recognizability with any extensional model.

Lemma 20 A set L of closed λ-terms is recognizable by a finite extensional model iff it is recognizable by a finite standard model.

As a corollary of Loader's theorems of this definition is that the emptiness problem for regular languages of λ-terms is undecidable. The situation is similar to that of graphs. But then we obtain several closure properties that are similar to the ones that are known for regular languages of strings or of trees [S23].

An important fact is that when considering terms of type $o \to o$ built on a string signature, the notion of recognizability on λ-terms coincide with the usual notion of recognizability on strings. And this is also the case for tree signatures and recognizable languages of trees. In this sense, recognizability in simply typed λ-calculus can be seen as a conservative extension of the notions of recognizability on strings and on trees.

Theorem 21 *Recognizable languages of λ-terms are closed under the following operations:*

- *union, intersection, complement,*
- *inverse homomorphic image.*

As showed in [S23], these simple remarks, Statman finite completeness theorem and some simple syntactic properties of fourth order λ-terms allows us to give a very simple proof that 4th order matching is decidable that has been originally proved by Padovani [236, 235]. Moreover, this method gives a satisfactory explanation as to why the automata based approach of Comon and Jurski [82] works for fourth order matching and does not generalize at the fifth order. Now higher-order matching has been showed decidable by Stirling [278, 279] based on some game theoretical approach. The proof remains highly combinatoric and technical though. In the light of how recognizability approach simplifies the understanding of the decidability of fourth order matching, it would be nice to connect Stirling's proof with denotational semantics. Such an approach as been already proposed by Laird in an unpubliched note [196]. Laird's subsequent paper [195] that the bidomains model proposed by Berry [55] is fully abstract for unary PCF suggests that one could define a fully abstract model of simply typed λ-calculus that would be effectively presented and thus imply the decidability of the problem.

An important difference between the notions of recognizability for strings and trees and recognizability for simply typed λ-calculus, is that the latter is not closed under relabeling. This problem is related to bound variables which introduce an element of unboundedness that finite model cannot tame (see [S23]).

The motivation for defining a notion of recognizability for the simply typed λ-calculus came from the description of parsing algorithms for abstract categorial grammars that will be the subject of the next chapter. In this study, the goal was to prove the decidability of the membership problem for a particular kinds of ACGs. The first results were obtained by means of typing rather by means of models. The idea was to use intersection types so as to capture syntactic properties of terms in the language [S21]. From this work and the proof methods that it was relying on, it was clear that intersection types were playing a similar role with respect to λ-calculus as the one of finite state automata for strings. It then became important to make this remark formal so as to integrate finite state methods and higher-order ones.

Intersection types have been introduced in [48] as a syntactic approach of Scott model D_∞^* of untyped λ-calculus [265]. These kinds of systems have then been used as means of proving properties of terms such as strong normalization, solvability etc... In our case, we are interested in the use of intersection types so as to capture certain properties of simply typed terms. Thus we do not use any intersection types, but only those intersection types that have a shape similar to simple types.

Definition 22 (Intersection Types) A family $(\mathcal{I}_A)_{A\in\mathcal{T}(\Sigma)}$, is family of intersection types when:

- $\mathcal{I}_A$ is finite when A is an atomic type,
- $\mathcal{I}_{A\to B}$ is the set $\{(Q,p) \mid Q \in \mathcal{P}(\mathcal{I}_A) \wedge p \in \mathcal{I}_B\}$.

We shall write $Q \to p$ for the element (Q,p) in $\mathcal{I}_{A\to B}$.

An intersection type system is a pair $(\mathcal{I},\rho)$ where $\mathcal{I}$ is a family of intersection types, and ρ is mapping that associates a subset of $\mathcal{I}_{\tau(c)}$ to each constant c.

A type-environment is a mapping with finite domain that associates a subset of $\mathcal{I}_A$ to variables x^A. We may write $x_1 : P_1, \ldots x_n : P_n$ to denote the type-environment that maps x_1 to P_1, ..., and x_n to P_n

We write $\Gamma \vdash M : p$ to mean that M has type p in the type-environment Γ according to the derivation rules given in Figure 2.1[4].

In Figure 2.1, we have not given rules for special constants as in the sequel, we prefer to work with models than with intersection types. The reason for it is mainly that intersection types can be seen as a syntactic representation of models. Already at the level of untyped λ-calculus, their introduction was meant to give a syntactic representation of models. In the sequel we favor the model approach over the typing approach as in the context of λ-calculus it gives a clearer understanding of the solutions to the problems we work on by revealing the structure of the invariants we use.

$$\frac{p \in P}{\Gamma, x : P \vdash x : p}\text{Ax.} \qquad \frac{p \in \rho(c)}{\Gamma \vdash c : p}\text{Const.}$$

$$\frac{\Gamma, x : P \vdash M : p}{\Gamma \vdash \lambda x.M : P \to p}\text{Abs.} \qquad \frac{\Gamma \vdash M : P \to p \quad \forall q \in P, \Gamma \vdash N : q}{\Gamma \vdash MN : p}\text{App.}$$

$$\frac{\Gamma \vdash M : p \quad p \sqsubseteq q}{\Gamma \vdash M : q}\text{Sub.}$$

$$\frac{}{p \sqsubseteq p} \qquad \frac{q \sqsubseteq p \quad \forall q' \in Q. \exists p' \in P. p' \sqsubseteq q'}{Q \to q \sqsubseteq P \to p}$$

Figure 2.1: Intersection type derivation rules

Intersection types capture properties that are invariant under $\beta\eta$-conversion.

Lemma 23 For every terms M and N, if $M =_{\beta\eta} N$ and $\Gamma \vdash M : p$, then $\Gamma \vdash N : p$.

[4]In Figure 2.1, we implicitly assume that the subsumption relation $\sqsubseteq$ is only comparing intersection types that belong to the same set $\mathcal{I}_A$.

Another thing to notice is that when we work with terms in η-long form, then the subsumption rule (Sub.) can be derived from the others.

Lemma 24 For every term M, if M is in η-long form and $\Gamma \vdash M : p$ then $\Gamma \vdash M : p$ is derivable without using the rule (Sub.).

The connection between intersection types and models is made by translating intersection types to a monotone model. Given an intersection type system $((\mathcal{I}_A)_{A\in\mathcal{T}(\Sigma)}, \rho)$, we construct a monotone model $((\mathcal{M}_A)_{A\in\mathcal{T}(\Sigma)}, [\![\cdot,\cdot]\!])$ so that, if A is an atomic type $\mathcal{M}_A = \mathcal{P}(\mathcal{I}_A)$. We can now translate intersection types into elements of the monotone model:

- if A is an atomic type and $p \in \mathcal{I}_A$, then $p^\bullet = \{p\}$,
- given $P \subseteq \mathcal{I}_A$, then $P^\bullet = \bigvee\{p^\bullet \mid p \in P\}$,
- given $P \to p$ in $\mathcal{I}_{A\to B}$, $(P \to p)^\bullet = P^\bullet \mapsto p^\bullet$.

The translation of types into elements of the monotone model allows us to connect typing properties of terms with semantic ones. But for this we need to translate type-environment into valuations, so given a type environment Γ, we write $\Gamma^\bullet$ for the valuation so that $\Gamma^\bullet(x) = (\Gamma(x))^\bullet$ (here when x is not declare in Γ, we assume that $\Gamma(x) = \emptyset$ so that $\Gamma^\bullet(x) = \bot$).

Theorem 25 *Given $p, q \in \mathcal{I}_A$ and $P, Q \subseteq \mathcal{I}_A$, the following properties hold:*

- *$p \sqsubseteq q$ iff $q^\bullet \leq p^\bullet$,*
- *for all $p \in P$ there is $q \in Q$ so that $p \sqsubseteq q$ iff $Q^\bullet \leq P^\bullet$.*

Moreover for every term M, type environment Γ and intersection type p,

$$\Gamma \vdash M : p \textit{ iff } [\![M, \Gamma^\bullet]\!] \geq p^\bullet$$

From the perspective of recognizability, typing disciplines give some nice generalizations of finite states automata. We can then see a parallel with the situation between recognizable string and tree languages: in the context of λ-calculus, finite state machines are represented with intersection types, and finite algebras are represented by finite models.

This translation of intersection types into models also emphasized that two difficult results in the literature about λ-calculus were related [S25]: Loader's result that the definability problem was undecidable and Urzyczyn's result that the inhabitance problem for intersection types was undecidable [287]. The main remark is that the types used by Urzyczyn make it so that the λ-terms that are typable with intersection types are not only strongly normalizable, but also simply typable; moreover the shape of those intersection types is actually representing elements in a finite monotone models. The correspondence between intersection types and monotone models together with logical relations between monotone models and standard models can reduce the definability problem to the inhabitance problem of intersection types and *vice-versa*.

2.6 Conclusion and perspective

The chapter has introduced some of the main tools that we are going to use in the sequel of this document. We have also presented the notion of recognizable languages of λ-terms which is a conservative extension of recognizability for strings and trees. Though simple, this notion will prove to be a powerful tool so as to obtain decidability results. Moreover, by connecting λ-calculus with formal language theory, this notion allows us to transfer various methods from one field to the other. We will see instances of this in the next chapters: as for example in the case of parsing algorithms, or in the definition of a notion of wreath product of models of λ-calculus.

The definition of recognizability in simply typed λ-calculus by means of semantic interpretation in finite models asks for a better understanding of the problem of definability and of the conditions in which it becomes decidable. There are two obvious parameters that have some influence here: (i) the class of terms in which we try to define the function, (ii) the properties of models or of the functions that are to be defined.

The class of terms that are *finitely generated*, in the sense that they are generated by application from a finite set of terms, have a decidable definability problem. A class of terms that does not satisfy this property is that of safe λ-terms. Nevertheless, a careful analysis reveals that we may consider that safe λ-terms are *locally finitely generated*. Let us be a bit more specific: once we fix a set of free variables X and a type A, the set of safe λ-terms in normal form of type A whose set of free variables is included in X is finitely generated (this is induced by the clone underlying structure and the subformula property). This remark also works for the definability in finite LFP and GFP models of λY-calculus as definability can be reduced to definability in $\lambda\Omega$-calculus and is thus decidable. We do not know yet if for other classes models of λY-calculus the definability problem is decidable for safe λY-terms. Finding other classes of λ-terms is somewhat difficult. The question would be whether there is some classes of λ-terms for which the definability problem is decidable but which need infinitely –even for fixed types and fixed set of free variables– many combinators to be defined. The fact that safe λ-calculus does not suffer from the result of Loader, calls for the definition of a fully abstract model of safe PCF. A good starting point, may be Kahn and Plotkin sequential functions on concrete domains [165]. Even though they do not form a model of λ-calculus, they may give some ideas about the structure of a model of the safe λ-calculus. A variant of sequential functions has been proposed by Brookes and Geva [67] which is fully abstract for a subclass of the safe λ-calculus where the arguments of a function may not contain any free variable.

When we consider classes of models for which definability is decidable, we may get some new insight on the full abstraction problem. Indeed, once we get a decidability result for observational equivalence in the $\lambda\Omega$-calculus, the underlying algorithm gives a procedure to construct a fully abstract model. This is what happened with Padovani [234] and Schmidt-Schauss [262] results about the decidability of the minimal model. Then Laird proved that the un-

derlying model was Berry's bi-domain generated by **2**; he also proves that this model is isomarphic to Bucciarelli and Ehrhard's strongly stable functions [68] generated by **2**. The decidability of higher-order matching also seems to point at the fact that somehow the syntactic operations that λ-terms can perform to construct another one can be decided, yet there is no description of the underlying model. The underlying model is made of the definable functions in Statman's syntactic model that characterizes a given λ-term. The decidability of higher-order matching makes us believe that the observational equivalence in Statman's model is decidable. Statman's models quotiented by the observational equivalence, let's call them *syntactic models*, might thus have a decidable definability problem. The main issue is that these models are more complicated than the minimal model. Recall also that the minimal model has been studied by Padovani [234] as a way of solving higher-order matching equations where the right member of the equation is a constant of atomic type.

The fact that the bi-domain or the strongly stable functions generated by **2** only contain definable functions in the $\lambda\Omega$-calculus on the signature $(\{o\}, \{\top\}, \tau)$ where $\tau(\top) = o$, is very surprising. Indeed, in that model, programs can in a certain sense fully explore the control flow of their arguments so as to compute their results. This means that it is possible for them to perform multiple tests on their arguments so as to decide which value they can yield. In syntactic models, in general, each occurrence of a constructor in a term limits the number of tests that are possible for that term. It seems very hard to understand how to express axiomatically this limitation. Probably a good idea is to start working on the linear decompositions of these domains as they described respectively in [97] and [110]. In these works the construction an analysis of the exponential modality may help in understanding how to actually express the *cost* of a test when some syntactic constants are traversed. A possible start could be the work we did on higher-order matching in the linear λ-calculus [S24].

An important feature of the syntactic models is that they model particular computations. Not only are they sequential, but they are in a sense top-down deterministic. To make this idea clearer, let us introduce a notion of top-down deterministic models of λ-calculus. They can be seen as an extension of top-down deterministic tree automata. They are constructed from a finite set Q, and they interpret every atomic type by $\mathcal{P}(Q)$, then every every type A and every element q in Q, we define a set of monotone functions $[q, A]$ as follows:

- when A is atomic $[q, A] = \{\{q\}, \emptyset\}$,
- when $A = B \to C$, then $[q, A]$ is the set of monotone functions f so that there is q' in Q, $f_1, \ldots, f_n \in [q', B]$ and $g_1, \ldots, g_n \in [q, C]$ so that

$$f = f_1 \mapsto g_1 \vee \cdots \vee f_n \mapsto g_n \ .$$

We then define the top-down-deterministic applicative structure $(\mathcal{T}_A)_{A\in\mathcal{T}(\Sigma)}$ by taking $\mathcal{T}_A$ to be joins of functions in $\bigcup_{q\in Q}[q, A]$. Such functions f are called top-down deterministic because every state q determines a tuple q_1,

$\dots$, q_n so that for every g_1, $\dots$, g_n, if $f(g_1)\dots(g_n) \geq q$ then $g_1 \in [q_1, A_1]$, $\dots$, $g_n \in [q_n, A_n]$. There are several things to remark about top-down deterministic functions. First of all, the finite model of PCF considered by Loader so as to prove the undecidability of its observational equivalence is not a top-down deterministic model. Second of all, second order functions in the bi-domain of functions in $\mathcal{P}(Q)^n \to \mathcal{P}(Q)$, or second order strongly stable function are top-down deterministic. This leads us to make the conjecture that definability is decidable in top-down deterministic models.

In the case of strings and trees recognizability is equivalent to MSOL. This equivalence is interesting as it allows one to think in terms of properties or specifications at an abstract level. This is an advantage when dealing with modelization tasks as it may be hard to implement and compose by means finite state automata abstract ideas expressed by simple logical formulae. Similarly, in the case of the semantics of natural language, it is sometime interesting to express certain syntactic properties of λ-terms so as to describe properties of semantic contexts. In turn, this allows a more precise description of semantic phenomena. We expect that most of those properties are captured by recognizability. Nevertheless, the fact that recognizable languages of λ-terms are not closed under relabeling makes the problem of defining a logic that would capture the syntactic properties expressed by models a bit delicate as the interpretation of quantification in algebras uses closure under relabeling. A possible candidate is modal μ-calculus on normal forms of λ-terms seen as graphs where there is an edge from binders to the occurrences of variables they bind.

Chapter 3

Abstract Categorial Grammars

Abstract Categorial Grammars (ACGs) have been introduced by de Groote [146] and at the same time Muskens [225] proposed a similar formalism he called λ-grammars. This grammatical formalism can be characterized by two of its main traits: it makes central the notion of syntactic structure and it is based on simply typed lambda-calculus for the description of the operations on object. ACGs implement formally ideas that have been proposed by Curry [98] as early as in 1961.

The first feature is hard to overestimate: it neatly separates the structure in which the relations between the syntactic constituents are represented and their actual incarnation in the language. Doing so, it raises the questions about what the right representations of these objects are and about how to represent abstractly syntactic notions. A natural way of representing syntactic structures consists in using trees as they naturally model the hierarchical structure of syntax on which most linguistic theories are based. Nevertheless, ACGs also allow one to model syntactic structures as linear λ-terms bringing the possibility of modeling the notion of *traces* with higher-order constants. This feature can be particularly useful when modeling certain linguistic theories such as Chomsky's minimalism [77]. In some recent work [S4] for which we are going to present the context in the next chapter, we illustrated how methods based on logic and automata are adapted for the concise description of syntactic structures based on trees.

The second feature of ACGs, the use of λ-calculus so as to represent syntactic operations, is strongly backing the first one. Indeed, if the repertoire of operations is too small, then it becomes tempting to use unnatural syntactic structure so as to model certain linguistic phenomena. In the context of ACGs, λ-calculus provides all the natural syntactic operations on various structures: strings, trees, logical formulae. From a theoretical perspective, using λ-calculus presents another advantage, it reduces the set of primitives used in the definition of the formalism which makes it much simpler to study while highly expressive.

The work we have pursued concerning ACGs has mainly been concerned

with:

- their expressiveness,
- their parsing algorithms,
- their possible extensions.

In this chapter, we will mainly illustrate the first two points. The last one will be exemplified with the extension of ACGs as OI grammars. We have proposed other extensions that we are going to quickly present in the next chapter and which aim at modeling non-configurational languages and free-word order phenomena [S11].

From a technical perspective, we illustrate a general method that connects grammar transformation/analysis and denotational semantics. The overall idea is to see grammars as non-deterministic programs. Most algorithms related to grammars require to use some invariants about the languages that grammars are defining, these invariants are best expressed by means of finite states methods and thus, when λ-calculus is involved, by means of denotational semantics.

3.1 Abstract Categorial Grammars

An ACG is a particular way of defining grammars. We here gives a quick formal presentation of ACGs. An ACG is a tuple $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$ where:

- Σ_1 is a higher order signature, the *abstract signature*,
- Σ_2, is a higher order signature, the *object signature*,
- $\mathcal{L}$ is a homomorphism between Σ_1 and Σ_2, the *lexicon*,
- and S is an atomic type of Σ_1, the *distinguished type*.

The abstract signature is meant to model the notion of syntactic structures; the object signature is meant to model the surface structure of the language. An ACG defines two languages:

- an *abstract language* $\mathcal{A}(\mathcal{G})$, the set of closed linear λ-terms in $\Lambda_S(\Sigma_1)$,
- an *object language* $\mathcal{O}(\mathcal{G})$ the set of λ-terms that are the normal forms of terms that are the image of terms in $\mathcal{A}(\mathcal{G})$ by $\mathcal{L}$.

The restriction to linear λ-terms in the abstract language comes from linguistic motivations. The control of traces and bindings at the level of syntax requires in general the resource-sensitivity of linearity.

The fact that the object signature is a higher-order signature, not only allows one to represent languages of strings or of trees, but more generally languages of λ-terms. Usually, languages of λ-terms are used to describe languages of logical formulae. These logical formulae represent the truth conditions of

the statement represented by the syntactic structures. The overall idea underpinning ACGs is to follow, in all the dimensions of language description, the approach of Montague [219] for the description of the semantics of natural language. This approach is based on the empirical observation that the meaning of natural language is compositional: the meaning of a whole structure is determined by the meaning of its parts and of their relations. If one is to take this notion in a strict sense, then idiomatic expressions pose some problems. Nevertheless, one may consider them as part of a language and give them a compositional treatment. Such a line has be proposed by Kobele [186]. Montague took a radical position concerning compositionality and modeled it using the notion of homomorphism. This consists in taking its weakest possible definition. ACGs try to push this stance as far as possible, and also model the mapping from syntactic structures to surface structures using Montague's notion of compositionality: homomorphism. This homomorphism is called *lexicon* as it provides the relation between syntactic constructs and the vocabulary of the language.

In the definition of ACGs proposed by de Groote in his original paper [146], the definition of ACGs allows the distinguished type to be a non-atomic type of Σ_1 and the lexicon is a linear homomorphism. Here we take a more restrictive definition as it allows us to give a simpler classification of definable languages. Moreover, following de Groote, we call ACGs grammars with a linear lexicon. We then call non-linear ACGs, grammars with any kind of lexicons, and affine ACGs grammars with affine lexicons etc. . . .

We classify grammars with linear lexicons in classes $\mathcal{L}(k, l)$ where:

- k is the order of the abstract signature, and
- λ is the *complexity* of the lexicon, i.e. the maximal order of the types is associates to atomic types of the abstract signature.

We have showed [S22] that when $k > 2$, then for every grammar in the class $\mathcal{L}(k + 1, l)$ there is a grammar in $\mathcal{L}(k, l + 1)$ that defines the same language.

So as to illustrate what ACGs are, we here give an example of a context-free grammar that models a small fragment of English that exhibits a self-embedding phenomenon. We will then turn this fragment into a second order ACG and give it a Montague-like semantic interpretation.

S → NP *VP*	C → that
VP → V NP	N → dog
NP → Det N	N → cat
N → N CP	N → rat
CP → C VP	N → cheese
CP → C S/NP	V → saw
S/NP → NP V	V → chased
Det → a	V → ate

This grammar can analyse sentences like: *a rat that a cat that a dog saw chased ate a cheese*. The derivation tree of that sentence is given Figure 3.1.

Representing this grammar as an ACG, simply amounts to see each rule as

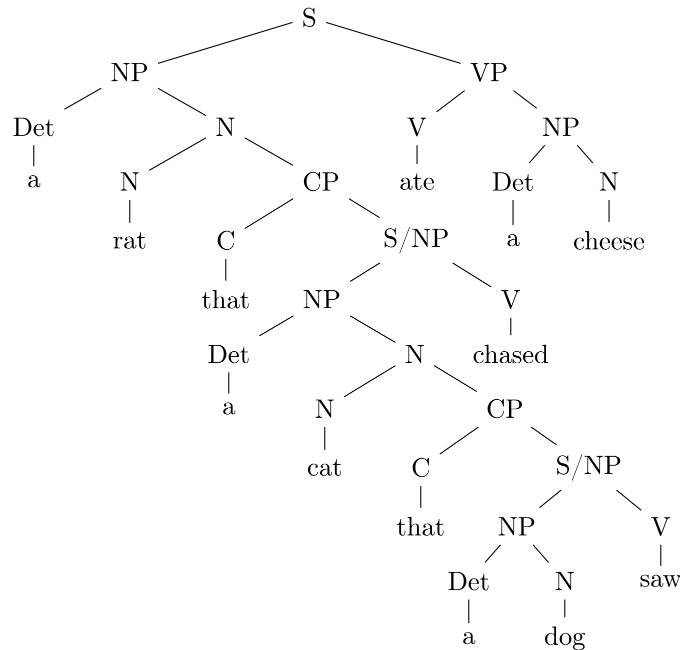


Figure 3.1: Derivation tree of *a rat that a cat that a dog saw chased ate a cheese*

an abstract constant: the types in the abstract signature are just the non-terminals of the grammar, and the abstract constant are rules which are typed according to the non-terminals they produce and the non-terminals they use in their right-hand part. We give the following mnemonic names and types to the abstract constant that represent the rules (each abstract constant is that the same position in that table as the rule it represents in the above table):

SENTENCE	: NP → VP → S	THAT	: C
VERBP	: V → NP → VP	DOG	: N
NOUNP	: Det → N → NP	CAT	: N
NADJ	: CP → N → N	RAT	: N
CPADJ	: C → VP → CP	CHEESE	: N
CPREL	: C → S/NP → CP	SAW	: V
REL	: NP → V → S/NP	CHASED	: V
A	: Det	ATE	: V

The derivation of Figure 3.1 is then represented by an abstract term as shown in Figure 3.2.

Concerning the surface structure, we just take the string signature that contains as constants the vocabulary of the grammar. The lexicon that maps abstract terms to the sentence that they analyze is given by the homomorphism

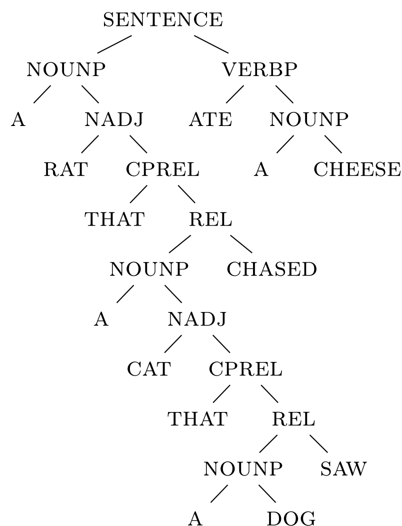


Figure 3.2: Abstract structure of the derivation of *a rat that a cat that a dog saw chased ate a cheese*

$\mathcal{L}_{syn}$ that maps each abstract type to the string type and the abstract constant to the following terms:

- $\mathcal{L}_{syn}(\text{SENTENCE}) = \mathcal{L}_{syn}(\text{VERBP}) = \mathcal{L}_{syn}(\text{NOUNP}) = \mathcal{L}_{syn}(\text{CPADJ}) = \mathcal{L}_{syn}(\text{CPREL}) = \mathcal{L}_{syn}(\text{NADJ}) = \mathcal{L}_{syn}(\text{REL}) = \lambda xyz.x(y\,z)$ (this λ-term, function composition, denotes the string concatenation as we have seen in the previous chapter)

- $\mathcal{L}_{syn}(\text{THAT})$ = that, $\mathcal{L}_{syn}(\text{A})$ = a, $\mathcal{L}_{syn}(\text{DOG})$ = dog, $\mathcal{L}_{syn}(\text{CAT})$ = cat, $\mathcal{L}_{syn}(\text{RAT})$ = rat, $\mathcal{L}_{syn}(\text{CHEESE})$ = cheese, $\mathcal{L}_{syn}(\text{SAW})$ = saw, $\mathcal{L}_{syn}(\text{ATE})$ = ate, $\mathcal{L}_{syn}(\text{CHASED})$ = chased.

We can now use another homomorphism so as to give the semantic interpretations of the sentences generated by the grammar. For this we take as object signature a higher-order signature in which we represent first-order logic formulae. Montague semantics is representing by means of formulae the set of logical models (usually called *worlds*) in which the sentence may hold true[1]. Montague semantics is based on a more complex logic, but for the purpose of our example, first order logic will be enough. The object signature will be built over two atomic types e, the type of entities, and t, the type of truth values. We will use the following logical connectives: $\exists : (e \to t) \to t$, $\wedge : t \to t \to t$; and the predicates (i.e. constants of type $e \to t$): **dog**, **cat**, **rat**, **cheese**; and the binary predicates (i.e. constants of type $e \to e \to t$) **saw**, **chased**, **ate**.

[1]Notice that with syntactic ambiguity, a sentence may be assigned different formulae accounting for the polysemy of certain sentences.

The syntax of λ-calculus is good to represent logical formulae; we will however follow the conventions of logic and use the standard syntax of logic using logical connectives as infix and quantifiers as binders.

The semantic homomorphism $\mathcal{L}_{sem}$ is mapping the abstract types to object types as follows: the meaning of sentences (terms of type S) are truth values and is thus of type t, so $\mathcal{L}_{sem}(\mathrm{S}) = t$; nouns (terms of type N) are interpreted as predicates, thus $\mathcal{L}_{sem}(\mathrm{N}) = e \to t$; noun phrases (terms of type NP) denote individuals and should thus be of type e, nevertheless, the proper treatment of quantification requires the use of a continuation passing style of programming (this amounts to see individuals as the set of their properties), so that $\mathcal{L}_{sem}(\mathrm{NP}) = (e \to t) \to t$. Now if we let np denote the type $(e \to t) \to t$, the other abstract types get intuitive interpretations: $\mathcal{L}_{sem}(\mathrm{V}) = \mathrm{np} \to \mathrm{np} \to t$, $\mathcal{L}_{sem}(\mathrm{VP}) = \mathrm{np} \to t$, $\mathcal{L}_{sem}(\mathrm{Det}) = (e \to t) \to \mathrm{np}$, $\mathcal{L}_{sem}(\mathrm{S/NP}) = \mathcal{L}_{sem}(\mathrm{VP}) = \mathrm{np} \to t$, $\mathcal{L}_{sem}(\mathrm{CP}) = e \to t$ and $\mathcal{L}_{sem}(\mathrm{C}) = e \to \mathrm{np}$.

We can now give the semantic interpretation of the abstract constants. The definition of $\mathcal{L}_{sem}$ is standard, we use the term $F = \lambda P\, O\, S.S(\lambda x.O(\lambda y.P\, x\, y))$ so as to shorten the notation of the interpretation of verbs.

SENTENCE	$: \mathrm{NP} \to \mathrm{VP} \to \mathrm{S}$	$= \lambda S\, P.P\, S$	THAT	$: \mathrm{C} = \lambda x\, P.P\, x$
VERBP	$: \mathrm{V} \to \mathrm{NP} \to \mathrm{VP}$	$= \lambda P\, O\, S.P\, O\, S$	DOG	$: \mathrm{N} = \mathbf{dog}$
NOUNP	$: \mathrm{Det} \to \mathrm{N} \to \mathrm{NP}$	$= \lambda D\, P\, Q.D\, P\, Q$	CAT	$: \mathrm{N} = \mathbf{cat}$
NADJ	$: \mathrm{CP} \to \mathrm{N} \to \mathrm{N}$	$= \lambda P\, Q\, x.P\, x \wedge Q\, x$	RAT	$: \mathrm{N} = \mathbf{rat}$
CPADJ	$: \mathrm{C} \to \mathrm{VP} \to \mathrm{CP}$	$= \lambda C\, P\, x.P(C\, x)$	CHEESE	$: \mathrm{N} = \mathbf{cheese}$
CPREL	$: \mathrm{C} \to \mathrm{S/NP} \to \mathrm{CP}$	$= \lambda C\, P\, x.P(C\, x)$	SAW	$: \mathrm{V} = F\, \mathbf{saw}$
REL	$: \mathrm{NP} \to V \to S/\mathrm{NP}$	$= \lambda N\, P\, S.P\, S\, N$	CHASED	$: \mathrm{V} = F\, \mathbf{chased}$
A	$: \mathrm{Det}$	$= \lambda PQ.\exists x.P\, x \wedge Q\, x$	ATE	$: \mathrm{V} = F\, \mathbf{ate}$

Then the sentence *a rat that a cat that a dog saw chased ate a cheese* is mapped to the following formula:

$$\exists x.\mathbf{rat}\, x \wedge (\exists y.\mathbf{cat}\, y \wedge (\exists z.\mathbf{dog}\, z \wedge \mathbf{saw}\, z\, y)) \wedge \mathbf{chased}\, y\, x \wedge (\exists u.\mathbf{cheese}\, u \wedge \mathbf{ate}\, x\, u)$$

This example illustrates on a simple grammar how ACGs work and in particular how they can represent standard constructions that model the interface between syntax and semantics in a unified framework. We can already notice that for the syntactic part, only a linear homomorphism is necessary and that for the semantics part we need a more complex one.

3.2 Expressiveness

When dealing with new kinds of formalisms, it is important to relate them with existing ones. The first thing to be done is to see whether usual formalisms can be represented in that new formalism. This is what has been done by de Groote and Pogodalla [104] who showed how to encode within $\mathcal{L}(2, k)$ formalisms with context-free derivations such as Context-Free Grammars (CFGs), Fischer's non-copying macro grammars and Multiple Context-Free Grammars (MCFGs). In [S22], we established that every grammar in $\mathcal{L}(l, k)$ –when $l > 3$–

could be represented by a grammar in $\mathcal{L}(3, k+l-3)$. We also proved in [S17] that every grammar in $\mathcal{L}(2, k)$ could be represented by a grammar in $\mathcal{L}(2, 4)$ and established that (linear) second order ACGs were not more expressive than MCFGs. In contrast, we do not know what class of languages is capture by $\mathcal{L}(3, 4)$ and from [S22] we know that this class is very complex as its Parikh image is more complex that the reachability sets of Petri nets.

In this section we sketch the proof that the hierarchies of (linear) of string and tree languages ACGs $\mathcal{L}(2, k)$ and $\mathcal{L}(3, k)$ collapse for $k = 4$. This shows that for strings and trees linear homomorphisms have a limited expressive power. It remains open whether this is also the case when dealing with languages of linear λ-terms. We then review some results that we established and which show that other grammatical formalisms can be encoded within ACGs. Those formalisms are various kinds of Lambek calculi and Stabler's formal account [272] of Chomsky's minimalist program [77]: minimalist grammars.

The collapse theorem

The proof in [S17] uses ideas that are comparable to a game interpretation (simple games restricted to the exponential free multiplicative fragment of linear logic) that was independently discovered by Lamarche [197] and Curien and detailed by Abramsky [33]. Actually this proof shows that deterministic tree transducers can normalize the image by a linear homomorphism of an abstract derivation of a second order ACG. The link with simple games becomes clearer when looking at the way deterministic tree walking transducers actually compute: they deterministically traverse the tree so as to find out the head symbol of the result. In other words they perform a sequential evaluation of the linear λ-term obtained by applying the homomorphism. As game models of λ-calculus were introduced so as to understand the nature of sequentiality, it is to be expected that an understanding of the sequentialization of normalization is based on a similar notion. With this method, we showed that the languages that were definable by second order ACGs were the output languages of deterministic tree walking transducers. Then using a result by Weir [294], it implies that every string language of a second order ACG is definable with an MCFG. Moreover, the construction given by de Groote and Pogodalla that encodes MCFGs into second order ACGs produces a an ACG in $\mathcal{L}(2, 4)$ implying that the class of string languages defined second order ACGs is captured by only using lexicons of complexity 4. It follows that every string ACG in $\mathcal{L}(2, k)$ can be converted into an ACG in $\mathcal{L}(2, 4)$ defining the same language. The situation is different for non-linear second order ACGs (or IO higher-order grammars) for which the growth of the language can be increased exponentially each time the order of the grammar or the complexity of the lexicon is increased. This gives rise to an infinite hierarchy of classes of languages.

After this result was obtained, we found a simpler proof which avoids the detour via simple games and gives a more satisfactory explanation of the result. We will give here the general lines of this proof. It also generalizes the result to tree languages which entails that the hierarchy of tree languages defined

by second order ACGs collapses also for lexicons of complexity 4. The main method goes through a proof-theoretic analysis of linear λ-terms. Given a closed linear λ-term M of type A that is built on a second order signature, one can decompose it into a linear λ-term M' and a substitution σ so that:

1. the free variables of M' are at most second order,
2. there is no subterm of M' of the form $xM_1 \dots M_n$ where x is an occurrence of a free variable of M' and one of the M_i has as head a free variable of M',
3. M' does not contain any occurrence of a constant,
4. for every variable x^A that is free in M', $\sigma(x^A)$ contains an occurrence of a constant.

For example if we were to take a term

$$M = \lambda fgxy.a(b(f(\lambda z.c(d\,x)z))y)(g(\lambda z_1 z_2.dz_1 z_2))$$

it would be decomposed into (M', σ) so that:

- $M' = \lambda fgxy.h_1(f(\lambda z.h_2\,x\,z)y)(g(\lambda z_1 z_2.h_3\,z_1\,z_2))$,
- $\sigma = [h_1 \leftarrow \lambda x_1 x_2.a(b\,x_1)x_2; h_2 \leftarrow \lambda x_1 x_2.c(d\,x_1)x_2; h_3 \leftarrow \lambda x_1 x_2.dx_1 x_2]$.

This decomposition of M is removing the *largest possible parts* of M that are only made with constants. This is the item 2 that expresses this maximality constraint. Then a simple analysis based on polarity counting shows that, for a fixed type A, there are finitely many possible terms M'. This comes from the fact that the positive atomic types of free variables in M' need to be matched against a negative occurrence of the same atomic type in A, while the negative occurrences of an atomic type in the type of free variables in M' need to be matched against a positive occurrence of the same atomic type in A. This has the consequence that the cumulated size of the types of the free variables of M' is bounded by the size of A finally showing that there are finitely many possible M'. Now, so as to construct an ACG in $\mathcal{L}(2,4)$ generating the same language, it suffices to use standard methods that make finite information flow in derivation trees. Here it suffices to build a new abstract signature whose types are pairs (A, M') where A is an atomic type of the original abstract signature and M' is term of type A the conditions given by the items $1, 2$ and 3. If M' contains $z_1, \dots, z_n$ as free variables, the image of abstract terms of type (A, M') by the lexicon will be terms of the form $\lambda k.kM_1 \dots M_n$ (which is the Church encoding of an tuple of n terms) so that the pair $(M', [z_1 \leftarrow M_1, \dots z_n \leftarrow M_n])$ is a decomposition of a term in the language of a term in the language described by the abstract type A in the original grammar. The main thing to remark is that the term $\lambda k.kM_1 \dots M_n$ has order 4, showing that the new ACG is in $\mathcal{L}(2,4)$. Such a construction shows the collapse of the hierarchy $\mathcal{L}(2,k)$ to $\mathcal{L}(2,4)$. Below 4 the hierarchy is strict $\mathcal{L}(2,1)$ corresponds to regular sets

of trees, $\mathcal{L}(2,2)$ corresponds to linear context-free sets of trees, $\mathcal{L}(2,3)$ has no corresponding definition in the literature but it strictly contains $\mathcal{L}(2,2)$, and $\mathcal{L}(2,4)$ corresponds to tree languages that are definable by hyperedge replacement grammars that were introduced by Courcelle [85].

This result has been extended further by Kanazawa [169] where he has showed that, seen as graphs, languages of λ-terms generated by second order ACGs could be represented by languages of hyperedge replacement grammars. It nevertheless remains open whether for languages of λ-terms, the hierarchy collapses. Kanazawa's result suggests that it could be the case. Though the construction we have outlined cannot be extended to the cases where the object language uses higher-order constants.

As we mentioned above the method we used in [S17] amounts to sequentialize the evaluation of abstract terms using simple games. The information that was added to the abstract types were *plays*, here, instead, the use of a linear λ-term amounts to enrich abstract types with *strategies*. This yields both a simple construction and a simple proof of its correctness. Now that we have made a connection with the denotational semantics of the linear λ-calculus, it becomes possible to extend the result to the hierarchy $\mathcal{L}(3,k)$. Similarly to the case of $\mathcal{L}(2,k)$, it is then possible to *reify* the values in the model directly in the types of the abstract signature. Thus the method yields that the hierarchy of string and tree languages definable with $\mathcal{L}(3,k)$ collapses for $k=4$.

Theorem 26 *For every tree or string ACG $\mathcal{G}$ we have:*

- *if $\mathcal{G}$ is in $\mathcal{L}(2,k)$, there is an ACG in $\mathcal{L}(2,4)$ that generates the same language as $\mathcal{G}$,*
- *if $\mathcal{G}$ is in $\mathcal{L}(l,k)$ with $l>2$, there is an ACG in $\mathcal{L}(3,4)$ that generates the same language as $\mathcal{G}$.*

Minimalist grammars and Lambek grammars

The inspiration that led to ACGs clearly takes its roots in Type Logical Grammars such as the calculi proposed by Lambek in the late 50's early 60's and that are now known as Lambek Grammars [199, 198] or Categorial Grammars. These grammars have been showed to define languages that are all context-free by Pentus [240] and Kandulski [172]. Nevertheless, it has always been assumed that the proof-theoretic nature of Lambek calculi and more precisely their capability of using hypothetical reasoning was making them able to describe syntactic structures with more subtleties than context-free grammars could. This is illustrated by the claim of van Benthem [53]:

> the move toward a more flexible [than Context-Free Grammars] Categorial Grammar has given us additionnal 'strong recognizing capacity', in terms of new constituent structures, while leaving 'weak capacity', concerning sets of flat strings recognized, at the old level.

This claim illustrates a sort of folklore knowledge in the community that Categorial Grammar has stronger descriptive capabilities than context-free grammars. We first proved that when considering non-associative Categorial Grammars, this was not the case [S13]. Indeed by encoding this formalism within the ACG framework we could prove that the syntax semantics relations that could be described with non-associative categorial grammars could also be represented with context-free grammars. With other techniques, we could prove a similar result for the original Lambek grammars [S10].

The issue here is that the notion of strong generative capacity is not precisely defined. Following Chomsky, the weak generative capacity of a formalism is the set of surface structures it generates. Its strong generative capacity represents the way the formalism ascribes syntactic structures to surface structures. In general, two grammars are considered strongly equivalent when they assign identical structures to surface structures. We think that this is much too strong a requirement and that it completely misses the point of what syntactic structures are. In general, when one proves that a grammar is *weakly equivalent* to another, the proof contains a way to map the derivations of one grammar to derivations of the other and vice versa. This mapping may be more or less complex, but without such a mapping, it is hard to make any proof of weak equivalence. We believe that the notion of strong equivalence should be parametrized by a class of possible mappings between derivations: rational equivalence, rational transductions, rational substitutions, macro-tree transductions, MSOL transductions etc... In this respect if we take a context-free grammar (with no unit rules and no ϵ-rule) and its Chomsky normal form, then we can consider them strongly equivalent as their derivations are rationally equivalent. If we take a context-free grammar G whose language does not contain the empty word, then a simple construction produces a grammar G' that generates the same language but with no ε-rules. This construction consists in removing ε-rules from G and for each rule add rules that are obtained by removing in the right-hand side some non-terminals that could derive ε in G. Should we consider G and G' strongly equivalent? Every derivation of G can be uniquely mapped (by a linear tree homomorphism) to an equivalent derivation (in the sense that it generates the same string) of G', while a derivation of G' may represent a possibly infinite set of derivations of G, but this set can be obtained by using a rational substitution (some symbol are replaced by a rational tree language) which inserts the regular tree languages that represent derivations of ε by adequate non-terminals. The relation between the derivations of G and G' is rather simple. It thus makes us believe that these two grammars share more similarities than differences. The current situation about strong and weak generative capacity is such that they are, by default, considered as essentially different.

Another example is that of Greibach normal form. It is generally considered that when putting a grammar G into Greibach normal form then the grammar G' that is produced is essentially different from G. In case G is in Chomsky normal form, then the derivations of the two grammars can be put in bijection by means of macro tree transductions. Here again we can question the idea

that G and G' have different structures. Somehow the need to use macro transductions instead of rational ones shows that the relation is more complex than in the case of Chomsky normal form.

So in a certain sense, we advocate in favor of a notion of strong generative capacity that emphasizes the notion of information: do the syntactic structures contain the same information? If so, how complex is the encoding and decoding of this information? Answering those questions show that, despite strong superficial differences, in the end weakly equivalent formalisms are rather similar. It also allows one to quantify how these superficial differences impact representations of syntactic structures. In the case of Lambek Grammars, their derivations can be obtained from those of the encoding context-free grammars using an MSOL transduction. The converse relation is more complex and it is not known if it belongs to an interesting class of transductions.

Another formalism that we have been able to represent within the framework of ACGs is the Stabler's derivational minimalism [272]. This formalization assumes a restriction called *the shortest move constraint* and we modeled within ACGs the formalization without this restriction. Our goal was two-fold, first reveal the structure of the derivations and second deduce some properties of the languages from this structure.

Derivational minimalism uses a feature checking mechanism that is resource sensitive. Moreover, with this feature mechanism, it formalizes the linguistic notion of *movement* that displaces constituants from right to left. The resource sensitivity and the directionality that movement was inducing made the community try to give logical representations of these grammars using variants of Lambek grammars (among others [205], [204], [202], [203], [40]). Contrary to the general intuition of the community, we proved [S18] that the derivations are best represented in terms of linear λ-terms, that is that the set of derivations can be faithfully represented by a set of closed linear λ-terms in a given signature. This shows that the impression of directionality was misleading. Moreover, it also shows that the class of languages that can be described by derivational minimalism without the shortest move constraint is particularly complicated as its Parikh images contain reachability sets of Petri Nets [S5].

3.3 Parsing Algorithms

Grammatical formalisms have a natural algorithmic problem called parsing. This problem consists in mapping a sentence to a representation of the set of its possible derivations. We have given [S22] an algorithmic solution for second order linear ACGs. As a first generalization, we proved that this algorithm can be extended to non-linear second-order ACGs [S21] using intersection types. This generalization shows that it is possible to generate texts from semantic representations that may be logical formulae. Of course, these formulae are not taken up to logical equivalence, but they can nevertheless serve as a high-level representation of meaning and some basic equivalence relations could be added such as the associativity and the commutativity of the conjunction and of the

disjunction.

One of our motivations to introduce the notion of recognizability for simply typed λ-calculus was to simplify the proof of this result. Indeed our proof looked quite similar to the usual proof of closure of context-free languages under intersection with regular sets [296]. Using the closure of recognizable languages of λ-terms under inverse homomorphism and the fact that singleton languages are recognizable (by Statman Theorem), we know that the set of syntactic structures of a given λ-terms in a second order ACG is a recognizable set of trees. Moreover as all the theorems that are used are effective, we thus have a parsing algorithm for ACGs.

When we look at the grammar that we used as an example of ACG in Section 3.1, this means that we may retrieve algorithmically the set of derivations whose interpretation is a given logical formula, as the one we have taken as example:

$$\exists x.\mathbf{rat}\, x \wedge (\exists y.\mathbf{cat}\, y \wedge (\exists z.\mathbf{dog}\, z \wedge \mathbf{saw}\, z\, y)) \wedge \mathbf{chased}\, y\, x \wedge (\exists u.\mathbf{cheese}\, u \wedge \mathbf{ate}\, x\, u)$$

This result is far from being intuitive as the operations that are performed by λ-calculus are complex. Nevertheless, the conceptual gain of recognizability makes the proof rather trivial. It also generalizes the remark of Mezei and Wright [213] about the regularity of the set of derivations of a sentence in a context-free grammar.

When we look at the algorithm this method gives, it amounts to compute least fixpoints in the domains of interpretation of atomic types. When instantiated on a context-free grammar, this naive algorithm is a bottom-up algorithm that does not beneficiate from the binarization procedure that accelerates the Cocke, Younger [297] and Kasami [173] algorithm. Binarization methods can be adapted, by transforming the abstract language, but, in general, the parsing problem of second order non-linear ACGs is non-elementary. If we fix the complexity of the lexicon at k, this problem has a tower of exponential of height $k-1$ [117] as time complexity.

An important feature of this algorithm is that denotational semantics is providing the representation of the information that is necessary to represent the set of derivation trees. This is in general the difficult part when dealing with parsing. This information may be rather complicated, for example in parsing algorithms for Tree Adjoining Grammars [261, 228] where it is represented with dotted trees with indices. Then proving that this information is indeed sufficient to deduce the existence of a syntactic structure requires most of the effort in proving the algorithm correct. Here this part is already contained in the fact that we use models of λ-calculus which ensures the correctness of the algorithm as a corollary. So technically, the use of denotational semantics seems to be a conceptual improvement.

The complexity of the parsing problem for non-linear ACGs pushed us to study some restrictions. The algorithm we proposed for linear second order ACGs is running in polynomial time. This algorithm has been recast by Kanazawa in terms of datalog program [168]. In this article, Kanazawa also

remarks that the result can be extended to second order *almost linear* ACGs. Such ACGs use lexicons which map constants to almost linear terms which obey the non-copying constraint for all variable of functional type but not necessarily for variables of atomic type.

Kanazawa's datalog method is very interesting, not only does it allow one to give a nice presentation of parsing algorithms for second order ACGs, but also it allows one to define parsing algorithms for other formalisms. The view datalog gives of parsing algorithms is that they are mostly specific strategies for computing fixpoints. In particular, Kanazawa has showed that many of the algorithms that were described in the literature in a rather technical way for particular formalisms could be described and generalized in terms of datalog program transformation [167]. This presentation provides simpler presentations of algorithms and also simpler proofs of their correctness. Interestingly, the community in datalog has tried to reduce every fixpoint computation strategy to a unique one called the semi-naive bottom-up algorithm. For this they have developed a wide range of program transformations which preserve program semantics. An important transformation is *magic supplementary set rewriting* that allows to reduce the top-down resolution algorithm to the semi-naive one [46]. On a datalog program that represents a context-free grammar, this transformation gives rise to an improved version of Earley's parsing algorithm [109]. The algorithm is improved in the sense that the *magic* predicate make the algorithm have a time complexity that is linear with respect to the size of the original grammar instead of being quadratic. The datalog methodology allows us to see parsing algorithms as program transformations and program optimizations. From a software engineering perspective this view of parsing allows to factor out the semi-naive bottom-up resolution algorithm which is responsible of the memoisation which is delicate to implement and may constitute a serious bottleneck in practice.

We may understand grammars as non-deterministic programs that with least fixpoints. Using datalog may seem as just another way of computing those fixpoints. Nevertheless, datalog offers richer computation capabilities, in other words, datalog is intentionally richer than grammars. And thus it allows us to define parsing algorithms that could not directly be described in terms of grammars. The magic predicate in magic supplementary set rewritting is an instance of this phenomenon. Another good example is Kanazawa's prefix-correct algorithm[2] for MCFGs which uses a program that cannot be represented as a grammar [167]

With my PhD student Pierre Bourreau, we have worked on generalizing Kanazawa's datalog approach to almost affine ACGs [S3, S2]. This work required to use game semantics as a way to prove the correctness of the algorithm. We have also extended this approach to copying formalisms like PCMFG [S1]. In this work we describe various transformation that allows us to obtain algo-

[2] A parsing algorithm has the prefix-correctness property when it reads the input from left to right and rejects incorrect sentences as soon as it has processed a prefix that cannot be completed into an element of the language.

rithms with or without the prefix-correctness property, and which may also use what is known as the left-corner strategy [227]. Here we take advantage of various program transformations. The way those transformations are combined results in different algorithms that may or may not have the prefix-correct property, that may use or may not use a left corner strategy etc...

3.4 OI grammars

Before ACGs were introduced, higher-order had been already studied in the context of formal language theory. The first moves toward higher-order languages has been done around the same time by Aho and Fischer. Aho introduced indexed grammars [38] and an equivalent abstract machine nested stack automata [39] (that would be called nowadays second order pushdown automata). His idea consisted in using context-free grammars where non-terminals are parametrized with a stack and where rules are allowed to push or pop symbols in or out of the stack. Fischer introduced the notion of IO and OI grammars [126]. These grammars can be seen as extensions of context-free grammars where non-terminals may have string parameters that they can use in the right-hand side of the production. The order in which the evaluation is made: the parameters are evaluated first (the Inside-Out –IO– evaluation); or the encompassing non-terminals are evaluated first (the Outside-In –OI– evaluation); is what distinguishes IO from OI. This distinction is nowadays best coined by call-by-values (IO) and call-by-names (OI). As a matter of fact, Fischer [126] has established that OI languages coincided with indexed languages.

What Aho and Fischer did was starting a move towards higher-order. This move was completed later on by Maslov [212, 211] for indexed grammars and higher-order pushdown automata. Concerning Fischer's macro grammars, their extensions were proposed slightly later. Their connection with algebras, exposed in [120, 121], opened the way to Damm for a definition of higher-order macro grammars: the IO and OI hierarchies [101]. Thereafter, Damm generalized Fischer's result showing that Maslov's higher-order pushdown automata were defining the same class of languages as higher-order OI grammars.

While Damm's definition of higher-order OI grammars was based on simply typed λ-calculus, he did not study full fledged OI grammars. Indeed, as it is often the case in such studies, Damm tried to work on grammars in a certain normal form. In the course of proving that these normal forms were defining the same class of languages as unrestricted grammars, Damm made a mistake related to the handling of α-conversion. Afterwards, the connection between higher-order pushdown automata and higher-order recursive schemes has been re-explored by [182]. The work of [218] makes it clear that Damm has defined a class of languages that does not use the full power of simply typed λ-calculus but rather a restriction that is known as *safe* λ-calculus (see Section 2.3). It is not known whether safe OI grammars define the same class of languages as unsafe ones. When restricting to order 2 grammars however safe and unsafe languages coincide [37]. There is also some partial result that has been

established by Parys [239] that shows the safety constraint higher-order safe *deterministic* grammars are less expressive that higher-order deterministic ones. Here the notion of determinism comes from the automata theoretic definitions of those grammars. Nevertheless, the deterministic unsafe language that Parys proves to be not a deterministic safe one can be defined by a non-deterministic higher-order pushdown automaton.

This recent view on Damm's results shows that a study of unsafe higher-order grammars was yet to be done. We started this study in [S12] and we report here its main results.

Second order ACGs when using non-linear lexicons correspond to unsafe higher-order IO grammars when they are used to define languages of strings or languages of trees. As they may also define languages of λ-terms, they are slightly more general. This possibility is important as it makes it possible to represent Montague's truth-conditional semantics. One of the motivation of OI grammars when Fischer introduced them was the syntactic modeling of programming languages. When designing the syntax of a programming language, OI grammars allow one to enforce at the level of the syntax that variable names can be used only when they have been previously declared. At the level of Montague semantics this feature may be used so as to approximate anaphora resolution by means of non-deterministic choice. This is in particular relevant when one considers the dynamic extension of Montague semantics proposed by de Groote [145, 147]. Moreover the capabilities of simply typed λ-calculus to manipulate finite models [152] gives the possibility to OI grammars of implementing rather advanced approximated anaphora resolution mechanisms, by choosing among the discourse referent that satisfy a certain set of properties expressed as a finite set of features.

It is possible to follow Damm's original definition of OI grammars, by simply taking a usual equational definition. But, as the method we use to study those grammars is semantics, it is simpler to consider $\lambda Y{+}\Omega$-calculus. This is made possible by means of Bekić's identity [50]. Given a $\lambda Y{+}\Omega$-term M, it defines a language as follows:

$$\mathit{lang}(M) = \{N \mid M \xrightarrow{*} N \text{ and } N \text{ is a } \lambda\text{-term in normal form}\}$$

We now confuse the notion of higher-order OI languages and the languages that are definable by means of $\lambda Y{+}\Omega$-terms. As an example of the use of Bekić's identity so as to transform a rule based description of a grammar into a $\lambda Y{+}\Omega$-terms based presentation, let us consider the following regular grammar:

$$\begin{aligned}
A &\to a\,(b\,A\,B)\,A\\
A &\to a\,A(b\,B\,A)\\
B &\to b\,(a\,A\,B)\,A\\
B &\to b\,A(a\,B\,A)
\end{aligned}$$

Abstracting over the A, we may see B as generating the same language as the λY+-term $M = \lambda A.Y(\lambda B.b\,(a\,A\,B)\,A + b\,A(a\,B\,A))$. Now we can see that A generates the same language as the term $Y(\lambda A.a\,(b\,(M\,A)\,B)\,A{+}a\,A(b\,(M\,A)\,A))$. This procedure amounts to perform a sort of Gaussian elimination of non-terminals.

We call order of a term M the maximal order a type A such that Y^A has an occurrence in M. A language is said to be an order n language when it is definable by an order n term.

Working directly with λ-calculus allows us to make several interesting remarks. First of all, the remark we made earlier about the way OI grammars can handle variables allows us to define sets of -terms of a given type by means of a grammar.

Theorem 27 *For every type A, there is a $\lambda Y{+}\Omega$-term M so that $lang(M) = \Lambda^A$.*

As there are some λ-terms that are not safe this theorem implies that, when considering languages of λ-terms, unsafe grammars are strictly more expressive. Of course, this could be showed in a simpler manner by simply considering a singleton languages made of an unsafe term. But this would be a slightly artificial way of separating safe and unsafe grammars. This theorem shows that one can define a language that contains infinitely many unsafe terms and that cannot directly be represented in the language of a safe grammar. It also shows that there are OI languages that are not IO languages. Indeed, IO languages are generated by finitely many -terms while it is not the case of Λ^A.

Given a full model $\mathbb{M}$ of λ-calculus, Loader's result [207] that the λ-definability problem is undecidable implies that the set of semantic values in $\mathbf{M}$ taken by the terms in an OI languages is not recursive.

Theorem 28 *Given a $\lambda Y{+}\Omega$-term M of type A, a finite model $\mathbb{M}$, the problem whether an element f of $\mathbb{M}$ of type A is in the set $\{[\![N]\!]_{\mathbb{M}} \mid N \in lang(M)\}$ is undecidable.*

This last result shows an important difference between OI grammars and IO ones for which the set of semantic values in a finite model taken by terms in a language can be enumerated. Moreover most of the results about IO grammars follow from the fact that this enumeration is algorithmically possible. This difference between IO and OI suggests that OI should be more expressive than IO. This is actually true, and using a standard CPS translation one can turn an IO grammar into a $\lambda Y{+}\Omega$-term whose language is the same as the original grammar.

Theorem 29 *For every non-linear second order ACG $\mathcal{G}$, there is a $\lambda Y{+}\Omega$-term M so that $lang(M) = \mathcal{O}(\mathcal{G})$.*

There are some comments to be made about the translation. First of all, a CPS translation does not preserve safety so that it remains open how the

class of languages definable in the safe IO hierarchy compares with the class of languages definable in the safe OI hierarchy. Second of all, the CPS translation make an order n IO grammar into an order $n+2$ OI grammar and we do not know how the two hierarchies compare for a fixed order. Fischer [126] has proved that IO and OI macro languages are incomparable; showing that order 2 languages form incomparable classes. Accordingly, we may conjecture that this remains true at every order. There is however no obvious way of generalizing Fischer's proof.

We are now going to see how we can decide simple properties of given OI languages. For this we adopt again a method based on denotational semantics. The main difficulty we face is to understand how to tame non-determinism within a finite model. This is achieved by working with monotone models. In these models, the non-deterministic operator is interpreted as join, so as to be able to capture certain semantic properties of terms in the language we will consider particular elements in the models: *join-prime elements*.

Definition 30 In a lattice $\mathcal{L}$, an element a is said to be join-prime when for every b and c, if $a \leq b \vee c$ then either $a \leq b$ or $a \leq c$.

We now need to make this definition suitable for higher-order languages and we introduce the notion of *hereditary join-prime elements* of a monotone model.

Definition 31 Given a monotone applicative structure $\mathcal{M} = (\mathcal{M}_A)_{A \in \mathcal{T}(\Sigma)}$, for every type A we define the sets $\mathcal{M}^+_A$ and $\mathcal{M}^-_A$ by:

1. $\mathcal{M}^+_0$ and $\mathcal{M}^-_0$ contain the prime elements of $\mathcal{M}_0$ that are different from $\bot_0$,
2. $\mathcal{M}^+_{A \to B} = \{(\bigvee F) \mapsto g \mid F \subseteq \mathcal{M}^-_A \wedge g \in \mathcal{M}^+_B\}$,
3. $\mathcal{M}^-_{A \to B} = \{f \mapsto g \mid f \in \mathcal{M}^+_A \wedge g \in \mathcal{M}^-_B\}$.

A valuation ν on $\mathcal{M}$ is said *hereditary prime* when, for every variable x^A, $\nu(x^A) = \bigvee F$ for some $F \subseteq \mathcal{M}^-_A$. The elements of $\mathcal{M}^+_A$ are called the *hereditary prime elements* of $\mathcal{M}_A$.[3]

A LFP model is said *hereditary prime* when each constant c of Σ is interpreted as $\bigvee F$ for some F included in $\mathcal{M}^-_{\tau(c)}$.

Hereditary prime models allow us to observe certain semantic properties of the λ-terms that are in the language of a $\lambda Y+$-term. This is stated by the Observability Theorem.

Theorem 32 (Observability) *Given a $\lambda Y{+}\Omega$-term M of type A, an hereditary prime LFP model $\mathbb{M} = (\mathcal{M}, [\![\cdot,\cdot]\!])$, an hereditary prime valuation ν and an hereditary prime element f of $\mathcal{M}_A$, we have the equivalence:*

$$f \leq [\![M,\nu]\!] \Leftrightarrow \exists N \in \mathcal{L}_{OI}(M). f \leq [\![N,\nu]\!]$$

[3] As is usual, we assume that, when $F \subseteq \mathcal{M}^+_A$ is such that $F = \emptyset$, then $\bigvee F = \bot_A$.

This theorem is rather easy to prove using standard finite approximation techniques.

A simple consequence of this theorem is that we may decide the emptiness of OI languages. For this it suffices to take the hereditary prime LFP model generated by $\mathbb{2}$ and so that constants are interpreted as the following elements of the corresponding types:

- $\mathbf{e}_o = \top$,
- $\mathbf{e}_{A\to B} = \mathbf{e}_A \mapsto \mathbf{e}_B$.

If we consider a valuation ν that maps a variable x^A to the corresponding element $\mathbf{e}_A$, then for every λ-term N, we have $[\![N^B, \nu]\!] \geq \mathbf{e}_B$. As a consequence, of this fact and of the Observability Theorem, we have:

Proposition 1 *Given a $\lambda Y{+}\Omega$-term M of type A, and a valuation that maps any variable x^B to $\mathbf{e}_B$, we have that*

$$\mathcal{L}_{OI}(M) \neq \emptyset \Leftrightarrow \mathbf{e}_A \leq [\![M, \nu]\!]$$

This implies that the emptiness problem for OI languages is decidable.

Now we wish to use the same technique so as to prove that the membership problem is decidable for OI languages. For this we refine Statman's construction for finite completeness and show that for every λ-term N, there is a hereditary prime LFP model that characterizes N.

Lemma 33 For every λ-term N, there is a hereditary prime LFP model $(\mathcal{M}, [\![\cdot,\cdot]\!])$, a hereditary prime valuation ν and an hereditary prime element f of the model that for every λ-term P we have:

$$[\![P, \nu]\!] \geq f \Leftrightarrow P =_{\beta\eta} N$$

As a consequence, using again the Observability Theorem, we obtain that the membership problem is decidable for OI languages. Moreover we obtain a generalization of Statman finite completeness result.

Theorem 34 *Given two $\lambda Y{+}\Omega$-terms of the same type M_1 and M_2, we have the following equivalence:*

- $\mathcal{L}_{OI}(M_1) = \mathcal{L}_{OI}(M_2)$,
- *for every LFP model $(\mathcal{M}, [\![\cdot,\cdot]\!])$ and every valuation ν, $[\![M_1, \nu]\!] = [\![M_2, \nu]\!]$.*

3.5 Conclusion and perspectives

In this chapter, we have seen how simply typed λ-calculus could serve in the definition of grammatical formalisms. Using λ-calculus has several advantages: a conceptual simplicity and the ability to use a rich set of tools. In particular, we have mentioned at several places how denotational semantics was giving elegant solutions to seemingly difficult problems. The collapse theorem and the intuitions provided by simple games are an example of this. But parsing algorithms form a more striking application of the conceptual efficiency of denotational semantics. For higher-order OI grammars, most of the formal work that needs to be carried out concerns the notion of hereditary primality which is a semantic notion. This study of a problem in formal language theory with denotational semantics has in return given us a class of functions, hereditary prime functions, for which the definability problem is decidable. It would be nice to understand which kinds of properties of programs they can express and see also whether we can decide definability for joins of such functions.

We have mentioned the use of datalog programs for describing parsing algorithms for almost affine second order ACGs and also to obtain several parsing algorithms for PMCFGs and in particular algorithms with the prefix-correct property. The collapse theorem allows us to transform second order ACGs into MCFGs and thus to construct prefix-correct parsers for them. Nevertheless, this requires heavy transformations. Somehow the prefix-correct property is related to the sequential execution of the grammar. It seems that simple games can provide a nice way of representing the invariants that need to be satisfied in order to execute the grammar sequentially. Another benefit is that it would then become possible to generalize the notion of prefix-correctness to trees and to linear λ-terms. Lamarche's exponential [197] that makes simple games be a model of linear logic which decomposes Berry and Curien's [54] sequential algorithms could then allow us to generalize prefix-correct algorithms for second-order non-linear ACGs. Here again, results in denotational semantics are paving the way to describe those algorithms. The underpinning ideas are non-trivial and may find here a natural application. Bucciarelli and Ehrhard's notion of strongly stable functions [68] that are an extensional representation of sequential algorithm [111] may serve to increase the parallelism of parsing by allowing the parallel resolution of independent subgoals. This approach is also simplified by Ehrhard's linear decomposition [110] of strong stability which may allow us to start with linear or affine second order ACGs and then use the exponential construction in order to extend the approach to the non-linear case.

The interest of datalog seems to be limited to almost affine second order ACGS. But an important feature that we mentioned is that datalog has greater capacities of describing fixpoint algorithms than grammars. So an important question consists in extending these capabilities to the non-linear case so as to be able to describe with the highest simplicity strategies for fixpoint computation. Clearly, the link with linear logic and in particular the exponential modality can reveal quite helpful in this line of work.

One of the motivations for studying OI grammars came from the idea of representing linguistic theories from the perspective of transduction or rather as compositions of transductions. The result of the decision for non-linear second order ACGs gives the decidability for a particular class of transductions and of their compositions. This class of transducers can be seen as a generalization of Nivat's account of rational transductions in terms of bi-morphisms [231]. We can describe a transduction with a pair of second order ACGs that share the same abstract signature $\mathcal{G}_1 = (\Sigma, \Sigma_1, \mathcal{L}_1, S)$ and $\mathcal{G}_2 = (\Sigma, \Sigma_2, \mathcal{L}_2, S)$ and so that Σ_1 is a tree signature and $\mathcal{L}_1$ is a relabeling homomorphism. The signature Σ models the runs of a finite state automaton on trees built on Σ_1 that are being translated and $\mathcal{L}_2$ represents the operations that are performed during the translation. Now if Σ_2 is a tree signature and $\mathcal{L}_2(S)$ is an atomic type, then we can compose this first transduction with another one and this can be done in several steps. The closure under inverse homomorphic image of recognizable sets of terms and the closure of recognizable sets of trees under relabeling gives us the possibility to effectively compute the regular set of trees that are the inverse images by a cascade of transductions.

Models of natural languages are often described with cascades of transductions, in particular in automatic translation, specialized transducers are composed in larger ones. These models rely on the closure under composition of the class of transducers they use so as to obtain efficient machines. The kinds of transductions we have just seen are not closed under composition, it thus seems hard to optimize their composition. The problem here is the articulation between non-determinism and copying. In the call-by-value strategy that is adopted by the kinds of transductions we have considered, the non-determinism needs to be resolved before copying. In a call-by-name strategy, the non-determinism is resolved when it is met and thus copied terms may yield different results. When we compose two transductions, we need to intertwine the two mechanisms. Indeed, parts of the output terms that are coming from the copy of a given term may then be treated independently in the subsequent run of the second transduction. The phenomenon has already been illustrated by Engelfriet in his proof of the incomparability of top-down and bottom-up tree transductions [116] and the introduction of IO and OI transductions [123].

So as to do this, a good way is to augment the $\lambda Y+$-calculus with a *let* construction that forces the evaluation of some term. Nevertheless so as to have nice operational semantics it would be better to make this construction be evaluated lazily. This would change the semantic of each individual transduction, but should not modify the class of definable relations. In this setting, we may hope that the analysis of transduction may yield to a closure under composition, but also that program optimization techniques can help in obtaining efficient transducers. Of course, with this control over evaluation, it becomes easy to construct transductions that define undecidable relations by encoding the definability problem. It would then be important to understand how the class of transductions that can be proved decidable with the aforementioned technique can be represented and manipulated in this setting. Another important restriction could be to work under the safety restriction as relations

might be decidable in this setting. A pre-requisite before starting this study would be to define a denotational semantics for these programs that would encompass the syntactic semantics and thus help us in finding interesting classes of transductions.

But before doing so, there is a more pressing problem which is the study of a syntactic semantics for OI languages. For IO languages, the syntactic semantic is rather simple, it is basically the languages themselves. Indeed, if L_1 is an IO language made of terms of type $A \to B$ and L_2 is an IO language made of terms of type A, then the language $L_1 \bullet L_2 = \{M_1 M_2 \mid M_1 \in L_1 \wedge M_2 \in L_2\}$ is an IO language. This is not true for OI languages. Consider the terms $M_1 = \lambda fxy.f(x+y)$ and $M_2 = \lambda fxy.(f\,x) + (f\,y)$, they define the same language $lang(M_1) = lang(M_2) = \{\lambda fxy.f\,x; \lambda fxy.f\,y\}$. But when we consider $P_1 = M_1(\lambda x.a\,x\,x)e_1\,e_2$ and $P_2 = M_2(\lambda x.a\,x\,x)e_1\,e_2$, then $lang(P_1) \neq lang(P_2)$ as:

- $lang(P_1) = \{a\,e_1\,e_1; a\,e_1\,e_2; a\,e_2\,e_1; a\,e_2\,e_2\}$,
- $lang(P_2) = \{a\,e_1\,e_1; a\,e_2\,e_2\}$.

This shows that, contrary to IO, the language a term defines does not characterize its behavior when applied to other terms. The difficulty is that a $\lambda Y+$-term both denotes a language and an operation on language. It seems thus necessary to interpret terms in a domain that can both represent languages and language operations. A natural choice has already been proposed in the literature in terms of relational semantics [69]. We should see in this setting how safety can be interpreted. This is important as for OI safe languages, we can prove that we can enumerate the sets of semantic values taken by the terms in the language. This would allow us to understand better how the relational models may connect to decision procedures. Then, we should try to make IO and OI semantics work together in that domain so as to be able to interpret and study higher-order transductions.

In this chapter we have also quickly glossed over Montague semantics. In formal semantics of natural language, we can witness an increasing complexity in the constructions that are proposed. It becomes quite difficult to verify whether the intended meaning of a sentence is actually computed correctly by the interpretations proposed in the literature. The system CoQ [281] offers most of the tools and automations so as to develop fragments of formal semantics. Ranta pioneered the use of type theory in the modelization of natural language in the 90's [254]. It is now gaining momentum with the work of Bekki [51, 52] and of Luo [73]. With Kobele, we could experience during a course at the summer school NASSLLI in 2014 that the use of CoQ was making concrete the constructions of Montague semantics and helped the student to understand them simply as it was giving them the opportunity to manipulate and experiment with them. In the long run, we would like to develop tools that are able to encompass syntax and semantics within CoQ so as to help the developments of formal semantics. This could serve both for teaching and research.

Chapter 4

Mildly Context Sensitive Languages

In natural language modeling, formal methods have played a central role. This trend has probably started under the impulse of Chomsky with his book *Syntactic Structures* [76]. Not only Chomsky's work has had a strong influence in linguistics, but also in the design of compilers. His introduction of context-free grammars has triggered huge developments in formal language theory, compiler design, verification of recursive programs etc... The interest of a formal approach to natural language is that it gives objective views about phenomena and models. It brings clarifications of notions and also points to where the problems are. As such, the methodology is a clear benefit. A good example of the outcome of the application of formal methods to a formerly informal field is described by Barbara Partee [238] in the context of Montague semantics:

> Before Montague, semanticists focused on the explication of ambiguity, anomaly, and "semantic relatedness"; data were often subjective and controversial. The introduction of truth-conditions and entailment relations as core data profoundly affected the adequacy criteria for semantics, and led to a great expansion of semantic research.

Another outcome of formal/mathematical methods is the idealization of the subject it studies. In the study of syntax, it leads rapidly to posit that natural language potentially contains infinitely many utterances even though, in practice, only a finite number of utterances can ever be recorded. In order to explain this, Chomsky distinguishes competence (the linguistic knowledge of native speakers) and performance (the produced sentences). This distinction allows him to advocate in favor of generalizations and thus to the possibility of sentences of unbounded length. In practice, this justifies the linguistic adequacy of grammars that describe infinite languages. These grammars generalize linguistic data so as to get the simplest descriptions. For example, there is *a priori* no bound on the number of adjectives that may be used to modify a

noun; thus it seems natural to build a grammar which may recognize sentence where an arbitrary number of adjectives may modify a noun. Not only would fixing a bound seem arbitrary, but also, it would make the grammar more complex as it would be necessary to incorporate a way of verifying whether this bound is reached or not. Somehow the distinction between competence and performance plays the role of the Ockham razor in linguistic descriptions. Another example of Chomsky is how he proves that natural languages cannot be modeled with regular languages using self-embedded (as in the toy grammar of the previous chapter) sentences which exhibit a pattern of the form $a^n b^n$ which is not regular. We can make a similar parallel with computers. The study of computation is made simpler by considering unbounded amount of resources while each computer has limited resources and is thus best modeled by a finite state machine. But, not considering potential infinity when we model computers with programming languages leads to much less elegant programs, more difficult to write and also specialized to a given architecture.

When considering language knowledge as represented in the brain by a specialized area, and the fact that babies are able to learn any language they are exposed to as their mother tongue, it becomes plausible that the class of human languages is biologically determined. This hypothesis is further strengthened by the argument of the poverty of stimulus which posits that the utterances to which children are exposed are too scarce so as to determine a language in an arbitrary class. Chomsky calls this class of grammar, the *Universal Grammar* [74]. This expression somehow means the grammar with which the grammars of human languages are written. As Kimbal [174] explains, the universal grammar can formally be understood as determining the class of formal languages in which human language can be modeled:

> The (Chomsky hierarchy) represents the fact that regular languages are the simplest or least complex, CF [context-free] languages are next, and CS [context-sensitive] are the most complex of the phrase structure languages. In a certain sense, the problem faced in the construction of a theory of universal grammar is to determine exactly how 'complex' natural language is.

Nevertheless, as Chomsky remarked that context-free grammars do not convey (at least explicitly) the linguistic knowledge that native speakers have about language such as the similarity of the structures of passive sentences and active ones, he started to work (and was subsequently widely followed by other linguists) to describe many linguistic phenomena by means of transformation from canonical sentences [74]. Studying transformational grammars from the point of view of computational power, Peters and Ritchie [241] showed that they could model every recursively enumerable language. This result showed that transformational grammars failed to delimit precisely enough the class of natural languages. This result also asks questions about the validity of the claims of linguists working with transformational grammars whether certain sentences are accepted or not by their grammars. These claims were never

formally proved and mostly relied the linguists' intuitions while Peters and Ritchie showed that there is no systematic method to verify those claims. Then the problem of using a linguistic metatheory with a limited generative power has pushed Gazdar and his co-authors to propose Generalized Phrase Structure Grammars (GPSG) [131, 130] which can only define the same class of languages as context-free grammars, but in which most (if not all) of the linguistic phenomena described by means of transformations could be described with simplicity. This work lead to the natural question of whether the class of context-free languages is rich enough to contain any natural language. This was recalled by Pullum and Gazdar [245]: "Whether non-context-free characteristics can be found in the stringset of some natural language remains an open question, just as it was a quarter century ago.". Finally, examples of cross-serial dependencies in Dutch [156] and in Swiss-German [268] showed that there were linguistic phenomena that were exhibiting the pattern $a^n b^p c^n d^p$ (in principle for any n and p) and thus were not context-free.

As the phenomenon of cross-serial dependencies is captured by a class of grammars proposed by Joshi et al. [162], tree adjoining grammars (TAG), this led Joshi to formalize a notion of *mildly context sensitive languages* which should capture the class of natural languages. This formalization allowed the community to identify a class of languages that is described by many seemingly different formalisms. In particular, the collapse result that we have proved [S17], shows this is the class of string languages that are definable with second order ACGs. This has led us to investigate further this class, but also to think about the general problem of the class of languages that capture human languages. In this chapter, we use Multiple Context-Free Grammars (MCFGs) so as to describe this class of languages.

In this chapter, we present our work around the notion of mildly context sensitive languages. First of all we present the notion itself as it was defined by Joshi. As the notion is semi-formal and sometimes a bit imprecise, we try to give a completely formal definition. Secondly, we will explain how, unexpectedly, we could prove that a language, called MIX, falls in the class of Multiple Context-Free Languages (MCFL), the class defined by MCFGs. Thirdly, we show that contrary to what was claimed in the literature, MCFLs are in general not iterable in any reasonable sense. Finally, we present a classification of the class of languages that have been proposed as representing mildly context-sensitive languages.

4.1 On mild-context sensitivity and its limitations

The proof of the collapse theorem for second order linear ACGs is a further illustration of an interesting phenomena that has been already remarked by Joshi and his students: the convergence of mildly context sensitive formalisms [163]. When Joshi started his work on Tree Adjoining Grammars (TAG) [41, 160], he tried to assess whether they were adequate from a linguistic point of view. For this he defined the notion of Mildly Context Sensitive Languages [161]. Joshi

gives a set of semi-formal properties that classes of languages whose ambition is to capture human languages should meet. These properties are as follow:

- the class should contain context free languages,
- each language in the class should exhibit limited crossing dependencies,
- each language in the class should have the constant growth property,
- each language in the class should have polynomial parsing problem.

Three of these properties are formally clear, the property of limited crossing dependencies is a less precise. Regarding these properties, it has been rather quickly agreed that MCFLs were the class of languages that was the best fitting those criteria. This agreement is questionable and the results we prove in the following sections show that another class seems to be more adequately fit Joshi's intuition. This class is called well-nested Multiple Context-Free Languages (MCFL_{wn}). Another problem is to make these properties slightly more precise. Finally, the literature has pointed at some phenomena which suggest that those properties may be too restrictive. In this section, we revisit these properties in the light of some results that were not known at the time Joshi published his work and we propose our own variation around Joshi's notion of mild context sensitivity. We then propose some methods so as to accommodate in this framework the linguistic phenomena that seem to escape the present definition of mild context sensitivity. This for us the opportunity to quickly report on some technical work and some linguistic modelization that we made in this setting.

First of all, we need to remark that the goal of the notion of mildly context sensitive languages is not to characterize exactly the class of natural languages but rather to give an over-approximation of that class. It is clear that the class of regular languages contains languages which cannot have any relation with human languages. Lets for example mention one letter languages that represent sets of numbers satisfying a Presburger formula. Similarly the class of context free languages also contains languages that are too complex to represent natural languages. Thus, the requirement that the class should contain all CFLs is somewhat too strong, but, with the goal of building an over-approximation, this seems a mild problem. From the point of view of formal language theory, it seems more natural to work with well defined classes of language that have nice closure properties such as closure under union, closure under rational transductions etc... and it thus seems unnatural to carve inside CFLs the smallest part that would be sufficient to model natural language while we are trying to give an upper bound on the complexity of natural language that requires some extra expressive power. Here the approximation is mostly concerned with syntactic constructions, being more precise would require to take into account some more properties of natural languages as for example its learning properties.

Let us now turn to the property of the constant growth property. Joshi, in his article mentions that this property is related to the semilinearity property. It seems that this is what he had in mind, but that he used the constant

growth property because it was easier to understand for the audience of his article. In his PhD [83], Weir writes "The slightly stronger property of semilinearity may come closer [than the constant growth property], but is still only an approximation of what is really intended". Up to our knowledge, there is no linguistic phenomenon that would satisfy the constant growth property and not the semilinearity property. Nevertheless, as mentionned by Weir, semilinearity alone is not sufficiently strong yet. It is indeed rather simple to make highly complex semilinear languages. For example, taking an arbitrary language L on an alphabet Σ, if we take a letter $\#$ that is not in Σ, then, using Dickson's lemma [106], we can show that the language $\mathrm{sl}(L) = \{w\#w' \mid w \in \Sigma^* \wedge w' \in L\}$ is semilinear. It thus seems better to take the following definition:

Definition 35 A language L is said *strongly semilinear* when every language L' that is the image of a rational transduction of L is also semilinear.

Notice that the language $\mathrm{sl}(L)$ is strongly semilinear iff L is strongly semilinear. This definition thus eliminates some pathological cases. Ginsburg and Spanier [133] have studied classes of languages that are strongly semilinear. They have proved that there is a maximal class languages that satisfy this property, but not much is known about it so that it is hard to know how complex the class of strongly semilinear languages may be.

We nevertheless believe that the constant growth property should be rephrased into the stronger property:

> The class should contain only strongly semilinear languages.

Concerning the polynomial parsing property, we follow the opinion of Makoto Kanazawa, that it should be strengthen to LOGCFL parsing complexity[1]. This class has been introduced as a possible separator of PTIME and LOGSPACEand contains the set of problems that can be solved by a LOGSPACEreduction to the membership problem of a CFL. It is in general considered as highly parallelizable. This stronger assumption does not exclude formalisms that have been proposed to be inside mildly context sensitive languages. Indeed, though many of them are most often proved to have PTIME parsing complexities, it turns out that their parsing complexities are LOGCFL-complete.

Lastly, it is rather hard to make more precise the criterion about crossing dependencies. One of the difficulties is that it is always tempting to describe it in terms of language only, i.e. in terms of what is called the *weak generating capacity* of formalisms. On the other hand, when one is speaking about dependencies, it presupposes a notion of syntactic structures from which these dependencies should be read off. This is precisely what Joshi does [161]: so as to illustrate crossing dependencies he uses arcs preserved during TAG derivations that represent those dependencies. In this context, what we are interested in is the notion of strong generative capacity. In a sense, this calls for a different understanding of grammars that is similar to ACGs, grammars do not only

[1] Actually, functional LOGCFL is a better choice as we expect the description of the set of derivations to be produced by a function that is computed in LOGCFL.

define strings/trees/lambda-terms/graphs languages, they define pairs made of syntactic structures and surface structures. This obviously raises several questions here: what are syntactic structures? What are surface structures? How can we define the relation between them?

Before we try to dig into those questions, we first recall certain interpretations or approximations we can find in the literature about what the criterion about crossing dependencies should be. An informal description is given in [163] where they say that this criterion may at least exclude languages without structure such as $MIX = \{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$. A more formal description is given in Groenink PhD dissertation [144] which consists in imposing that each language in the class should be finitely iterable[2]. Finally Kallmeyer in one of her lecture proposes a restriction in terms copy-language: there is n so that, if $\{w^k \mid w \in \Sigma^*\}$ is in the class, then $k \leq n$.

Kallmeyer's proposal is verified as soon as we take a natural restriction on MCFLs (bound the dimension of the non-terminals). It is rather close to Joshi's initial intention to limit the parameter k in patterns of the form $a_1^{n_1} a_2^{n_2} \dots a_k^{n_k} b_1^{n_1} b_2^{n_2} \dots b_k^{n_k}$. The part of Joshi's intuition that is not captured with that pattern is that the jth occurrences of a_i and of b_i are related to each other. While the parameter k captures the *limited* part of the criterion, this last part captures its *crossing* part. Kallmeyer's criterion succeeds in making a rather concise account of Joshi's initial intuition as it contains both parts. The *limited* part is clearly modeled, and the *crossing* part seems captured by the fact that, intuitively at least, the ith letter of each occurrence of w are related to each other in each copy. However, we believe that this criterion should clearly mention a notion of derivation and a formalization of dependencies.

We will see in the next sections that, surprisingly, apart from Kallmeyer's formal interpretation, the other interpretations are not verified by MCFLs. This is surprising since, as we already mentioned, MCFLs are generally considered as being the class of languages that fits best Joshi's notion of mild context sensitivity.

Weir in his dissertation summarizes the difficulty of making the criterion about crossing dependencies precise:

> The problem of comparing the strong generative capacity of different formalisms and making general statements about how the strong generative capacity should be limited is that such criteria should be applicable to a range of radically different systems. Such criteria have been difficult to develop as, for example, the objects comprising the grammars (e.g., productions of trees), and the structural descriptions (e.g., trees, graphs) could be very different notationally.

If, as we propose above, we see strong generative capacity as a binary relation between syntactic structures and surface structures, limiting the strong

[2]This notion is a generalization of a notion we will come back to in Section 4.4

generative capacity amounts to limit the possible relations. A nice class of relations that fits with the intuitions developed around mild context sensitivity is the notion of Monadic Second Order Logic (MSOL) transduction proposed by Courcelle [86]. This class of transductions is rich enough to capture the relation between derivation trees of MCFGs and the string they generate and it is restricted enough so that MSOL transductions map regular tree languages to MCFLs [90, 119]. Moreover, as this kind of transduction is based on logic, it naturally describes, trees to graphs, graphs to strings, etc... mappings and proposes thus a possible solution to the heterogeneity problem that Weir is raising about formalizations of natural language. Another advantage of logic, is that its succinctness gives a strong leverage to organize linguistic concepts. Logic may thus be a nice way of representing the linguistic knowledge as transformational grammars or GPSG were trying to do. Moreover, logic overcomes one of the difficulties that have been systematically met when formalizing natural language: organizing the flow of contextual information. In most formalisms, such as GPSG, Lexical Functional Grammars (LFG) [64], Head-Driven Phrase Structure Grammars (HPSG) [244], the flow of information that describes contextual information has been directly implemented in grammatical rules. The main mechanism that has been used is that of unification which (when the formalism is not lexicalized) often results in capturing all recursively enumerable languages. Logic, on the other hand, allows one to easily represent long distance relations and logical connectives let one combine these relations so as to describe context with a high precision and simplicity. Finally, the equivalence of MSOL on trees and finite state automata makes MSOL transductions be restricted relations which somehow formalize a notion of mapping that uses only finite memory. As a first approximation, we are now tempted to rephrase Joshi's criterion about crossing dependencies as:

> The relation between syntactic structures and surface structures is an MSOL transduction.

This criterion while a bit more precise than Joshi's original criterion is still a bit vague as it leaves open what syntactic and surface structures are. It also leaves open whether we should choose a more restricted class of relations. A natural choice for syntactic structures is to take regular tree languages and strings for surface structures. As we already mentionned, in that case, we describe a class of relations whose surface structures form precisely the class of MCFLs. Thus, in that case, all the other criteria that we have proposed become consequences of this very one: as MCFLs is a class of strongly semilinear languages, for which parsing is in functional LOGCFL, and which contain CFLs. This illustrates how natural the class of MCFLs is.

The community of Model Theoretic Syntax (MTS) already proposed to exploit the connection between MSOL and finite state automata so as to model natural languages. The logical description of syntax has first been explored by Rogers [257, 256, 258]. This allowed him to formalize a large part (actually the least controversial one) of Chomsky's theory known as Government and

Binding (GB) [75]. This work showed that this part of GB could only model CFLs. One of the difficulty met by the MTS approach to natural language description is that it presupposes that the syntactic structures are regular tree languages and that surface structures are obtained from syntactic structures only by taking the yield of these trees. This immediately implies that the class of languages that such an architecture can describe is only the class of CFLs. Somehow, Moennich, Morawietz and their co-authors [188, 220] tried to overcome this limitation by advocating a two-step approach that would rely on a transduction from tree to strings to obtain larger classes of languages. Rogers also proposed higher-dimensional trees [259] so as to model richer classes of languages at the cost of having complex syntactic structures. Recently, we proposed a way of describing grammars using logic [S4]. We did not use MSOL, but an *ad hoc* logic (a bit less expressive than MSOL) which we found more adapted to the description of natural languages. We described syntactic structures with this logic and modeled the relation between syntactic structures and surface structures by means of a transduction with logical look around expressed with that logic and whose operations implemented with λ-terms. With this formal apparatus we proposed concise models of control and island phenomena. The interest of this logical approach is that it induces a modular description of natural language that allowed us to model these phenomena in Dutch, English, German and also Montague semantics. Of course, there is still some need to refine the methodology and model wider classes of phenomena, but those preliminary results are encouraging. Moreover, the overall architecture guaranties that the formalism is no more expressive than second-order ACGs[3].

There are some arguments that indicate that MCFLs may be a too restrictive class of languages. One such limitation comes from the modelisation of copying phenomena. Some linguistic phenomena such as Chinese numbers [250], genitive suffixes (*Suffixaufnahme*) in Old Georgian [214], or relativized predicates in Yoruba [187] seem to require some treatment that violate the semilinearity constraint. Though it is unclear whether these phenomena make natural language not semilinear or not strongly semilinear, it seems best to model them as explicitly using copies. In the context of mild context sensitivity that we have outlined, this can be achieved by making surface structures be strings with sharing, the unfolding of which would yield the desired utterance. In the proposal we made, we use the copying capability of λ-calculus to model copying phenomena. Clearly copying phenomena challenge the strong semilinear property that we stipulate for the class of mildly context sensitive languages, but it would similarly challenge the constant growth property. Concerning the complexity, explicit copying extend the class of MCFLs into Parallel Multiple Context-Free Languages (PMCFLs) which still have a LOGCFL parsing problem [118]. This problem of copying is further discussed by Stabler [273].

Free word phenomena and non-configurational languages also challenge the class of MCFLs. Though our result that MIX is a MCFL [S19] makes it unclear

[3]The ACGs are possibly non-linear if non-linear terms are used.

whether free word order languages escape the expressive power of MCFLs, if we think that this is the relation between syntactic structures and surface structures that matters in a linguistic model, then it has been showed by Becker *et al.* [49] that some free order phenomena were not in the scope of MCFLs. Here again, it seems interesting to consider that surface structures are no longer mere strings and make them represent sets of strings that are equal up to some reordering of words and which have the same syntactic structure. Crucially, extending the ability of the class of mildly context sensitive languages into a class that captures free word order phenomena should not result in languages which are not strongly semilinear nor that have parsing of high complexity. Several proposals have been made in the literature to cope with this problem. But as pointed by Schmitz [263], most of them turn out to have a complexity at least as high as the reachability problem for Petri Net, that of a tree extension Petri Nets, called Vata [S5] or BVass [288]. We have proposed [S11] yet another approach that is based on an algebra whose terms represent a set of sentences. This algebra uses two kinds of concatenations, the usual one and a commutative one. Thus modulo the equational theory of that algebra, a term represents a finite set of utterances. We consider second order ACGs which generate terms on that algebra. An important property of this architecture is that the languages those grammars define are all strongly semilinear. Other choices of operations, such as shuffle, or arbitrary counting lead to classes beyond context-sensitive languages. For the moment, the work we have pursued is mainly a work of classification and of systematic study of complexity. We have been able to identify classes with LOGCFL parsing complexties. But the classes that seem natural to model free word order phenomena such as German scrambling are NP-complete. We need to refine those classes to see whether certain restrictions can capture natural language phenomena and still have low parsing complexity. These classes of languages are not in general closed under rational transductions, but their closure under rational transductions contain interesting classes of languages such as the rational cone generated by the permutation closure of regular languages, the class of languages definable by Unordered Vector Grammars (UVG) [96] which have been used to model certain free word order phenomena by Rambow and Satta [251].

4.2 Multiple Context Free Grammars

Multiple Context-Free Grammars (MCFG) have been introduced by Seki *et al.* [266] is similar to Linear Contex-Free Rewriting Systems (LCFRS) introduced by Joshi's students, Vijay-Shanker and Weir [289]. We give here a presentation that is slightly different from the one proposed in the original article but which is slightly more intuitive.

A Multiple Context Free Grammar (MCFG) G is a tuple (Θ, Σ, R, S) where Θ is a ranked alphabet, Σ is a finite set of letters, R is a set of rules and S is an element of $\Theta^{(1)}$. The rules in R are of the form

$$A(\alpha_1, \ldots, \alpha_n) \leftarrow B_1(x_1^1, \ldots, x_{l_1}^1), \ldots, B_p(x_1^p, \ldots, x_{l_p}^p)$$

where A is in $\Theta^{(n)}$, B_j is in $\Theta^{(l_j)}$, the x_j^k are pairwise distinct variables and the α_j are elements of $(\Sigma \cup X)^*$ with $X = \{x_j^k \mid k \in [p] \wedge j \in [l_k]\}$ and the restriction that each x_j^k may have at most one occurrence[4] in the string $\alpha_1 \cdots \alpha_n$. Note that p may be equal to 0 in which case the right part of the rule is empty, in such a case we will write the rule by omitting the symbol $\leftarrow$.

An MCFG such as G defines *judgments* of the form $\vdash_G A(s_1, \ldots, s_n)$ where A is in $\Theta^{(n)}$ and the s_i belongs to Σ^*. Such a judgment is said to be *derivable* when there is a rule $A(\alpha_1, \ldots, \alpha_n) \leftarrow B_1(x_1^1, \ldots, x_{l_1}^1), \ldots, B_p(x_1^p, \ldots, x_{l_p}^p)$ and there are derivable judgments $\vdash_G B_k(w_1^k, \ldots, w_{l_k}^k)$ for all k in $[p]$ such that s_j is equal to α_j where the possible occurrences of the x_j^k are replaced by w_j^k. The language defined by G is the set $\{w \in \Sigma^* \mid S(w) \text{ is derivable}\}$.

An MCFG is said *well-nested* when all its rules:

$$A(\alpha_1, \ldots, \alpha_n) \leftarrow B_1(x_1^1, \ldots, x_{l_1}^1), \ldots, B_p(x_1^p, \ldots, x_{l_p}^p)$$

verify the following properties (where $X = \{x_j^k \mid k \in [p] \wedge j \in [l_k]\}$):

- for $i \in [p]$, if $j < l_i$ then $\alpha_1 \ldots \alpha_n \in (\Sigma \cup X)^* x_j^i (\Sigma \cup X)^* x_{j+1}^i (\Sigma \cup X)^*$,
- if $i \neq i'$, $j < l_i$ and $j' < l_{i'}$, then $\alpha_1 \ldots \alpha_n \notin (\Sigma \cup X) x_j^i (\Sigma \cup X)^* x_{j'}^{i'} (\Sigma \cup X)^* x_{j+1}^i (\Sigma \cup X)^* x_{j'+1}^{i'} (\Sigma \cup X)^*$.

This means that the variables of introduced in the right-hand side of the rule appear in the same order in its left hand-side and that furthermore, whenever, for some i' different from i, $x_{j'}^{i'}$ occurs in between x_j^i and x_{j+1}^i in $\alpha_1 \ldots \alpha_n$, then for all j'' in $[l_{i'}]$ the variable $x_{j''}^{i'}$ occurs in between x_j^i and x_{j+1}^i. The rules that satisfy these conditions are called *well-nested rules* and the class of languages that can be defined with well-nested MCFG is called *well-nested Multiple Context Free Languages* and written MCFL_{wn}.

Even though this restriction may seem intricate, it decreases the expressive power of MCFGs significantly and MCFL_{wn} is a very natural class of languages that, as we mentioned earlier, coincides with many formalisms, like non-duplicating IO and OI grammars (so that MCFL_{wn} are included in indexed languages [126]), second order ACGs of complexity 3, coupled context-free grammars [170]. Furthermore, MCFL_{wn} satisfy a strong form of pumping lemma [171], but there is a 3-MCFL that does not satisfy such a lemma [S6].

An MCFG $G = (\Theta, \Sigma, R, S)$ is a k-MCFG(r) when the maximal arity of the elements of Θ is less than k and when the maximal number of non-terminal in the right hand side of a rule in R is r. A k-MCFG, is an MCFG that is a k-MCFG(r) for some r and similarly a MCFG(r) is an MCFG that is a k-MCFG(r) for some k. It is known [266] that for each k, k-MCFLs, the languages definable by k-MCFGs, form substitution-closed full Abstract Family of Languages [132]. In particular, this implies that k-MCFLs form a class of

[4]If we allow more than one occurrence, we obtain Parallel Multiple Context Free Grammars that we mentioned earlier.

languages that is closed under rational transduction for every k. Furthermore k-MCFLs form a strictly increasing hierarchy of languages. The two-dimensional hierarchy of k-MCFL(r) has been studied in detail by Rambow and Satta [252, 253]. Their results are summarized by the following theorem.

Theorem 36

- 1-*MCFL is equal to the class of context-free languages,*
- 1-*MCFL*(r) = 1-*MCFL*$(r+1)$ *when* $r > 1$,
- 1-*MCFL*(1) *is equal to the class of linear context-free languages,*
- 2-*MCFL(2)* = 2-*MCFL(3)*
- *if* $k > 2$ *or* $r > 2$, *then* k-*MCFL*$(r) \subsetneq k$-*MCFL*$(r+1)$.

In particular, this theorem implies that, in general, given a k-MCFG, there is no k-MCFG(2) defining the same language. Interestingly this is different when we consider $\mathrm{MCFG_{wn}}$. We proved [S9] the following theorem (a slightly stronger form of that theorem has been independently obtained in [138]).

Theorem 37 k-$MCFL_{wn}$ = k-$MCFL_{wn}(2)$.

This theorem gives a way of putting k-$\mathrm{MCFG_{wn}}$ in a sort of Chomsky normal form.

A result by Staudacher [277] can be exploited so as to prove the proper inclusion of $\mathrm{MCFL_{wn}}$ into MCFL. This result gives an example of a language that is an MCFL but that is not an Indexed language and thus not a $\mathrm{MCFL_{wn}}$. But when separating classes of languages, rather than finding a language that is in one class and not in the other, it is better to characterize certain phenomena that are possible within one class and not within the other. In the case of MCFGs, one such phenomenon is copying: given a language L, the language $L^{(p)}$, the p-copying of L, is defined by:

$$L^{(p)} = \{w^p \mid w \in L\} \ .$$

It can easily be showed that MCFLs are closed under p-copying for every p, i.e. if L is an MCFL, then for every p, $L^{(p)}$ is an MCFL. Though k-MCFL is not closed under 2-copying. An interesting result Engelfriet and Skyum [122] shows that Indexed languages and thus $\mathrm{MCFL_{wn}}$ are not closed under 3-copying. More precisely, they prove that, for every L, $L^{(3)}$ is indexed iff L is an EDT0L (a restricted kind of indexed languages). In [S9], we refine slightly that result and we prove the following:

Theorem 38 $L^{(2)}$ *is a* $\mathrm{MCFL_{wn}}$ *iff* L *is a* 1-*MCFL*.

1-MCFLs are languages with a very simple structure: unary trees, and those languages are always captured by well-nested grammars. Thus, modeling linguistic phenomena with $\mathrm{MCFG_{wn}}$ implies that copying can be performed only on very simple structures. Such a hypothesis has to be confronted with linguistic models and linguistic data. So far, we have not observed complex structural copies, at least in configurational languages. This makes an interesting argument in favor of the well-nestedness constraint for the modelisation of natural language. Well-nestedness has also been studied from the point of view of dependency structures by Kuhlmann and Nivre [194] where they show that in the Pragues Dependency Treebank [150] and in the Danish Dependency Treebank [191], almost all the dependencies satisfy the well-nestedness property. Moreover and as we will see, it seems to us that $\mathrm{MCFL_{wn}}$ are closer to Joshi's definition of mildly context sensitive languages proposed than MCFLs.

4.3 The language MIX

In this section, we are going to investigate the criterion about crossing dependencies as illustrated by Joshi *et al.* in terms of the language MIX. We will sketch two results, the first one shows that MIX is actually an 2-MCFL, and the second one that it is not a 2-$\mathrm{MCFL_{wn}}$. This result gives some more weight to the argument that $\mathrm{MCFL_{wn}}$ may well be the class that best captures Joshi's notion of mild context sensitivity.

MIX is a 2-MCFL

As we have seen the language MIX has been showed as an example of a language that, according to Joshi *et al.*, should not be in a class of languages that pretends to be mildly context sensitive. This language was first introduced by Emmon Bach in one of his lecture and it is also referred to as *Bach language*. The language MIX is rationally equivalent[5] to the origin crossing language of dimension 2 [127] $O_2 = \{w \in \{a; \overline{a}; b; \overline{b}\}^* || w|_a = |w|_{\overline{a}} \wedge |w|_b = |w|_{\overline{b}}\}$ which is also the language of words representing 0 in the group $(\mathbb{Z}^2, 0, +)$ if we let $a\overline{a} = \overline{a}a = b\overline{b} = \overline{b}b = \epsilon$. As 2-MCFLs are closed under rational transductions [266], proving that O_2 is a 2-MCFL is equivalent to proving that MIX is a 2-MCFL.

In the paper [S19], we prove that MIX is generated by the grammar G (with starting symbol S) that is defined by:

1. $S(x_1x_2) \leftarrow \mathrm{Inv}(x_1, x_2)$,
2. $\mathrm{Inv}(t_1, t_2) \leftarrow \mathrm{Inv}(x_1, x_2)$ where $t_1t_2 \in \mathrm{perm}(x_1x_2a\overline{a}) \cup \mathrm{perm}(x_1x_2b\overline{b})$,
3. $\mathrm{Inv}(t_1, t_2) \leftarrow \mathrm{Inv}(x_1, x_2), \mathrm{Inv}(y_1, y_2)$ where $t_1t_2 \in \mathrm{Perm}(x_1x_2y_1y_2)$
4. $\mathrm{Inv}(\epsilon, \epsilon)$

[5]This means that there is a rational transduction mapping MIX to O_2 and another mapping O_2 to MIX

where $\mathrm{perm(a_1 \dots a_n)}$ denotes the the language

$$\{a_{\sigma(1)} \dots a_{\sigma(n)} \mid \sigma \text{ is a permutation of } [n]\} \ .$$

So as to prove that is generates exactly the words in O_2, we prove the following lemma.

Lemma 39 Given w_1 and w_2 in $\{a, \overline{a}, b, \overline{b}\}^*$, $w_1 w_2$ is in O_2 iff $\mathrm{Inv}(w_1, w_2)$ is derivable in G.

The right to left part of the equivalence is obtained by a simple induction on the derivations in G. The left to right implication is more involved, and is based on a geometric representation of strings. Words in $\{a, \overline{a}, b, \overline{b}\}^*$ can be interpreted as curves on the plane grid that is induced by $\mathbb{Z}^2$: the letter a is interpreted as going up, while $\overline{a}$ is interpreted as going down, the letter b is interpreted as going right and the letter $\overline{b}$ is interpreted as going left. An example is given

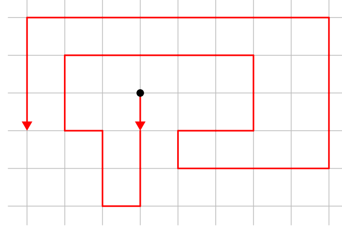


Figure 4.1: Curve representation of $\overline{a}\overline{a}\overline{a}\overline{b}aa\overline{b}aabbbbb\overline{a}\overline{a}\overline{b}\overline{b}\overline{a}bbbbaaaa\overline{b}\overline{b}\overline{b}\overline{b}\overline{b}\overline{b}\overline{b}\overline{b}\overline{a}\overline{a}\overline{a}$

Figure 4.1. With this graphical representation, pairs of strings can be seen as two curves concatenated to each other. A pair of words (w_1, w_2) is so that $w_1 w_2$ is in O_2 iff the curve that represents $w_1 w_2$ is closed.

The proof that if $w_1 w_2 \in O_2$, then $\mathrm{Inv}(w_1, w_2)$ is derivable in G is an induction $|w_1 w_2| + \max(|w_1|, |w_2|)$. The cases where either w_1 or w_2 is the empty string is treated simply by induction by splitting the non empty part of the pair. In case on the borders (their first and last letters) of w_1 and w_2 contain a pair of compatible letters (either a and $\overline{a}$ or b and $\overline{b}$), then a simple use of the induction hypothesis gives the conclusion. Another simple case is when the arcs described by w_1 and w_2 intersect each other, in that case, we must have that $w_1 = v_1 v_2$ and $w_2 = u_1 u_2$ so that $v_1 u_2$ and $v_2 u_1$ are in O_2. Here again the induction hypothesis immediately gives the conclusion. The last easy case is when either w_1 or w_2 has a left or right non-trivial factor that is in O_2: for example if $w_1 = v w_1'$ with $v \in O_2 - \{\varepsilon\}$, then the induction hypothesis gives that $\mathrm{Inv}(v, \varepsilon)$ and $\mathrm{Inv(w_1', w_2)}$ are derivable and we are done.

Now when looking at the complement of all those cases, we must have that the pair (w_1, w_2) satisfies the following properties:

1. neither w_1, nor w_2 is equal to ε,

2. they have no compatible letters on their borders and, using symmetries, we may assume that the first and last letters of w_1 and w_2 are in $\{a, b\}$,

3. the curves representing w_1 and w_2 do not intersect each other,

4. neither w_1 nor w_2 have a left or right factor that is in $O_2 - \{\varepsilon\}$.

So as to complete the proof, we prove that whenever a pair of words (w_1, w_2) verifies all these properties, then it has the following decomposition property: either $w_1 = u_1u_2u_3$ or $w_2 = u_1u_2u_3$ with u_1, u_2 and u_3 all different of ε and respectively, both u_1u_3 and u_2w_2 are in O_2, or both u_1u_3 and w_1u_2 in are in O_2. The proof of this property is the most technical part of the proof and is treated mainly geometrical and topological means.

We start by restricting our attention to pairs (w_1, w_2) so that neither w_1 nor w_2 contain a factor in $O_2 - \{\varepsilon\}$. Having the decomposition property for this case implies almost immediately the decomposition property for the general case. Indeed, if w_1 and w_2 contain factors in $O_2 - \{\varepsilon\}$, let w_1' and w_2' be obtained by removing those factors until no more remains. Because of the conditions 1 and 4, it cannot be the case that w_1' nor w_2' are equal to ε. Moreover, condition 4 also implies that the letters on the borders of w_1' and w_2' are the same as the ones on the borders of w_1 and w_2. Now if we can obtain the expected composition of the pair (w_1', w_2'), then by putting back the factor of $O_2 - \{\varepsilon\}$ that we have removed in that decomposition, then we obtain a decomposition of (w_1, w_2).

Considering pairs (w_1, w_2) for which neither w_1 nor w_2 contain a factor in $O_2 - \{\varepsilon\}$ has as consequence that the curve representing w_1w_2 is a closed curve that is not self-intersecting, in other words, it is a Jordan curve. A Jordan curve divides the plane into two components a bounded one, its interior, and an unbounded one, its exterior. When orienting the curve, we can distinguish its left from its right. At every point of the curve, its interior is always on the same side. Exploiting this fact, we can identify the property that causes the existence of a decomposition. This property also allows us to forget the combinatorial aspect of the problem and use purely topological methods so as to solve it.

So as to define this property, we remark that if we consider a simple (i.e. not self-intersecting) arc on the square grid, then certain squares that are adjacent to that arc will be in the interior or in the exterior of any Jordan curve that contains that arc. Figure 4.2 shows those square for a particular arc, moreover, we take the convention that we color the squares on the left of the arc in green and the ones on its right in yellow. Now, because of condition 2, we can enumerate the possible configurations of the curves near the two points where the arcs meet each other. We shall call A the point from which the curve representing w_1 starts and D the one where it ends. This means that the curve representing w_2 starts at D and ends at A. Enumerating those possible configurations, it happens that we can conclude that inside the closed curve that w_1 and w_2 draw there are two points A' and D' so that the vector $\overrightarrow{A'D'}$ is equal to the vector $\overrightarrow{AD}$. It suffices to take either $A' = A + \frac{1}{2}(1, -1)$ and $D' =$

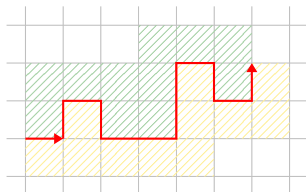


Figure 4.2: Adjacent squares on the left (in green) or on the right (in yellow) of an arc.

$D + \frac{1}{2}(1, -1)$ or $A' = A + \frac{1}{2}(-1, 1)$ and $D' = D + \frac{1}{2}(-1, 1)$. The enumeration of the configurations is given by Figure 4.3 and Figure 4.4. Figure 4.3 present the general cases, and the second one 4.4 presents the cases where either w_1 or w_2 is one letter long.

For these enumerations, we only present the case where the first letter of w_1 is a, the case where it starts with b is symmetric. On these figures, following our convention, the adjacent squares on the left and on the right of the arcs have been colored respectively in green and yellow. Moreover, we have drawn some green arrows representing A' and D' when there are defined by $A' = A + \frac{1}{2}(-1, 1)$ and $D' = D + \frac{1}{2}(-1, 1)$ and we can remark that in any case the end of both green arrows are always in the green adjacent squares meaning that whenever the arcs are part of a Jordan curve whose interior is on its left, then both A' and D' are in its interior. Similarly we have drawn some yellow arrows representing A' and D' when they are defined by $A' = A + \frac{1}{2}(1, -1)$ and $D' = D + \frac{1}{2}(1, -1)$, and similarly we can observe that when the arcs are part of a Jordan curve whose interior is on its right then both A' and D' as defined by those yellow arrow are in its interior. Figures 4.5 and 4.6 illustrate how some configurations can be completed into a Jordan curve; they also show that, as expected, either both the green or both the yellow arrows are inside the Jordan curve.

So as to complete the proof, it suffices to show that the existence of such A' and D' is enough to ensure the existence of a decomposition. For this we prove a general theorem about Jordan curves. With some small combinatorial reasoning, this theorem can be use to ensure the existence of a decomposition and thus complete the proof of Lemma 39.

Theorem 40 *Given a Jordan curve $\mathcal{J}$ and two points A and D on $\mathcal{J}$ if there is A' and D' in the interior of $\mathcal{J}$ so that the vectors $\overrightarrow{A'D'}$ and $\overrightarrow{AD}$ are equal, then there are B and C both different from A' and D' so that:*

- *either B and C appear in that order on the arc of $\mathcal{J}$ going from A to D and $\overrightarrow{BC} = \overrightarrow{AD}$,*
- *or C and B appear in that order on the arc of $\mathcal{J}$ going from D to A and $\overrightarrow{CB} = \overrightarrow{DA}$.*

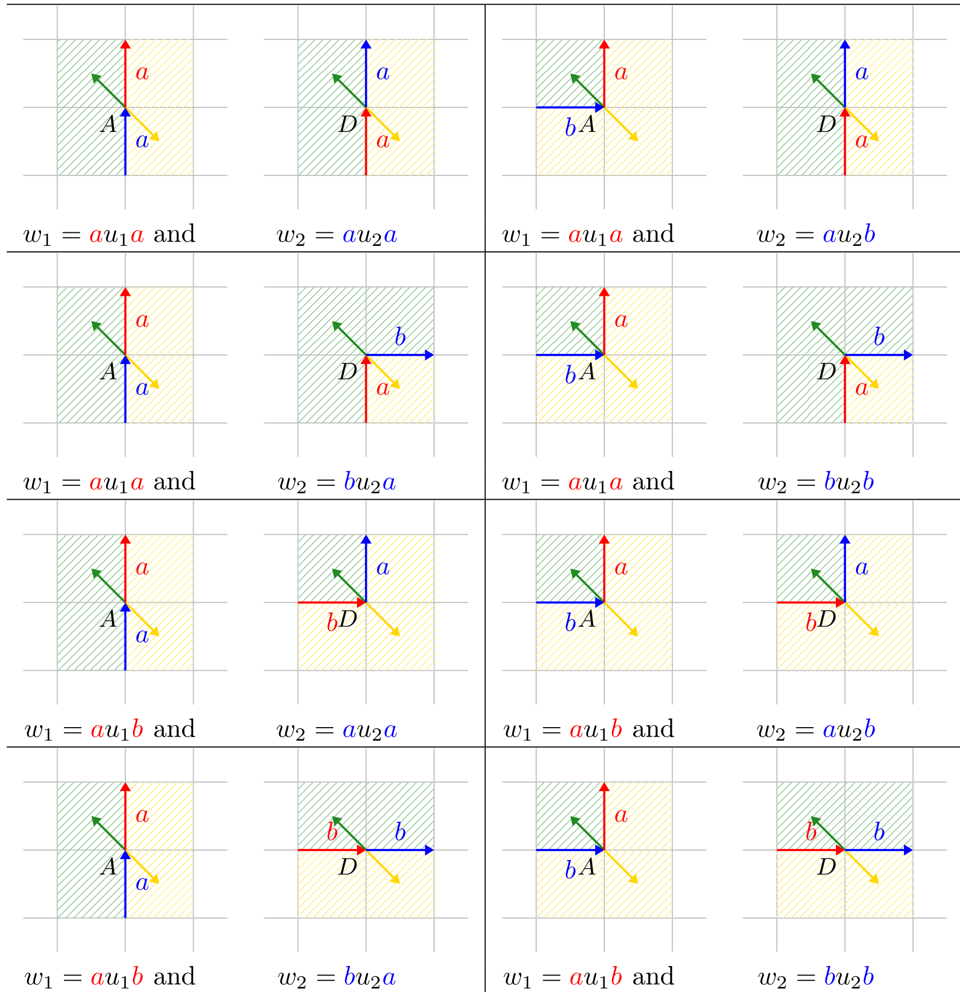


Figure 4.3: All cases where $w_1 = au_1l$ and $w_2 = s_1u_2s_2$ with $l, s_1, s_2 \in \{a, b\}$

The proof of this theorem is mainly based on algebraic topology (see [271]). In this context, it is possible to strengthen slightly the hypothesis of the theorem: we first assume that $\overrightarrow{AD} = (1, 0)$, that A' is the point $(0, 0)$, that D' is the point $(1, 0)$ and that the curve $\mathcal{J}$ lies in the countably punctured plane, i.e. the plane where the points $\{(k, 0) \mid k \in \mathbb{Z}\}$ have been removed. Now if we confuse the plane with complex numbers $\mathbb{C}$, the continuous mapping $\exp(z) = e^{2i\pi z}$ maps the countably punctured plane to the twice punctured plane, i.e. the plane where the points $(0, 0)$ and $(1, 0)$ have been removed. Moreover, from algebraic topology we obtain that if $\mathcal{J}$ is a curve that contains p elements of $\{(k, 0) \mid k \in \mathbb{Z}\}$, then $\exp(\mathcal{J})$ is a curve of the twice punctured plan that winds p (or, depending on the orientation of $\mathcal{J}$, $-p$) times around $(1, 0)$. So when we

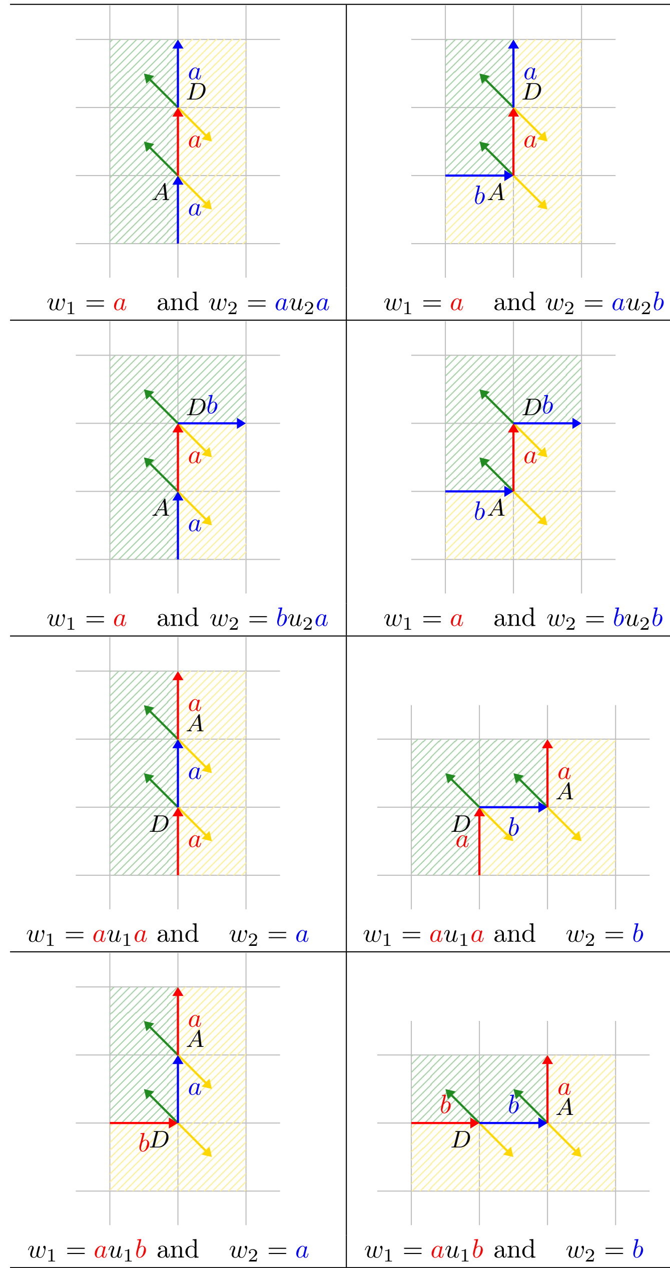


Figure 4.4: All cases where $w_1 = a$ and $w_2 = s_1u_2s_2$ or $w_1 = au_1l$ and $w_2 = s$ with $l, s, s_1, s_2 \in \{a, b\}$.

consider a curve $\mathcal{J}$ satisfying the hypotheses of Theorem 40, the curve $\exp(\mathcal{J})$

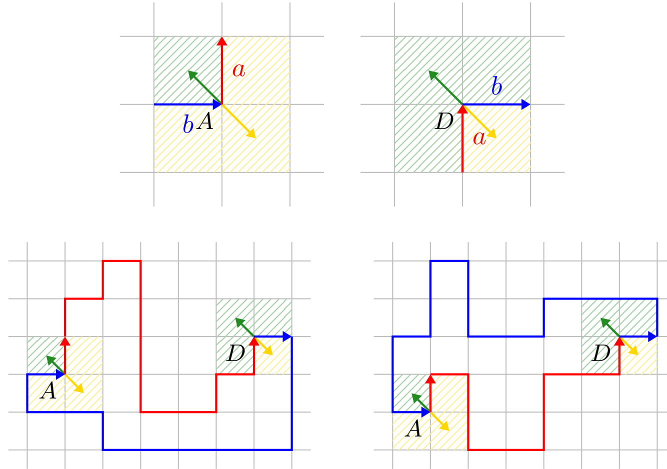


Figure 4.5: Illustrations of the case where w_1 is of the form au_1a and w_2 is of the form bu_2b

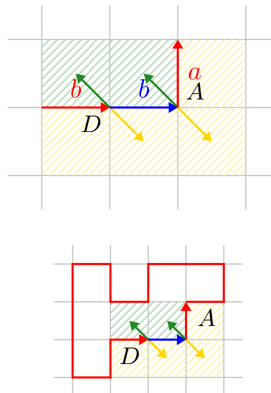


Figure 4.6: Illustration of the case where w_1 is of the form au_1b and $w_2 = b$

must wind at least 2 (or at most -2) times around $(1, 0)$. Then Theorem 40 follows from the following intuitive lemma.

Lemma 41 Given A and D in the countably punctured plane so that $\overrightarrow{AD} = (1, 0)$ (*resp.* $(-1, 0)$) and an arc $\mathcal{C}$ from A to D, the following properties are equivalent:

- there is no point B and C (*resp.* C and B) in that order on $\mathcal{C}$ that are both different from A and D and so that $\overrightarrow{BC} = (1, 0)$,
- $\exp(\mathcal{C})$ is a Jordan curve of the twice punctured plane.

Indeed, suppose that $\mathcal{J}$ is a Jordan curve verifying the hypotheses of Theorem 40. If $\mathcal{J}$ the conclusion of Theorem 40 is not fulfilled, then Lemma 41 entails that the arc of $\mathcal{J}$ going from A to D is mapped to a Jordan curve by exp and so it may wind either 0 or 1 time around $(1, 0)$, similarly the arc of $\mathcal{J}$ going from D to A is mapped to a Jordan curve by exp that may wind either 0 or -1 time around $(1, 0)$. Therefore, $\exp(\mathcal{J})$ is a curve that may wind either -1, 0 or 1 time around $(1, 0)$. But we have seen that the hypotheses of Theorem 40 implied that the winding number of $\exp(\mathcal{J})$ around $(1, 0)$ should be in $]-\infty, -2] \cup [2, +\infty[$, yielding a contradiction.

MIX is not a 2-MCFL$_{\text{wn}}$

In the course of the proof that O_2 is a 2-MCFL, the difficult case is the only one where the induction hypothesis is using on non-well-nested rules of the grammar. A natural question is then whether those rules are necessary for the grammar to capture O_2. As it happens this is the case, if we were to remove the non-well-nested rules from the grammar, then the word

$$aabb\overline{a}\overline{b}\overline{a}bbaaa\overline{bbb}\overline{aaa}b$$

whose curve representation is given Figure 4.7, would not be recognized by the grammar. Using this observation, it is then possible (see [S8]) to show that

Figure 4.7: Curve representation of $aabb\overline{a}\overline{b}\overline{a}bbaaa\overline{bbb}\overline{aaa}b$

MIX is not a 2-MCFL$_{\text{wn}}$ and thereby, as the class of 2-MCFL$_{\text{wn}}$ is the same as the class of languages definable by tree-adjoining grammars, this closes the long-standing open conjecture of Joshi that MIX is not definable by a tree-adjoining grammar.

The proof is in two steps: first we reduce the problem of MIX being a 2-MCFL$_{\text{wn}}$ to the problem of MIX being captured by a particular grammar; second we use a computer program to prove that a word in MIX (which is a variant of the word of the word of O_2 of Figure 4.7) is not in the language generated by that grammar.

The first step is based on simple ideas. The first one consists in remarking that if G is a 2-MCFG$_{\text{wn}}$ deriving MIX, then, for each of its non-terminal A, the Parikh image of words $w_1 w_2$ so that $A(w_1, w_2)$ is derivable must be included in sets of the form $\{(k, k, k) + v | k \in \mathbb{N}\}$ for a fixed vector v in $\mathbb{N}^3$. Such a grammar G is said p-bounded when each component of the vector v associated to a non-terminal A of G is smaller than p. Given a word w

in $\{a, b, c\}^*$, we write $w \cdot k$ for the word obtained from w by replacing each occurrence of a by a^k, each occurrence of b by b^k and each occurrence of c by c^k. Using the fact that when w is in MIX, then $w \cdot k$ is also in MIX allows us to show that whenever MIX is recognized by a p-bounded grammar then it is recognized by a 2-bounded grammar. With this it is then possible to reduce the problem of MIX being a 2-MCFL$_{\text{wn}}$ to the problem whether it is definable by a particular 2-bounded 2-MCFG$_{\text{wn}}$.

We conjecture that MIX is not a MCFL$_{\text{wn}}$. It seems hard to generalize our approach. Our method allows us to reduce that MIX is a k-MCFL$_{\text{wn}}$ to MIX being defined by a particular k-bounded k-MCFL$_{\text{wn}}$. But we cannot get any intuition of how to generalize the word that fails to be in the grammar and thus cannot find any induction argument on k. It may have been somewhat easier if we could have reduced the problem to 0-bounded k-MCFL$_{\text{wn}}$. But even proving that MIX is not recognized by any 0-bounded 3-MCFL$_{\text{wn}}$ seems hard. The main issue being to find with a computer program an example of a word that is not recognized by such a grammar.

Recently a new proof based on Ogden style pumping lemma for 2-MCFL$_{\text{wn}}$ has been proposed by Sorokin [270]. Nevertheless this method cannot be extended to higher dimensions as such pumping properties do no hold in general for k-MCFL$_{\text{wn}}$ when $k > 2$ (see [166]).

4.4 Iteration properties for MCFGs

We now turn towards the iteration properties of MCFL and show that Groenink's interpretation [144] about crossing dependencies is not verified by MCFLs. Again, it has been showed by Kanazawa [171] that MCFL$_{\text{wn}}$ satisfied intuitive iteration properties.

Iteration properties in languages have been formalized by Greibach [139, 140]. She distinguishes two notions:

Definition 42 (k-iterativity) A language L is k-iterative if there is a constant n so that if $w \in L$ and $|w| > c$ then we have:

- $w = u_0 w_1 \dots u_{k-1} w_k u_k$ so that $w_1 \dots w_k \neq \varepsilon$ and,
- for every i in $\mathbb{N}$, $u_0 w_1^i \dots u_{k-1} w_k^i u_k$ is in L.

Definition 43 (Weak k-iterativity) A language L is weakly k-iterative if either it is finite or it contains a language of the form

$$\{u_0 w_1^i u_1 \dots u_{k-1} w_k^i u_k \mid i \in \mathbb{N}\}$$

where $w_1 \dots w_k \neq \varepsilon$.

It has been showed by Seki *et al.* [266] that either every k-MCFL is weakly $2k$-iterative.

Theorem 44 *Every k-MCFL is weakly $2k$-iterative.*

Strangely Radzinski claimed [250] that k-MCFLs were all $2k$-iterative. More precisely, he claimed that iterations in the derivations trees of MCFLs translated into iterations in the generated string which is far from obvious. However, this claim has been taken for granted by Goenink [144] and also by Kracht [189].

Even the fact that k-MCFLs$_{\mathrm{wn}}$ are $2k$-iterative requires a non-trivial proof. The proof given by Kanazawa [171] actually does not translate iteration in the derivation tree into iteration in the string language, it actually requires some transformation of the tree structure of the grammar so as to achieve the iterativity at the level of strings. Kanazawa shows moreover that 2-MCFLs are 4-iterative:

Theorem 45 *Every k-MCFL$_{\mathrm{wn}}$ is $2k$-iterative and every 2-MCFL is 4-iterative.*

In [S6] we prove the following theorem:

Theorem 46 *There is a 3-MCFL L that contains an infinite language L' so that for every word w in L' and every n in $\mathbb{N}$, if $w = u_0 w_1 \dots u_{n-1} w_n u_n$ with $w_1 \dots w_n \neq \varepsilon$, then*

$$|L \cap \{u_0 w_1^i \dots u_{n-1} w_n^i u_n \mid i > 1\}| \leq 1$$

The language L can be thus seen as *strongly anti-iterative*. This means that somehow for $k > 2$, k-MCFL do not satisfy any sensible strong iteration property.

The language L of Theorem 46 is defined by the following grammar G (its starting symbol is H):

1. $H(x_2) \leftarrow J(x_1, x_2, x_3)$,
2. $J(ax_1, y_1 c x_2 \bar{c} d y_2 \bar{d} x_3, y_3 b) \leftarrow J(x_1, x_2, x_3)\,, J(y_1, y_2, y_3)$
3. $J(a, \varepsilon, b)$

If we delete the occurrences of a and of b in L, we obtain a language which is a homomorphic representation of binary trees, where:

1. a node with two children t_1 and t_2 is mapped to $c w_1 \bar{c} d w_2 \bar{d}$ if t_1 and t_2 are respectively represented by w_1 and w_2, and
2. a leaf is mapped to the empty string ε.

Notice that this gives a bijection between binary trees and their representations in the language generated by G. In that language, each representation of a binary tree is decorated with some more information as follows:

1. a node with two children t_1 and t_2 is mapped to $a^m c w_1 \bar{c} d w_2 \bar{d} b^n$ if t_1 and t_2 are respectively represented by w_1 and w_2, and m is the length of the left most branch of t_2 and n is the length of the right-most branch of t_1,

2. a leaf is mapped to the empty string ε.

So as to prove this, it suffices to proceed by induction on the size of derivations and remark that when $J(u_1, u_2, u_3)$, then we always have that $u_1 = a^{k+1}$, $u_3 = b^{l+1}$ with $k, l \in \mathbb{N}$ and u_2 is either ε or a word of the form $a^m cv\overline{c}dw\overline{d}b^n$. In the latter case, the unique derivation tree of $J(a^{k+1}, a^m cv\overline{c}dw\overline{d}b^n, b^{l+1})$ is a

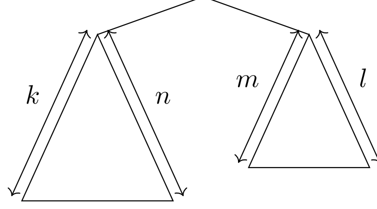


Figure 4.8: Derivation tree for $J(a^{k+1}, a^m cv\overline{c}dw\overline{d}b^n, b^{l+1})$

binary tree where $k+1$ is the length of its leftmost branch, $l+1$ is the length of its rightmost branch, m is the length of the leftmost branch of its right daughter, and n is the length of the rightmost branch of its left daughter. The situation is illustrated Figure 4.8.

Now the language L' of Theorem 46, is simply the set of words whose derivation trees are complete binary trees. The language L' is thus the set $\{v_n \mid n \in \mathbb{N}\}$ where v_n is inductively defined as:

- $v_0 = \varepsilon$,
- $v_{n+1} = a^{n+1} cv_n\overline{c}dv_n\overline{d}b^{n+1}$.

The proof that whenever one iterate a fixed number of factors in a word v_n, then the iterations are all, except maybe one, outside L is highly technical. The intuition is that when iterating factors in a word v_k, one needs to make a left or a right branch of a subtree grow. Each time this branch grows, so as to remain in L, it requires the update of one extra counter made of a's or of b's. Due to the non-well-nestedness of the rules, this extra counter is in a place different from the the branch that is being grown. Moreover, the combinatoric is so that it needs to be either strictly inside a factor that is is either being iterated or a factor that is not. The effect is that so as to remain in L at each iteration there would be a need to iterate more factor. An immediate consequence is that the strings obtained by iterations are not in L.

4.5 Classifying Mildly Context Sensitive Formalisms

Somehow in view of those results it seems that the class of languages that is the closest to Joshi's definition is that of well-nested Multiple Context Free Languages ($\mathrm{MCFL_{wn}}$) that have been introduced by Kanazawa [170] based on ideas of Kuhlman and Möhl [192, 193]. $\mathrm{MCFL_{wn}}$ is yet another class of languages that is defined by a wide variety of formalisms such as non-copying

IO/OI languages, coupled context-free grammars [154], string languages definable by ACGs in $\mathcal{L}(2,3)$. Moreover they generalize TAGs in a natural way. We have conjectured that MIX is not a MCFL_{wn}, and Kanazawa showed that every k-MCFL_{wn} is $2k$-iterable. This coincides with the interpretation of Groenink concerning crossing dependencies for mild context sensitivity. The 3-MCFL that shows that MCFL are not iterable in general crucially uses non-well-nested rules so as to construct complex dependencies which clearly go beyond the kinds of dependency that Joshi describes as being the ones that are of interest of natural language modeling.

Apart from LCFRS, Weir describes another hierarchy or languages [293] called *control languages*. We will not go into the details of the definition of that hierarchy. This hierarchy inductively defines classes of languages indexed by natural numbers. The idea consists in controlling the derivations of a context-free language. The first level of the hierarchy is simply the class of context-free languages. A language of level $k+1$ is defined from a context-free grammar for which each derivation tree is assigned a spine (i.e. a branch) with a rule-based mechanism, then a derivation is licensed when each spine of its sub-derivation belongs to a fixed language of level k. It is remarkable that the second level of this hierarchy coincides with Tree Adjoining Languages. The idea of using spines captures the linguistic idea that constituents are constructed around a principal element, its head. It is easy to remark that Staudacher language is definable in that hierarchy and that this hierarchy is also closed under copying. Thus, there are languages that are in the hierarchy but which are not MCFLs_{wn}. Palis and Shende [237] proved that each level of that hierarchy enjoys an Ogden style pumping lemma [232]. A recent note of Kanazawa [166] shows that MCFLs_{wn} do not in general satisfy Ogden style pumping lemma. If one were to discriminate between Weir's hierarchy and MCFL_{wn} for modeling natural languages, there would thus be two kinds of criteria:

- the need for arbitrary finite copying would push in favor of Weir's hierarchy,
- simpler crossing dependencies that allow for Ogden style pumping lemma would also push in favor of Weir's hierarchy.

We also proved that Weir's hierarchy was included in MCFLs [S7]. Figure 4.9 presents a quick summary of the classes of languages that have been proposed to capture the properties of mildly context sensitive languages.

4.6 Conclusion and perspectives

In this chapter, we have given some results that are in contradiction with formerly assumed conjectures. These results were particularly hard and technical to obtain and often using tools that are not usual in the context formal language theory. Nevertheless, they gave us the impetus to question further Joshi's central notion of mildly context sensitive languages. In particular, the

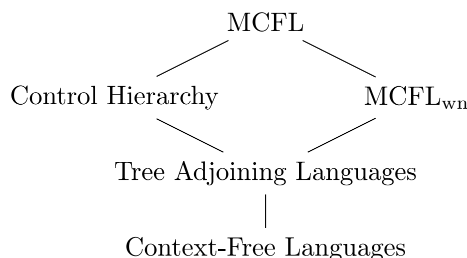


Figure 4.9: Classes of purposed mildly context sensitive languages

various interpretations about the limitation of crossing dependencies makes us believe that the notion Joshi intended to define is best captured by MCFL$_{wn}$. To be more conclusive, we would need to prove that the language MIX is not a MCFL$_{wn}$. This problem seems particularly hard, but taking the language MIX_4 that is the language of permutations of the words in $(abcd)^*$ seems to be easier and may give some evidence that MCFL$_{wn}$ are sufficiently restricted. For this, a possible route would be to adapt a result by Rozoy on Dyck language [260] to the two-sided Dyck language and then use the copying theorem we proved in [S9] so as to conclude. This seems technical but feasible. We have also tried to revisit Joshi's notion in the light of developments that were not available to him at the time of his definition. In particular, we argue that MSOL transductions is the right tool to set an upper bound on the way dependencies between constituent can be read off syntactic structures. MSOL is indeed a tool of choice that is at the same time expressive and flexible, but also restricted in the sense that this logic, via its connection with finite state automata, formalizes the notion of finite memory on logical structures. It is likely that MSOL transductions are too powerful for natural language, but the hypothesis may be improved by taking simpler kinds of transductions. Importantly, this hypothesis may be considered with an extended notion of surface structure that would allow sharing so as to model copying and also commutative concatenations so as to model free-word order phenomena.

This has led us to provide some logical model of natural languages which has the advantage to be adapted to the development of multi-lingual grammars. A first continuation to this work would be to model more phenomena and try to compare in a more fine grained way various constructions in various languages. In particular, we wish to model some free word order phenomena using the algebra we proposed. Up to now, we have studied it from a complexity theoretic perspective. The outcome of this study points towards fragments that satisfy the mildly context sensitive constraints (in particular LOGCFL complexity for parsing and strong semilinearity) and we now need to see if we can use them to account naturally for free word order phenomena observed in natural language. We also hope that we can propose synchronous grammars that relate configurational languages and free-word order ones. Such results would naturally

allow for the implementation of automatic translation systems between those languages. Other complex phenomena such as ellipsis may also benefit from the fresh look the logical approach brings to modelization. Our idea is to follow Kobele [208] and assume that ellipses are modeled by syntactic structure where some parts are deleted/elided. These parts are to be found in the context, for this the use of logic to relate elided trees to their possible *antecedent* seems to be promising. Indeed, if we can model this relation with logic then the semantic interpretation of ellipsis can be modeled in a nice way with Montague semantics, allowing for parts of the meaning to be used several times.

The use of MSOL and of logically defined transductions as means of describing language asks for tools that compile actual grammars. For surface structures (be it semantic interpretation or phonological interpretation), we would obtain grammars that are second order ACGs or extensions of them with the algebras we have proposed for free word orders. In the previous chapter, we have proposed to obtain parsers for second order (almost affine) ACGs by using datalog and datalog program transformations. This overall architecture constitutes a chain of compilation from a logically defined grammar to an optimized datalog program. This compilation process may be highly complicated and ask for a huge amount of computational resources. In particular, the compilation of MSOL formulae to finite state automata is in general non-elementary, the height of the exponential being determined by the number of alternations between existential and universal quantifiers. We can remark that, in the formulae proposed by Rogers [256] and in the ones we used in our own models [S4], the alternations of quantifiers is limited to 2 or 3. Still this may cause some complexity problems and we need to make experiments so as to assess the feasibility of such a system of description of language. These experimental studies will of course be concerned in finding some compilation heuristics. For example, an analysis of the logical formulae reveals that many constraints can be verified locally. Such constraints can be implemented by using extra parameters in datalog programs. Being able to identify such constraints and compile them using extra parameters would thus make the output programs more compact and also the compilation time probably much shorter. We need to make such analyses automatic and find algorithms that choose to implement automata transitions with extra parameters or directly as datalog predicates. A difficulty here is whether to use tools that can compile MSOL formulae to automata such as MONA [176] or build our own tool. On the one hand, MONA uses a particular representation of trees and may not suit our needs for modeling. On the other hand, developing a tool is costly and it will be hard to reach MONA's efficiency. So probably we will first make some experiment with MONA so as to compile part of our linguistic specifications and then, if needed, develop a dedicated tool.

Related to parsing, we need also to adapt the datalog approach to parsing to free-word orders. When looking at the algorithms we have described in [S11], we observe that this amounts to incorporate some counting capabilities in the resolution mechanism. This mechanism is instantiated by the computation of a representation of semilinear sets that represent the freely ordered parts of

sentences. In the compilation process, we will need to compute this semilinear set and then to allow datalog to make tests against this set. We need to see whether this requires an extension of datalog or if we may directly use datalog so as to perform those computations.

Another interesting problem is whether we can generalize the result we obtained about MIX being a MCFL to languages that mix a larger number of letters. The difficulty is that our method is essentially using properties of the plane and that it is very hard to understand the expressive power of MCFLs. In this situation, we refrain from making any conjecture. Nevertheless, if it happens that MIX like languages over an arbitrary number of letters happen to be MCFLs, then from a result by Latteux [200], it implies that every language which is the inverse image by Parikh mapping of a semilinear set is an MCFL. Such a result would be of high importance, not only for its modeling capability in natural language but also in verification as it would allow the modelization of systems with counting capabilities in a rather simple setting.

Chapter 5

Logic and Verification

Program verification consists in designing algorithms that are able to check that programs satisfy their specifications, in other words it consists in building programs that certify that other programs work. Rice's theorem [255] tells us that the problem is undecidable and thus that such programs do not exist in general. Reliable programs meet a strong demand in our computerized society. When programs control industrial installations ranging from refineries to nuclear plants, when they pilot cars, planes, trains, when they operate surgical instruments, bugs may have devastating consequences. Similarly as Gödel's theorem did not abolish the search for new theorems in mathematics, its relative, Rice theorem, is only a conceptual barrier that tells us that there is no systematic method so as to establish the correctness of programs.

A rather large consensus has emerged in computer science that programs are too complex objects so as to be amenable to paper and pencil proofs of correctness. It seems that the only objects that can cope with program complexity are programs themselves. There are several parameters that can be adjusted so as to make the problem of program verification automatic; a first obvious parameter is the degree of automation; a second parameter is the pair class of programs/class of specifications. Type theory is a good example of an approach to program verification that is able to adjust these parameters. For example, Damas-Hindley-Milner type inference [153, 217, 100, 99] allows to automatically ensure memory safety for functional programming languages such as OCaml, Haskell. In that case, the method is fully automatic and is used in practice to verify large software. The specification is rather weak, but it already rules out many common programming errors. The price to pay for this full automaticity is that there are programs which would be memory safe that are rejected by the type inference mechanism. Nevertheless, this restriction is worse paying for the gain and, in practice, it is rarely (never?) barring interesting programs from being accepted. At the other end of the spectrum Martin-Löf type theory and Higher-Order Logic allow programmers to formally fully specify and prove their programs. These logics are implemented in proof assistants such as CoQ, Agda, HOL, Isabelle etc... In this setting the effort to

prove the program correct relies on programmers. The gain is that the proofs are checked by machines. This guaranties that no case has been omitted and that the proofs are correct. These methods are more often called certification methods than verification methods which are usually relying more on programs to obtain proofs of correctness of programs.

Between fully automatic and manual ones (such as in CoQ), there is a wide range of possibilities. Certain methods such as Why3 [60, 59] or the B method [35, 34] form an interesting balance between automation and manual proofs. Moreover, in the context of CoQ, the development of programs can rely on various automation techniques such as the ones offered by the language LTac or by reflection methods. These techniques have made proof assistants able to cope with large scale mathematical or software projects. When certifying/verifying a program with another program, we need to rely on that other program. An advantage of proof-assistants is that they rely on very small cores to check programs, these cores are rather easy to implement and thus they present less risk to be trusted than larger piece of softwares. This is why recent projects have focused on developing program performing automatic verification within proof assistants thus constructing another bridge between manual and automatic methods [164].

Among fully automatic methods, there are also incomplete methods, i.e. methods that can certify that a program a correct but may not be able to prove a program incorrect. A general theory of this method is proposed by Cousot and Cousot under the name of *abstract interpretation* [94]. These methods have been able to verify simple properties of industrial softwares. The program *Astrée* is the flagship of these methods and has been used in many cases [95].

Finite state methods and their connection with logic offer a rather rich class of specifications that can can be used effectively for many classes of programs. The interest of finite state methods is that they provide in general fully automatic and complete verification algorithms. Rabin Theorem [248] opened the way to the development of a large body of work around these methods. Though this result is mainly formulated in terms of logic, it implies that it is possible to automatically verify that the possibly infinite executions of non-deterministic finite state machines satisfy MSOL specifications.

When considering possibly infinite behaviors, the connection between logic and finite state machines is not as simple as in the case of finite structures. Indeed, as the recognition process may never stop, there is a need for conditions that discriminate among infinite runs those which are accepting. A formulation of such a condition is *parity condition*. Nevertheless, regular programs are sufficiently rich to realize any coherent MSOL specification [249]. So if we are to check a program against an MSOL specification, as it would have been possible to generate a program satisfying the MSOL specification, this means that we may only check part of the program's functionality.

In this chapter, we are going to present some verification methods that are based on MSOL specifications. From a technical perspective, these methods present the interest of integrating wide areas of research that have become independent but that have the same origin: schematology, λ-calculus, finite-

state automata, denotational semantics, linear logic, typing. From an abstract perspective, this chapter is describing how models of λY-calculus are able to recognize (in the sense of Section 2.5) programs that satisfy certain logical specifications. From the perspective of verification, this work provides some effective methods so as to check certain properties of programs. From the perspective of denotational semantics, it provides new kinds of conceptual problems and asks for new methods of model construction. From the perspective of automata theory, it allows us to use high level methods coming from denotational semantics so as to model finite state properties.

5.1 Schematology

Programming languages come with basic operations such as integer addition, calls to the operating system etc... and with some way of organizing those operations which is called the *control flow* of the program. Scott and Elgot [113] proposed to see the execution trees of programs as an intermediate step towards the program semantics. This method consists in splitting the semantic interpretation of programs in two steps:

- interpret the control-flow of the program and obtain a possibly infinite tree, the execution tree, composed with basic operations of the programming language,
- evaluate the execution tree, in a specific domain so as to obtain a denotational value of the program.

This perspective has been adopted, among others, by Manna [210, 209], Wand [291, 292], Nivat [230], and Courcelle [84]. *Schematology* has appeared, the study of evaluation trees generated by programs, or in other words the study of program's semantics in the free interpretation. This perspective on the semantics of programs opens the possibility of verifying their properties simply by verifying syntactic properties of their execution trees.

Consider an ML code generating a Javascript program to be executed on a client machine. The code reads a command from an untrusted stream, surrounds it by an alert function, and passes it to the server.

```
let makecode(x)="<script>␣alert("  + x +  ");␣</script>"
in
    y=first(untrusted_stream);
    output(makecode(y));
```

An attacker can prepare a special string in order to escape the `alert()` function and execute arbitrary code which may in particular lure a client into disclosing secret information such as her password, or a bank account number:

```
makecode(");␣form.submit(http://...);")=
                              alert(); form.submit(http://...);
```

The defense against this attack is to use a validation function that removes potentially dangerous parts of the input.

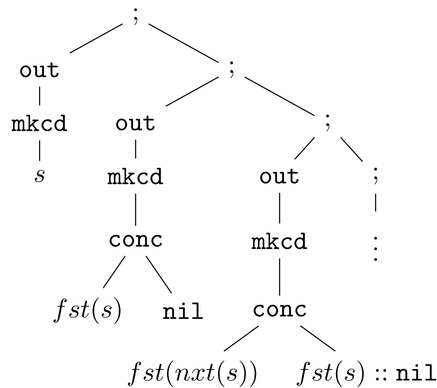
```
let makecode(x)="<script> alert(" + validate(x) + "); </script>"
in
     y=first(untrusted_stream);
     output(makecode(y));
```

This defense strategy is a programming guideline: "all strings sent to the client should be validated". Let us see how we can approach the task of verifying if a given program satisfies this guideline. For this, we will take a slightly more complicated code exhibiting the same kind of phenomenon:

```
let makecode(x)=.. in
letrec f(x,s,g)=
                 let y=first(s) in
                      output(g(x));
                      f(conc(y,x),next(s));
in
        f("",untrusted_stream,makecode);
```

In this code we have a recursive function `f` that reads from an untrusted stream, and calls a function `g`, later instantiated to `makecode`, to prepare a Javascript command based on all the input read so far. Hence, this function transforms an input stream into a sequence of Javascript commands. The stream is infinite, so the function does not terminate.

The first question is what should be the meaning of such a function. The answer of schematology is that it should be an *evaluation tree* representing the execution of the program:



This is an infinite tree obtained by interpreting the control flow, but letting all constants non-interpreted. As we can see, the evaluation tree starts with a semi-colon, followed on the left branch by a call to `makecode` and `out` functions. The rightmost path of the tree is infinite while the left branches get increasingly bigger. We have left the `makecode` function unspecified, but we can imagine that if it were given, the *mkcd* constant in the above tree would be replaced by some tree using the `validate` function.

Our programming guideline now translates into the property: on every path, between an occurrence of `out` and s there should always be an occurrence of `validate`.

The connections between the results of Büchi [70] and Rabin [248] on one side, and schematology on the other have not been immediate. Rabin's theorem says that the monadic second-order theory of a regular tree is decidable. This formulation naturally leads to a question: what happens for other trees? In particular for evaluation trees studied in schematology? In the mid-seventies Courcelle [84] has shown that evaluation trees of first-order programs can be represented by pushdown automata. In mid-eighties, coming from a completely different perspective, Muller and Schupp [222] have proved that monadic second-order theory of pushdown graphs is decidable. These two results have been put together and extended only in this century by Knapik, Niwiński, and Urzyczyn [179, 182]. They have established the decidability of the MSOL theory of evaluation trees of so-called safe recursion schemes. Finally, a decade ago, Ong [233] has shown that the safety restriction can be removed. Thus evaluation trees of a simple programming language with higher-order functions and recursion, the *λY-calculus*, have decidable MSOL theory.

5.2 Parity automata

Rabin's original proof of the decidability of the MSOL theory of the infinite binary tree inductively translates MSOL formulae to particular finite state automata that are now called Rabin automata. The major difficulty of the proof resides in the translation of negation and thus of proof that the class of languages definable by Rabin's automata is closed under complementation. The idea to use games so as to simplify Rabin's proof was already present in Büchi's work [71]. It has been successfully applied by Gurevich and Harrington [148] who were able to propose a simplified proof of Rabin's result following Büchi's lead. Probably the simplest proof of Rabin's result has been proposed by Emerson and Jutla [115]. Their proof is based on the relation between μ-calculus [43] and tree automata recognizing infinite trees. They prove that μ-calculus captures the same class of properties as MSOL on the infinite binary tree[1]. Importantly, they introduce a new acceptance condition for these automata called *parity condition*. It is now standard to use tree automata with parity conditions as a concrete representations of MSOL formulae. The interest of this formulation is that the emptiness problem of parity tree automata can be reduced to determining whether a player has a winning strategy in so-called *parity games*. These games have nice properties (see [298] for a clear and self-contained presentation): they are determined (i.e. one of the player has a winning strategy), winning strategies are particularly simple as

[1] On arbitrary structures μ-calculus is less expressive than MSOL. Janin and Walukiewicz [158] show which fragment of MSOL is equivalent to μ-caculus.

they are memoriless (i.e. a winning strategy can be described only by stating which move is to be performed by the player at each position without having to take previous moves into account). The inherent symmetry of parity condition and the determinacy of parity games give a simple proof of the closure under complementation of tree automata.

Usually tree automata are defined to accept trees constructed on a tree signature. Here we are going to use them as means of recognizing Böhm trees of closed λY-terms of atomic type over a tree signature. Not only are those trees built with a tree signature, but they may also contain leaves labeled with Ω which mark places where unproductive computations of λY-calculus occurred. Thus when defining automata that are to run on trees built on a given tree signature Σ, we also need to define their behavior when they meet Ω. This subtle distinction is of importance and has been often overlooked in the literature.

So as to simplify the notations and the exposition we only consider tree signatures with binary o nullary operators when dealing with the formal presentations. In examples, we indulge ourselves to use arbitrary arities.

Definition 47 A *finite parity tree automaton* over the signature $\Sigma = \Sigma^{(0)} \cup \Sigma^{(2)}$ (here, $\Sigma^{(0)}$ is the part of the signature with nullary operators and $\Sigma^{(2)}$ is the part with binary ones) is

$$\mathcal{A} = \langle Q, \Sigma, \delta_0 : Q \times (\Sigma_0 \cup \{\Omega\}) \to \{\mathit{ff}, \mathit{tt}\}, \delta_2 : Q \times \Sigma_2 \to \mathcal{P}(Q^2), \mathtt{rk} : Q \to \mathbb{N}\rangle$$

where Q is a finite set of states. The transition function of parity automata may be subject to the additional restriction:

$$\textbf{Ω-blind:} \quad \delta_0(q, \Omega) = \mathit{tt} \text{ for all } q \in Q.$$

Automata satisfying this restriction are called Ω-*blind*. For clarity, we use the term *insightful* to refer to automata without this restriction.

Another restriction is

$$\textbf{Ω-even}: \quad \delta_0(q, \Omega) = \mathit{tt} \text{ when } \mathtt{rk}(q) \text{ is even.}$$

A parity automaton is said *trivial* when the image of Q by $\mathtt{rk}$ is only made of even numbers.

A parity automaton is said *weak* when for every q, $a \in \Sigma_2$ and $(q_1, q_2) \in \delta_2(q, a)$, $\mathtt{rk}(q_1) \leq \mathtt{rk}(q)$ and $\mathtt{rk}(q_2) \leq \mathtt{rk}(q)$.

Weak Parity automata have been introduced by Muller *et al.* [223, 224] and capture weak Monadic Second Order Logic (wMSOL). In wMSOL, set variables can only be interpreted as finite sets over the structure.

We are now in position to define what it means for a parity automaton $\mathcal{A}$ to accept a tree. For this definition we consider that a tree T is defined as a partial function from a prefix closed subset of $\{0, 1\}^*$ to Σ that respect the arities of the labels, that is:

- if $u0$ or $u1$ is $\mathrm{dom}(T)$ in then both $u0$ and $u1$ are in $\mathrm{dom}(T)$ and $T(u)$ is a binary operator of Σ, in that case we call u a *binary node*,
- if $T(u)$ is in $\Sigma^{(0)}$ or $T(u) = \Omega$, then neither $u0$ nor $u1$ is in $\mathrm{dom}(T)$, in that case we call u a *nullary node* or a *leaf*.

Definition 48 A *run of $\mathcal{A}$ on a tree T from a state q^0* is a labelling of the nodes of T with the states of $\mathcal{A}$ such that:

- the root is labelled with q^0,
- if a node u is a leaf and the run labels u with a state q, then $\delta(q, T(u)) = tt$,
- if u is a binary node, and the run labels u with a state q, then the run labels $u0$ and $u1$ respectively with q_0 and q_1 such that $(q_0, q_1) \in \delta(q, T(u))$.

A run is *accepting* when for every infinite path of T, the labelling of the path given by the run satisfies the *parity condition*. This means that if we look at the ranks of states assigned to the nodes of the path then the maximal rank appearing infinitely often is even. A tree is *accepted by $\mathcal{A}$ from a state q^0* if there is an accepting run from q^0 on the tree.

It is well known that for every MSOL formula there is a parity automaton recognizing the set of trees that are models of the formula. The converse also holds. Let us also recall that the automata model can be extended to alternating parity automata without increasing the expressive power. Here, for simplicity of the presentation, we will work only with nondeterministic automata but the constructions we will present later apply also to alternating automata.

When an automaton is trivial, notice that the parity condition is degenerated as every state has an even rank. Thus the acceptance of trivial automata just amount to the existence of some run. In the context of verification of higher-order properties, trivial automata with have gathered considerable attention [184]. This kind of verification problems are in direct relation with some work that was conducted in the early 90's by Jensen [159] who was using intersection types as a sort of refinement type in the sense of Freeman and Pfenning [128]. This work was in direct connection with domain theory, and intersection types were used as syntactic representations of monotone models as is explained in section 2.5 using Abramsky's idea of domain in logical forms [32].

The literature on higher-order model checking is implicitly assuming the Ω-blindness condition which has as consequence to unconditionally accepts divergent computations, while insightful automata or Ω-even automata can test divergence. For technical reasons related to our constructions, we are only

going to consider Ω-even automata[2]. Using alternation, insightful automata can easily be transformed into Ω-even automata.

Definition 49 A parity automaton $\mathcal{A}$ together with one of its state q^0 *recognizes* a language of closed terms of type o:

$$L(\mathcal{A}, q^0) = \{M : M \text{ is closed term of type } o,\ BT(M) \text{ is accepted by } \mathcal{A} \text{ from } q^0\}$$

While trivial automata are concerned with safety/reachability properties which describe certain ill configurations that are to be avoided, parity automata capture properties that are behavioral and that cannot be captured by finite approximation. In particular, we have proven that monotone models exactly capture Ω-blind trivial automata [S31].

Theorem 50 *L is a set of closed $\lambda Y\Omega$-terms of type o recognized by a monotone model iff L is recognized by a boolean combination of Ω-blind trivial automata.*

As least and greatest fixpoints are computed by finite approximations of terms, the evaluation of whether a term M is in a language recognized by a trivial automaton can be done by only exploring a finite prefix of the Böhm tree of M. The model that is used to recognize the same language as a given trivial automaton $\mathcal{A}$ with states Q, is simply a GFP model where the base type is the complete lattice $(\mathcal{P}(Q), \subseteq)$ and where the constants have straightforward interpretations that simulate the transitions of the automaton. We also remark that monotone models cannot detect unproductive computation, i.e. Ω. In [S31], we show how this can be accommodated. Our proposal consists in mixing a monotone model constructed from the automaton and the model $\mathbb{2}$. Interestingly the construction requires the interaction of two different fixpoints. We will see how to generalize this construction so as to capture wMSOL in Section 5.3.

MSOL properties are more challenging than just safety and reachability properties. From the perspective of automata, we pass from the acceptance by a final state to infinitary parity acceptance conditions. From the perspective of semantics, we pass from least fixpoints to some more complicated non-extremal fixpoints. The reason for this fundamental change is that while reachability properties can be decided by looking at a prefix of the tree, behavioral properties are different since they talk about repeated occurrences of events.

The property in the first example was a safety property because its negation is a reachability property. Indeed, in order to exhibit a violation of the property it is sufficient to find an occurrence of s with an ancestor labelled `out`, and no `validate` in between.

Many behavioral properties cannot be expressed this way. For example, we can ask that every call to `makecode` uses s. If, as it would normally be the case, the evaluation tree of `makecode` is infinite, we cannot decide if s is not used in this tree just by looking at its prefixes. For similar reasons properties like:

[2]For trivial automata being insightful in equivalent to be Ω-even.

every `open(file)` is eventually followed by `close(file)` are neither safety, nor reachability properties.

There are liveness and fairness properties that exhibit even more complicated patterns. For example, we could ask that the input stream is accessed infinitely often, or in other words that, there are infinitely many `first`, and `next` calls. Going further, we can ask for a kind of productiveness property: if the input stream is accessed infinitely often then some output is produced infinitely often. This property says that the program cannot continue to read from the input stream without producing any output.

The kinds of properties from the last paragraph naturally lead to the parity acceptance condition in automata. A property "there is a path with infinitely many calls to some procedure" can be checked by an automaton that enters an accepting state each time it sees a call; assuming that the acceptance condition of the automaton is that an accepting state should appear infinitely often. The property "if infinitely many `in` then infinitely many `out` on the path" can be checked by an automaton whose states have three *ranks*: $0, 1, 2$. The automaton will normally be in a state of rank 0, but it will enter rank 1 when seeing `in`, and rank 2 when seeing `out`. Now, the acceptance condition is a *parity condition*: the biggest rank seen infinitely often is even. With the assignment of ranks we have described, this parity condition corresponds directly to the property we consider.

When considering the emptiness problem of parity automata, that is whether there is a tree which is accepted by the automaton, one naturally reduces the problem to a finite game, called parity game.

Definition 51 A parity game is a two player game, called Eve and Adam, the game is played on a graph $(V_0, V_1, E, \texttt{rk}, v, k)$ where, letting $V = V_0 \cup V_1$:

- V_0 and V_1 are disjoint,
- $E \subseteq V_0 \cup V_1$, the set of moves,
- $k \in \mathbb{N}$ and $\texttt{rk} : V \to [0, k]$,
- $v \in V$.

The elements of V_0 are Eve's positions and the elements of V_1 are Adam's positions.

A play starts in position v, the player to which the position belongs chooses a new position v' so that (v, v') is in E. This process is repeated possibly infinitely. Thus, a play $p = v_0 \dots v_i \dots$ is a possibly infinite sequence of elements of V, so that:

- $v_0 = v$,
- for every i, (v_i, v_{i+1}) is in E.

In case a play is finite the looser is the one to which the last position belongs.

To each play such as p, $\mathtt{rk}$ associates a sequence of numbers $n_0 \dots n_i \dots$ where $n_i = \mathtt{rk}(v_i)$. Eve wins the infinite play p when $\max\{l \mid \forall i \exists j \geq i, n_j = l\}$ is even. Otherwise Adam wins. In other words Eve wins an infinite play iff the maximal rank that occurs infinitely often in the play is even.

Parity games are determined: either Eve or Adam have a winning strategies. Moreover, there are memoriless winning strategies, i.e. strategies for which the player can decide which move to play by only taking into account the current position. When the parity game is finite the problem of computing a (memoriless) winning strategy is decidable and is NP $\cap$ coNP.

The reduction of parity automata to parity games is rather simple, for a parity automaton $\mathcal{A}$, we define the parity game as follows:

- V_0 is Q, the set of states of $\mathcal{A}$,
- V_1 is $Q \times Q \uplus Q \times tt$,
- E is the set of pairs:
  - $(q, (q_1, q_2))$ where there is a binary symbol a so that $(q_1, q_2) \in \delta_2(a, q)$,
  - $(q, (q, tt))$ when there is nullary symbol a so that $\delta_0(a, q) = tt$,
  - $((q_1, q_2), q_i)$ with $i \in \{1, 2\}$.

In other words, in position q, Eve tries to construct a tree that $\mathcal{A}$ accepts from state q by choosing a transition, then Adam tries to choose to pursue the play by inspecting either the left or the right part of the tree so as to contradict the parity condition.

When Eve has a winning strategy this means that she is able to construct a tree together with a run that accepts it. While when Adam has a winning strategy, this means that no matter which tree and run Eve constructs, he is able to find a branch of the run that does not verify the parity condition.

One of the lines of work we have followed when studying higher-order verification of behavioral properties was to connect it with traditional domain theoretic semantics. There are several reasons to do so. From the perspective of higher-order verification, it reduces model checking to the evaluation of programs into denotational models. It gives thus simple concepts so as to devise algorithms in a simple way. From the perspective of the communities it is interesting to have a cross-fertilization by exchanging tools and methods. For example, it asks how to construct models that compute non-extremal fixpoints. This question has been barely looked in the literature [56, 269]. On the other hand, many of the constructions and the ideas that have been developed in denotational semantics should find natural applications in higher-order verification. As an example we can site the use of sequential algorithms as a mean of accelerating verification of safety properties [125].

5.3 Wreath product and weak parity automata

We are going to see here how to construct models that recognize the same class of terms as Ω-even weak parity automata. For this, in the course of this section, we fix an Ω-even weak parity automaton $\mathcal{A} = \langle Q, \Sigma, \delta_0 : Q \times (\Sigma_0 \cup \{\Omega\}) \rightarrow \{\mathit{ff}, \mathit{tt}\}, \delta_2 : Q \times \Sigma_2 \rightarrow \mathcal{P}(Q^2), \mathtt{rk} : Q \rightarrow \mathbb{N}\rangle$. The construction of the model is presented in [S15] and is a generalization of the construction of [S31]. Here we are going to give a slightly more general presentation of the construction that is at the cross-road of denotational semantics and automata theory. For this we introduce the notion of wreath product of applicative structures. As we will use the wreath product so as to construct models of $\lambda Y \Omega$-calculus, we describe a notion wreath product for applicative structures that are made of lattices and that we thus call *lattice applicative structures*.

Definition 52 Given a lattice applicative structure $\mathcal{M} = ((\mathcal{M}_A)_{A \in \mathcal{T}(\Sigma)}, \bullet)$ and a monotone applicative structure $\mathcal{N} = (\mathcal{N}_A)_{A \in \mathcal{T}(\Sigma)}$ we construct a lattice applicative structure $\mathcal{M} \,\mathtt{w}\, \mathcal{N} = ((\mathcal{F}_A)_{A \in \mathcal{T}(\Sigma)}, \star)$. So as to define $\mathcal{M} \,\mathtt{w}\, \mathcal{N}$ we use an intermediate family of lattices $(\mathcal{G}_A)_{A \in \mathcal{T}(\Sigma)}$, the definition is as follows:

- $\mathcal{G}_o = \mathcal{N}_o$
- $\mathcal{G}_{A \rightarrow B} = \mathcal{M}_A \rightarrow_m \mathcal{G}_A \rightarrow_m \mathcal{G}_B$ where $S \rightarrow_m T$ is the lattice of monotone functions from the lattice S to to the lattice T ordered pointwise,
- $\mathcal{F}_A = \mathcal{M}_A \times \mathcal{G}_A$ which is ordered coordinatewise,
- $(f, g) \star (a, b) = (f \bullet a, g(a)(b))$.

Note that the operation $_ \,\mathtt{w}\, _$ is asymmetric not only in the treatment of its components but also on their requirements. The left argument of the operator could have been *a priori* any applicative structure. We have chosen to use lattice applicative structures because it gives a nicer connection with intersection types (though we don't give the details here on intersection types for wMSOL, they can be found in [S15]). For the right argument, we require that it is a monotone applicative structure. The reason for this is that we wish to use wreath products of applicative structures so as to construct models of $\lambda Y \Omega$-calculus which requires that we are able to define the semantics of fixpoints and the lattice structure allows us to prove the existence of such fixpoints. Moreover, the asymmetry in requirements allows us to obtain lattice applicative structures as results of wreath products and thus to iterate the construction. Again,we could have chosen weaker requirements but they suit our needs so as to construct models capturing wMSOL.

The idea of the wreath product is to create a dependence of the evaluation in the second applicative structure on the evaluation in the first applicative structure. In the context of finite state automata, when we want to build an automaton whose runs are built on the runs of another automaton on the input string, this amounts at the level of syntactic monoids to construct their wreath product.

The definition of weak parity automata implies that in every of their runs the ranks of states are decreasing along each branch of the tree. The idea of using wreath products then comes naturally. If we know for each node of a Böhm tree of a term M in which are the states of rank 0 from which the automaton has an accepting run, then we can deduce from which state of rank 1 the automaton has an accepting run in each node and so on. The idea is thus to construct a λ-model by induction on the ranks and then construct a model for the next rank using wreath product.

We let Q_k be the set $\{q \in Q \mid \mathtt{rk}(q) = k\}$ and $Q_{\leq k}$ be the set $\{q \in Q \mid \mathtt{rk}(q) \leq k\}$. We write $\mathcal{A}_k$ for the automaton obtained from $\mathcal{A}$ by restriction to the states in $Q_{\leq k}$.

Note that $\mathcal{A}_0$ is a trivial automaton. Thus from Theorem 50, we know that there is a GFP model $\mathbb{M}_0 = ((\mathcal{M}_{0,A})_{A\in\mathcal{T}(\Sigma)}, [\![\cdot,\cdot]\!]_0)$ so that $\mathcal{M}_{0,o} = \mathcal{P}(Q_{\leq 0})$ and so that, for a closed term M of type o, $q \in [\![M]\!]_0$ iff $\mathcal{A}_0$ has a run on $BT(M)$ starting with state q. Now supposing that $\mathbb{M}_k = ((\mathcal{M}_{k,A})_{A\in\mathcal{T}(\Sigma)}, [\![\cdot,\cdot]\!]_k)$ is a lattice model so that $\mathcal{M}_{k,o} = \mathcal{P}(Q_0) \times \ldots \mathcal{P}(Q_k)$ and so that, for a closed term M of type o, $[\![M]\!]_k = (R_0, \ldots, R_k)$ and $q \in R_i$ iff $\mathcal{A}_k$ has an accepting run on $BT(M)$ starting with state q. Then we define $\mathbb{M}_{k+1}$ by taking as applicative structure$(\mathcal{M}_{k+1,A})_{A\in\mathcal{T}(\Sigma)} = (\mathcal{M}_{k,A})_{A\in\mathcal{T}(\Sigma)} \,\mathtt{w}\, \mathcal{R}_{k+1}$ where $\mathcal{R}_{k+1}$ is the monotone applicative structure generated by $\mathcal{P}(Q_{k+1})$. We now need to define the value of constants, we start by the definition of the fixpoint. For this, following the definition we have given of the wreath product, we use an intermediate family of lattices, $(\mathcal{G}_{k+1,A})_{A\in\mathcal{T}(\Sigma)}$, that is defined as expected. The value $[\![Y^A, \emptyset]\!]_{k+1}$ needs to be in

$$\mathcal{M}_{k+1,(A\to A)\to A} = \mathcal{M}_{k,(A\to A)\to A} \times \mathcal{G}_{k+1,(A\to A)\to A} \ ,$$

Thus the value of $[\![Y^A, \emptyset]\!]_{k+1}$ is a pair (f, g) where f is in $\mathcal{M}_{k,(A\to A)\to A}$. As the idea of the wreath product is to compute the value of the term in $\mathbb{M}_k$ and use this information to compute its value in $\mathbb{M}_{k+1}$, we need to take f as $[\![Y^A, \emptyset]\!]_k$. Let's see now which value of g we need. We first unfold the definition of the wreath product and we have that g should be in:

$$\mathcal{G}_{k+1,(A\to A)\to A} = \mathcal{M}_{k,A\to A} \to_m (\mathcal{M}_{k,A} \to_m \mathcal{G}_{k+1,A} \to_m \mathcal{G}_{k+1,A}) \to_m \mathcal{G}_{k+1,A} \ .$$

So, given x, that represents the semantics of a program in $\mathbb{M}_k$, and y that represents the semantics of that program in $\mathbb{M}_{k+1}$, we need to compute a fixpoint. By applying y to $[\![Y^A, \emptyset]\!]_k(x)$, we obtain a value in $\mathcal{G}_{k+1,A} \to_m \mathcal{G}_{k+1,A}$, for which we can compute either a least or a greatest fixpoint finally obtaining a value in $\mathcal{G}_{k+1,A}$. Moreover, we may notice that indeed, $[\![Y^A, \emptyset]\!]_k(x)$ represents the value of the fixpoint of the program that is denoted by x in $\mathbb{M}_k$ which is precisely the value on which we want the evaluation of the fixpoint to depend on. In a nutshell, we obtain that[3]:

- $[\![Y^A, \emptyset]\!]_{k+1} = ([\![Y^A, \emptyset]\!]_k, \boldsymbol{\lambda}(x, y).\mu t.y([\![Y^A, \emptyset]\!]_k(x))\, t)$ when $k + 1$ is odd,

[3]We write $\mu t.f\, t$ and $\nu t.f\, t$ respectively for the least and greatest fixpoint of f.

- $[\![Y^A,\emptyset]\!]_{k+1} = ([\![Y^A,\emptyset]\!]_k, \boldsymbol{\lambda}(x,y).\nu t.y([\![Y^A,\emptyset]\!]_k(x))\, t)$ when $k+1$ is even.

Computing a least fixpoint in the case of an odd value amounts to only authorize finitely many repetitions of states with odd ranks in an accepting run. Dually, computing a greatest fixpoint in the case of an even rank amounts to authorize possibly infinite repetitions of states with even ranks in accepting runs. Notice that with this definition a non-converging computation, Ω, is interpreted with every states with even rank and no state of odd rank which mirrors the Ω-even convention.

The other constants are either binary or nullary, by convention. With the definition of wreath product, we have that a binary constant is interpreted in

$$Q_0 \times \cdots \times Q_k \times Q_{k+1} \to_m Q_0 \times \cdots \times Q_k \times Q_{k+1} \to_m Q_0 \times \cdots \times Q_k \times Q_{k+1}$$

and a nullary constant in

$$Q_0 \times \cdots \times Q_k \times Q_{k+1} \; .$$

We simply define their intuitive interpretation that mimics the transitions of the automaton as follows:

- when a is binary, $[\![a,\emptyset]\!]_{k+1}(R_0,\ldots,R_{k+1})(R'_0,\ldots,R'_{k+1}) = (P_0,\ldots,P_{k+1})$ iff:
  - $[\![a,\emptyset]\!]_k(R_0,\ldots,R_k)(R'_0,\ldots,R'_k) = (P_0,\ldots,P_k)$,
  - $P_{k+1} = \{q \in Q_{k+1} \mid \exists q_1 \in R_0 \cup \cdots \cup R_{k+1}. \exists q_2 \in R'_0 \cup \cdots \cup R'_{k+1}. (q_1,q_2) \in \delta_2(a,q)\}$,
- when a is nullary, $[\![a,\emptyset]\!]_{k+1} = (R_0,\ldots,R_{k+1})$ if $[\![a,\emptyset]\!]_k = (R_0,\ldots,R_k)$ and $R_{k+1} = \{q \in Q_{k+1} \mid \delta_0(a,q) = tt\}$.

A simple induction on k shows the property we have announced:

Theorem 53 *For every closed term M of type o, if $[\![M,\emptyset]\!]_k = (R_1,\ldots R_k)$, then $q \in R_i$ iff $\mathtt{rk}(q) = i$ and $\mathcal{A}_k$ has an accepting run from q on $BT(M)$.*

Another interesting fact about the proof of that theorem is that it uses usual techniques for proving adequacy theorems in simply typed λ-calculus. It can be therefore considered as a purely semantic theorem.

Moreover the symmetries induced by the construction give a nice syntactic representation of the model with intersection types that give two dual systems for reasoning about programs. One of the type system allows one to prove that the automaton accepts the Böhm tree of a term from a given state, while the other system allows one to prove that the automaton does not accept the Böhm tree from a given state (see [S15]).

5.4 λY-calculus and abstract machines

The construction of a model for wMSOL we have just seen is mainly based on purely model theoretic methods. The proof presented in [S15] follow the usual adequacy proofs that have a similar form as Tait's proof that proves the strong normalization of simply typed λ-calculus. A good example of such a proof can be found in [246] concerning the adequacy of Scott models for PCF. When we want to turn to the full MSOL, some difficulties arise from the fact that the hierarchy of ranks is no longer connected to the transition functions of the automata. And all the methods that have been proposed to prove the decidability of the MSOL theory of the Böhm tree of a λY-term M goes through an argument tightly linked to an evaluation mechanism of the λY-calculus. The first proof proposed by Ong [233] is based on Games Semantics [157] which is used to combine a parity game with the evaluation through the notion of *traversals*. A second proof has been proposed by means Collapsible Pushdown Automata (CPDA) [149], abstract machines that are performing the computation of Böhm trees.

This second proof is of importance because it relates Ong's theorem with former research on higher-order computation and automata that was carried out using tools coming from formal language theory. The interest about the verification of higher-order systems can be traced back to extensions Rabin theorem for classes of trees of increasing complexities. First Muller *et al.* [222] proved that the MSOL theory of the graphs generated by pushdown systems had a decidable MSOL theory. Then Courcelle and Walukiewicz clarified the relation between the MSOL theories of objects generated by machines (in particular pushdown automata) and the MSOL theories of those machines by a series of so-called transfer theorems [88, 93, 290]. Knapik *et al.* [179] have extended the results to hyperalgebraic trees and then to tree generated by higher-order *safe* schemes by means of higher-order pushdown automata [182]. These results raised the question of whether the safety property was a genuine restriction for the expressivity of higher-order schemes. They also make it clear that Damm's results [102, 103] were implicitly assuming the restriction of safety. This problem has been subsequently solved by Parys [239] who showed that, indeed, safety is hampering the expressiveness of higher-order computation. These results also asked how to translate unsafe schemes into the traditional framework of pushdown automata. A first attempt has been proposed in the term of *panic automata* by Knapik *et al.* [181] which could capture unsafe higher-order computation up to the third order. CPDA provide the right generalization of panic automata to all orders. These automata are an extension of higher-order pushdown automata introduced by [211] and which are using stacks of stacks of stacks... up to a fixed depths. The generalization consists in linking symbols in the higher-order stack to lower parts of the stack so as to restore some evaluation environment via an operation called *collapse*.

A third proof in terms of taylor-made intersection types has been proposed by Kobayashi and Ong [185]. Here again, the most technical part of the proof is about proving a relationship between typing properties of a term and

its Böhm tree which is similar to usual conversion theorems for intersection types. Nonetheless, it is unclear whether this type system can accommodate λ-abstraction or fixpoints.

Finally we proposed [S28] a proof based on Krivine machines and models of λ-calculus. From the very beginning the idea was to generalize transfer theorems from [88, 93, 290] and, if possible, to formulate it as a property of preservation of recognizability by inverse homomorphism of some notion of recognizability that would be similar to Theorem 21. This goal naturally leads to setting Ong theorem is the framework of recognizability. We have made precise account of translations between CPDAs, higher-order schemes, and λY-calculus in [S30]. In the proofs of equivalence we show between λY-calculus and CPDAs, the Krivine machine plays a nice role as it allows us to give very simple invariants so as to show that head-reduction and execution of CPDAs compute the same trees.

All the proofs of Ong's theorem follow a similar reduction of the *infinite* parity game induced by the parity automaton on the Böhm tree of a term to a *finite* parity game played on the syntax of the program. The computational mechanism, be it game semantics traversals, CPDA, schemes unfolding, or Krivine machine, is meant to prove that the two games are equivalent. The main advantage of Krivine machine is to make the link between the computation and the syntax of the program rather direct and thus the proof of equivalence between the games rather transparent. In the case of games semantics, the details of the theorem concerning traversals are particularly tedious. In the case of CPDAs, the method consists first in showing that they generate (exactly) the class of trees generated by higher-order schemes and then work on the syntax of CPDAs providing stack invariants. Here, using Krivine machine presents another advantage as all its configurations are built with λY-terms, and invariants may thus naturally be expressed in terms of models. It is then immediate that they are indeed invariants. The use of Krivine machines makes a nice bridge between automata theory and λ-calculus allowing us to simplify and generalize the proof in two directions: first by providing the expected generalizations of the transfer theorems, second by setting Ong's theorem in the context of recognizability.

We now give the definition and basic properties of Krivine machines as we are going to use them in the next section so as to construct a finite λ-model that recognizes MSOL properties.

A Krivine machine [190], is an abstract machine that computes the weak head normal form of a term. For this it uses explicit substitutions, called *environments*. Environments are functions assigning *closures* to variables, and closures themselves are pairs consisting of a term and an environment. This mutually recursive definition is schematically represented by the grammar:

$$C ::= (M, \rho) \qquad \rho ::= \emptyset \mid \rho[x \mapsto C]\ .$$

As in this grammar, we will use $\emptyset$ for the empty environment. The notation $\rho[x \mapsto C]$ represents the environment which associates the same closure as

ρ to variables except for the variable x that it maps to C. We require that in a closure (M, ρ), the environment is defined for every free variable of M. Intuitively such a closure denotes a closed λ-term: it is obtained by substituting for every free variable x of M the λ-term denoted by the closure $\rho(x)$. When $\rho(x) = C$, we say that ρ *binds* C *to* x. Given a closure (M, ρ), we say that it has type A when M has type A.

A configuration of the Krivine machine is a triple (M, ρ, S), where M is a term, ρ is an *environment*, and S is a *stack*. A stack is a sequence of closures. By convention the topmost element of the stack is on the left. The empty stack is denoted by ε. The rules of the Krivine machine are as follows:

$$
\begin{aligned}
(\lambda x.M, \rho, (N, \rho')S) &\to (M, \rho[x \mapsto (N, \rho')], S) \\
(MN, \rho, S) &\to (M, \rho, (N, \rho)S) \\
(Yx.M, \rho, S) &\to (M, \rho[x \mapsto (Yx.M, \rho)], S) \\
(x, \rho, S) &\to (M, \rho', S) && \text{when } \rho(x) \text{ is defined} \\
& && \text{and equal to } (M, \rho')
\end{aligned}
$$

Note that the machine is deterministic. We will write $(M, \rho, S) \to^* (M', \rho', S')$ to say that the Krivine machine goes in some finite number of steps from configuration (M, ρ, S) to (M', ρ', S').

The intuitions behind the rules are rather straightforward. The first rule says that in order to evaluate an abstraction $\lambda x.M$, we should look for the argument at the top of the stack, then we bind this argument to x, and calculate the value of M. To evaluate an application MN we create a closure out of N and the current environment so as to be able to evaluate N correctly when necessary and put that closure on the stack; then we continue to evaluate M. The rule for $Yx.M$ simply amounts to bind the variable x in the environment to the current closure of $Yx.M$ and to calculate M. Finally, the rule for variables says that we should take the value of the variable from the environment and evaluate it; the value is not just a term but a closure: a term with an environment giving the right meanings to the free variables of the term.

We will be only interested in configurations accessible from $(M, \emptyset, \varepsilon)$ for some closed term M of type o. Every such configuration (N, ρ, S) enjoys very strong typing invariants summarized in the following definition and lemma.

Definition 54 Given M a term of type o, an environment ρ is M*-correct* when for every variable x^A, if $\rho(x^A)$ is defined, then $\rho(x^A)$ is a closure (N, ρ') of type A that is M*-correct*, meaning that:

1. N is a substerm of M,
2. ρ' is M-correct, and
3. for every variable y, if $y \in FV(N)$, then $\rho'(y)$ is defined.

A configuration of a Krivine machine (N, ρ, S) is M*-correct* when:

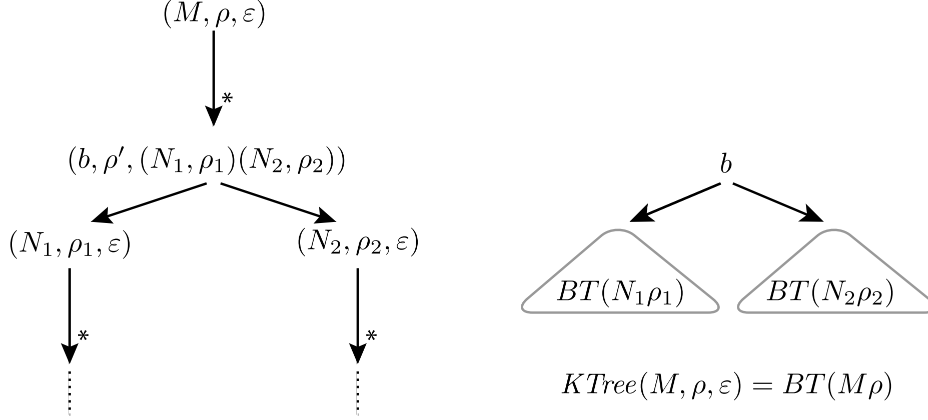


Figure 5.1: Computation of Krivine machine and the resulting $KTree(M, \rho, \varepsilon)$.

1. (N, ρ) is an M-correct closure, and
2. if N has type $A_1 \to \cdots \to A_n \to o$, then $S = C_1 \ldots C_1$ and the closures C_i are M-correct and have types A_i.

Lemma 55 If M is a simply typed term of type o, given two configurations (N_1, ρ_1, S_1) and (N_2, ρ_2, S_2) so that $(N_1, \rho_1, S_1) \to (N_2, \rho_2, S_2)$, if (N_1, ρ_1, S_1) is M-correct, then (N_2, ρ_2, S_2) is also M-correct.

This lemma is easy to prove and is somehow providing a nice link between computations carried out by Krivine machines and the original term M. Technically, it plays a key role in reducing a parity game on the computation of Böhm tree of M and parity games on M itself. It says that the typing is preserved throughout computation and also that the code being evaluated is always a combination of code from an initial program.

Let us now explain how to use Krivine machines to calculate the Böhm tree of a term (cf. Figure 5.1). For this we define an auxiliary notion of a tree constructed from a configuration (M, ρ, ε) where M is a term of type o over a tree signature. (Observe that the stack should be empty when M is of type o.) We let $KTree(M, \rho, \varepsilon)$ be the tree consisting only of a root labeled with Ω^o if the computation of the Krivine machine from (M, ρ, ε) does not terminate. If it terminates then $(M, \rho, \varepsilon) \to^* (b, \rho', (N_1, \rho_1) \ldots (N_k, \rho_k))$, for some constant b. In this situation $KTree(M, \rho, \varepsilon)$ has b in the root and for every $i = 1, \ldots, k$ it has a subtree $KTree(N_i, \rho_i, \varepsilon)$. Due to typing invariants and since we are working with a tree signature, by our convention we have that the constant b must have type $o^k \to o$ with $k \in \{0, 2\}$. In consequence all terms N_i have type o.

Definition 56 For a closed term M of type o we let $KTree(M)$ be $KTree(M, \emptyset, \varepsilon)$ where $\emptyset$ is the empty environment, and ε is the empty stack.

The next lemma says what $KTree(M)$ is. The proof is immediate from the fact that Krivine machine performs weak head reduction.

Lemma 57 For every closed term M of type o over a tree signature: $KTree(M) = BT(M)$.

This last lemma is the last element that allows us to relate the two aforementioned parity games.

Example 58 We give a very simple illustration of the computation of a Böhm tree by the Krivine machine. We compute the Böhm tree of the term M defined as follows:

$$M = \big(\lambda y.\ (\lambda gx.\ Yf.\ gx)ay\big)\ e$$

For readability we adopt the following shorthands:

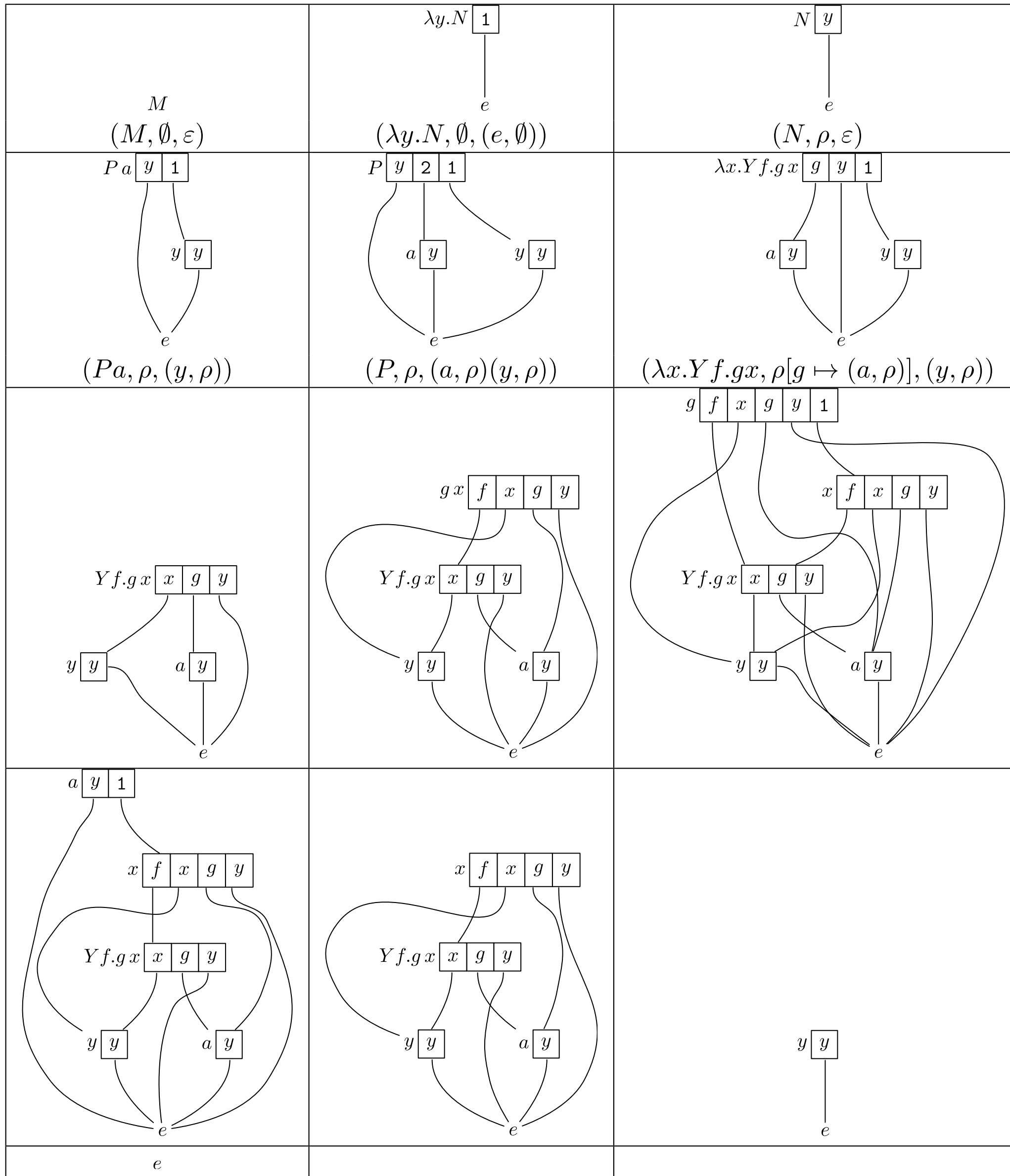
$$M = (\lambda y.N)e, \quad N = Pay, \quad P = \lambda gx.Yf.gx$$

We take this example because it illustrates all the reduction rules of the Krivine machine. It also gives us the opportunity to introduce a graphical representation of configurations of the Krivine machine. The reduction sequence is presented in Figure 5.2 (as expected, the order of execution is represented from left to right and top to bottom).

In the pictures of Figure 5.2, a closure (Q, ρ) is represented by a node labeled by Q followed to the right by boxes containing the variables bound in ρ. Each variable-box is linked to the closure it is bound to. This closure is drawn lower in the graph. When there are no such box on the right, it means that the environment is empty. Finally a configuration $(Q, \rho, C_n \dots C_1)$ is represented similarly to a closure, it is represented by a node labeled Q followed by variable-boxes but also by numbered boxes, the box numbered i is linked to the representation of the closure C_i. So as to make the representation sufficiently compact to fit in the paper, we allowed ourselves to make different variables point to the same closure. This graphical sharing is just an artifact of the presentation and even though implementations of the Krivine machines perform some sharing, it is not in general maximal as in our representation.

To make it clear how to interpret those graphical representations as configurations, we have written the six first configurations below their representations.

The ten first configurations correspond to the computation of the label of the root of the Böhm tree. Indeed the tenth configuration is of the form $(a, \rho, (x, \rho'))$. This tells us that the root of $KTree(M, \emptyset, \varepsilon)$ is labeled by a. The eleventh configuration, (x, ρ', ε), starts the computation of the daughter of the root that is simply given by the thirteenth configuration and is e. The Böhm tree of M is therefore $\begin{array}{c} a \\ | \\ e \end{array}$.



$M = (\lambda y.N)e$, $N = Pay$, $P = \lambda gx.Yf.gx$ and
ρ is the environment such that $\rho(y) = (e, \emptyset)$.

Figure 5.2: Computation of the Krivine machine from the configuration $(M, \emptyset, \varepsilon)$

5.5 A λY-model for parity automata

Before we have been able to construct a model, we reproved Ong's theorem using Krivine machines [S14, S29]. This has put us in a position to construct a model. Nevertheless, the task revealed more difficult than we expected. Intuitions coming from the exponential construction in linear logic have played an essential role. Linear logic constructions have been at the heart of the work of Grellois and Melliès [141, 143, 142]. In the course of the construction of the model, we will see where these ideas are used. A particularity of our construction is that it works for Ω-even automata, while the one of Grellois and Melliès works for Ω-blind automata. This problem could be overcome by combining their construction the model $\mathbb{2}$ by means of wreath product. For the moment, it is unclear how to refine our construction into a model of linear logic. Not only the construction of a finite model gives Ong's theorem as a corollary, but it also sets higher-order model checking within the framework of recognizability.

Let us fix an Ω-even parity automaton $\mathcal{A}$ for which assume that the maximal rank it assigns to a state is m. We wish to construct a λ-model so that for every closed term M of type o, the semantics of M is precisely the set of states from which $\mathcal{A}$ accepts $BT(M)$. The model cannot be solely built with the states of $\mathcal{A}$, we need a mechanism that allows us to keep track of ranks that appear in runs of $\mathcal{A}$ so as to check the parity condition. The evaluation in models typically works bottom-up: one computes the values of subterms and composes those values so as to obtain the value of larger subterms. A problem is that parity automata traverse trees in a top-down manner. The solution we have adopted is to have an asymmetric treatment of arguments and of results. Arguments are meant to represent occurrences of variables. They need to convey two kinds of information: the states involved in runs (the bottom-up information) and some contextual information (the top-down information) about ranks met on the path from the root of the tree to where they are used. For results, we only wish to know from which states there is an accepting run. Such an asymmetry raises a difficulty when dealing with application. Indeed, in that case, what used to be the root of the argument term may not be the root of the whole resulting term. So we need to update the contextual information related to free variables when we perform application. For this we will use some operators in the model that are inspired from the exponential in linear logic. In the case of the relational model of linear logic a similar problem arises. Indeed, in an application, the number of times a free variable is used in the computation of the value of an argument term needs to be multiplied by the number of times this argument is used in the functional part of the application. This phenomenon exhibits a similar tension between bottom-up and top-down information.

We thus define two families of lattices indexed by types $(\mathcal{S}_A)_{A\in\mathcal{T}(\Sigma)}$ (for accepting States) and $(\mathcal{R}_A)_{A\in\mathcal{T}(\Sigma)}$ (for Residuals) where for $A = o$, we let

$$\mathcal{S}_o = \mathcal{P}(Q) \qquad \text{and} \qquad \mathcal{R}_o^{ms} = \mathcal{P}(\{(q,r) : q \in Q, rk(q) \leq r \leq m\})$$

and these sets are ordered by inclusion. We also define the following operation: for $h \in \mathcal{R}_o$, and $r \in [m]$ we let

$$h\lfloor_r = \{(q,i) \in h : r \leq i\} \cup \{(q,j) : (q,r) \in h, rk(q) \leq j \leq r\}$$

This operation updates the ranks associated to arguments when a term is used as the argument of another term.

For $f \in \mathcal{S}_o$ and $g \in \mathcal{R}_o^{ms}$ we define another operation $(\cdot)\Downarrow_q$ for every $q \in Q$:

$$g\Downarrow_q = \{r : (q,r) \in g\}, \qquad f\Downarrow_q = f \cap \{q\}$$

With this we are ready to define objects for higher types. The set $\mathcal{S}_{A\to B}$ is the set of all monotone functions in $\mathcal{R}_A^{ms} \to_m \mathcal{S}_B$ satisfying the stratification condition:

$$\forall g \in \mathcal{R}_A^{ms}.\ \forall q \in Q.\ (f(g))\Downarrow_q = (f(g\lfloor_{rk(q)}))\Downarrow_q \qquad (\texttt{strat})$$

The set $\mathcal{R}_{A\to B}^{ms}$ is the set of all monotone functions in $\mathcal{R}_A^{ms} \to_m \mathcal{R}_B^{ms}$ satisfying the same (`strat`) condition. The orders in $\mathcal{S}_{A\to B}$ and $\mathcal{R}_{A\to B}^{ms}$ are the pointwise order.

For f in $\mathcal{S}_{A\to B}$ and $g \in \mathcal{R}_{A\to B}^{ms}$ we extend the operators we have defined above as follows:

$$f\Downarrow_q(h) = (f(h))\Downarrow_q \quad \text{and} \quad g\Downarrow_q(h) = (g(h))\Downarrow_q \quad \text{and} \quad g\lfloor_r(h) = (g(h))\lfloor_r\ .$$

Remark: The above definitions are covariant and they become more intuitive when we consider types written as $A_1 \to \cdots \to A_l \to o$, or in an abbreviated form as $\vec{A} \to o$. In this case we have:

$$\mathcal{S}_{\vec{A}\to o} = \mathcal{R}_{A_1}^{ms} \to \cdots \to \mathcal{R}_{A_l}^{ms} \to \mathcal{S}_o \quad \mathcal{R}_{\vec{A}\to o} = \mathcal{R}_{A_1}^{ms} \to \cdots \to \mathcal{R}_{A_l}^{ms} \to \mathcal{R}_o^{ms}$$
$$g\Downarrow_q(\vec{h}) = (g(\vec{h}))\Downarrow_q \qquad g\lfloor_r(\vec{h}) = (g(\vec{h}))\lfloor_r$$

where $\vec{h}$ is vector of elements from $\mathcal{R}_{A_1}^{ms} \times \cdots \times \mathcal{R}_{A_i}^{ms}$, and operations $\Downarrow_q$, $\lfloor_r$ are applied only to elements from $\mathcal{S}_o$ or $\mathcal{R}_o^{ms}$, depending on whether g is from $\mathcal{S}_{\vec{A}\to o}$ or $\mathcal{R}_{\vec{A}\to o}^{ms}$.

Before we define the interpretation of terms, we observe several properties of the domains and of the operations we have introduced.

Lemma 59 For every type $B = A_1 \to \cdots \to A_l \to o$, $g \in \mathcal{R}_B^{ms}$, $\vec{h}$ in $\mathcal{R}_{A_1}^{ms} \times \cdots \times \mathcal{R}_{A_l}^{ms}$ and $r, r_1, r_2 \in [m]$:

- $(g\lfloor_{r_1})\lfloor_{r_2} = g\lfloor_{\max(r_1,r_2)}$;
- $(q, rk(q)) \in g\lfloor_r(\vec{h})$ iff $(q, \max(rk(q), r)) \in g(\vec{h})$.

Here we remark that indeed the operation $(\cdot)\lfloor_r$ can compute maximal values of ranks on path of runs and it thus plays a central role in the definition of the application in the model.

Lemma 60 For every type A, both $\mathcal{S}_A$ and $\mathcal{R}^{ms}_A$ are finite complete lattices.

Lemma 61 For every A, if g_1, g_2 are in $\mathcal{R}^{ms}_A$, then $(g_1 \vee g_2){\downharpoonleft_k} = g_1{\downharpoonleft_k} \vee g_2{\downharpoonleft_k}$ and $(g_1 \wedge g_2){\downharpoonleft_k} = g_1{\downharpoonleft_k} \wedge g_2{\downharpoonleft_k}$.

We now define other operations $(\cdot)^\partial$ and $(\cdot)\cdot r$ that follow the same covariant pattern as $(\cdot)\Downarrow_q$ and $(\cdot){\downharpoonleft_r}$.

$$\begin{aligned} g^\partial =& \{q : (q, rk(q)) \in g\} & f \cdot r =& \{(q, r) : q \in f, rk(q) \leq r\} & \text{if } A = o \\ g^\partial(h) =& (g(h))^\partial & (f \cdot r)(h) =& (f(h)) \cdot r & \text{if } A = B \to C \end{aligned}$$

Thus $(\cdot)^\partial$ converts an element of $\mathcal{R}_A$ to an element of $\mathcal{D}_A$, and $(\cdot) \cdot r$ does the opposite.

Notation: we abbreviate $g{\downharpoonleft_{rk(q)}}$ by $g{\downharpoonleft_q}$.

These operations are well-behaved in the sense of the following lemma.

Lemma 62 For every type A, every $f \in \mathcal{D}_A$, $g \in \mathcal{R}^{ms}_A$, and $r \in [m]$, we have

$$f \cdot r \in \mathcal{R}^{ms}_A \qquad g{\downharpoonleft_r} \in \mathcal{R}^{ms}_A \qquad g^\partial \in \mathcal{D}_A$$

A *valuation* v is a function assigning values to variables, such that a variable of a type A is assigned an element of $\mathcal{R}^{ms}_A$. We write $v{\downharpoonleft_r}$ for the valuation so that $v{\downharpoonleft_r}(x) = v(x){\downharpoonleft_r}$

The semantics of a term M of a type A, under a given valuation v is denoted $[\![M, v]\!]$. It is an element of $\mathcal{S}_A$ provided v is defined for all free variables of M. The semantics is defined by induction on the structure of M:

$$\begin{aligned} [\![x, v]\!] =& (v(x))^\partial \\ [\![a, v]\!]h_0h_1 =& \{q : \exists_{(q_0,q_1)\in\delta(q,a)}.\ q_i \in (h_i{\downharpoonleft_{rk(q)}})^\partial \quad \text{for i=0,1}\} \\ [\![\lambda x.M, v]\!]h =& [\![M, v[h \leftarrow x]]\!] \\ [\![MN, v]\!] =& [\![M, v]\!]\langle\!\langle N, v\rangle\!\rangle \quad \text{where } \langle\!\langle N, v\rangle\!\rangle = \bigvee_{r=0}^{m} \big([\![N, v{\downharpoonleft_r}]\!] \cdot r\big) \\ [\![Y^A, v]\!]h =& \text{fix}^A(h, 0) \quad \text{where for } l = 0, \dots, m \text{ we define} \\ \text{fix}^A(h, l) =& \sigma f_l. \dots \mu f_1.\nu f_0.\ (h{\downharpoonleft_l})^\partial\big(\bigvee_{i=0}^{l} f_i \cdot i \vee \bigvee_{i=l+1}^{m} \text{fix}^A(h, i) \cdot i\big) \end{aligned}$$

In this definition σ denotes the least fixpoint when l is odd and the greatest fixpoint when l is even.

Let us explain the intuitions behind this interpretations of terms. The main specificity of the model is the treatment it does of variables. An element of f of $\mathcal{R}^{ms}_A$ can always be decomposed $f_0, \dots, f_m$ of $\mathcal{D}_A$ so that $f = f_0 \cdot 0 \vee \dots \vee f_m \cdot m$.

Here, the function f_i represents how f is to be used in a context of a run where i is the maximal rank used from the root of the run to that use of f. This explains why the semantics of variables is defined by $[\![x, \upsilon]\!] = \upsilon(x)^{\partial}$ as the maximal rank met in a run starting with the state q on a one node path is $\mathtt{rk}(q)$.

A question is why do we actually need to take this information into account. An example may give some insight. If we consider the automaton running on trees built with two binary operations a, whose states are $\{q_1, q_2, \top\}$, the rank of q_1 is 1, the rank of q_2 and $\top$ is 2 and whose transition function δ is defined by:

- $\delta(a, q_i) = \{(q_2, \top), (\top, q_2)\}$,
- $\delta(b, q_i) = \{(q_1, \top), (\top, q_1)\}$,
- $\delta(a, \top) = \delta(b, \top) = \{(\top, \top)\}$.

This automaton recognizes from the state q_1 the trees that have a path with infinitely many occurrences of a. If we do not take the parity information into account for the arguments, this would amount to interpret terms in the monotone applicative structure generated by $\mathcal{P}(\{q_1, q_2, \top\})$ and where a and b are interpreted as the monotone functions defined by:

- $q_1, q_2 \in [\![a]\!](Q_1, Q_2)$ iff $q_2 \in Q_1$ and $\top \in Q_2$, or $\top \in Q_1$ and $q_2 \in Q_2$,
- $\top \in [\![a]\!](Q_1, Q_2)$ iff $\top$ is in Q_1 and in Q_2
- $q_1, q_2 \in [\![b]\!](Q_1, Q_2)$ iff $q_1 \in Q_1$ and $\top \in Q_2$, or $\top \in Q_1$ iff $q_2 \in Q_2$,
- $\top \in [\![b]\!](Q_1, Q_2)$ iff $\top$ is in Q_1 and in Q_2.

Then the terms:

- $M_1 = \lambda x.b\,x\,x$ and,
- $M_2 = \lambda x.b\,x(a(b\,x\,x)(b\,x\,x))$,

have the same denotation, the functions f defined by

$$f = (\top \mapsto \top \mapsto \top) \vee \{q_1, \top\} \mapsto \{q_1, \top\} \mapsto \{q_1, q_2\}$$

Thus, no matter how we interpret the fixpoint in that structure, the meaning of $Y M_1$ must be the same as the meaning of $Y M_2$. Now if we interpret M_1 and M_2 in the model that we have proposed, we obtain that the meaning of M_1 is:

$$\begin{array}{rcl} f_1 & = & (\top, 2) \mapsto (\top, 2) \mapsto \top \vee \\ & & \{(q_1, 1), (\top, 2)\} \mapsto \{(q_1, 1), (\top, 2)\} \mapsto q_1 \vee \\ & & \{(q_1, 2), (\top, 2)\} \mapsto \{(q_1, 2), (\top, 2)\} \mapsto \top \end{array}$$

and the meaning of M_2 is:

$$\begin{array}{rcl} f_2 & = & (\top, 2) \mapsto (\top, 2) \mapsto \top \vee \\ & & \{(q_1, 2), (\top, 2)\} \mapsto \{(q_1, 2), (\top, 2)\} \mapsto q_1 \vee \\ & & \{(q_1, 2), (\top, 2)\} \mapsto \{(q_1, 2), (\top, 2)\} \mapsto \top \end{array}$$

The fact that $f_2 \geq \{(q_1,2),(\top,2)\} \mapsto \{(q_1,2),(\top,2)\} \mapsto q_1$, marks that fact that in a run starting from the state q_1 at the root of M_2 reaches:

- an occurrence of the argument x in the state q_1 having met a the maximal rank 2 on the way, and

- an occurrence of the argument x in the state $\top$ having met a maximal rank 2 on the way.

This is in sharp contrast with the semantic of M_1 where we have

$$f_1 \geq \{(q_1,1),(\top,2)\} \mapsto \{(q_1,1),(\top,2)\} \mapsto q_1 \ .$$

This difference is due to the occurrence of a in M_2 and its absence in M_1. Now, when computing the meaning of YM_2, we will obtain $q \in [\![YM_2,\emptyset]\!]$ while we will have $q_1 \notin [\![YM_1,\emptyset]\!]$, as the fixpoint will take advantage of the knowledge that in M_2 the iteration is building a run in which in all path the maximal rank repeated infinitely often is 2 while it is not the case in M_1.

The main technicalities of the model come from the necessity to maintain compositionally these marks of ranks in the model. In particular, so as to define this we need to treat free variables in a different way from bound variables. Indeed, when we take the term MN, the semantics of N accounts for the maximal ranks seen on runs from the *root*[4] of N to the occurrences of the free variables, this information need to be updated as now the semantics of MN needs to account for the maximal ranks seen on runs from the root of MN to the occurrences of the free variables. This means that in the semantics of MN the treatment of the free variables that are in N need to be updated. The semantics of M assigns some ranks to describe the context in which its argument is to be used. For a variable free in N, so as to update its contextual information, it suffices to take the max of the ranks that describe its uses in N and of the ranks in M that describe the uses of its arguments.

Technically, this is achieved by the action on valuations with the operation $(\cdot)\lfloor_r$ in the application. The value of $[\![N, \upsilon\lfloor_r]\!]$ is to be understood as the semantic of N in when used after having seen r as maximal rank. Notice that when N is closed, we have $[\![N,\upsilon\lfloor_r]\!] = [\![N,\upsilon]\!]$ and that this update only concerns free variable. The construction of $\langle\!\langle N,\upsilon\rangle\!\rangle$ reflects this idea simply because it is defined as $[\![N,\upsilon\lfloor_0]\!]\cdot 0 \vee \cdots \vee [\![N,\upsilon\lfloor_m]\!]\cdot m$. The action of the operation $(\cdot)\lfloor_r$ is to make the rank information flow top-down. The valuation $\upsilon\lfloor_r$ is an update of υ when it has met the rank r. What is happening is that the part of the valuation that is concerned with ranks strictly smaller than r is erased; the part that is concerned with ranks strictly greater than r is left unchanged. And for the part that is concerned with the rank r, as this rank has been met, it is

[4]Strictly speaking, as N may be of arbitrary type and may contain free variable of arbitrary type, its Böhm tree may not be a suitable structure for a run of the automaton. Actually, for higher order variables, the model describes part how accepting runs are being constructed. This is the reason why, the metaphor rather adequate. We thus indulge ourselves in following this intuition.

allowed to see any lower ranks, provided that they agree with the rank of the concluding state.

The work of Grellois and Melliès [141] emphasize that such a treatment of ranks in the model is reminiscent of constructions in linear logic. In particular the identity $\langle\!\langle M, \upsilon\rangle\!\rangle^{\partial} = [\![M, \upsilon]\!]$ makes us understand the operation that maps $\boldsymbol{\lambda}\upsilon.[\![M, \upsilon]\!]$ to $\boldsymbol{\lambda}\upsilon.\langle\!\langle M, \upsilon\rangle\!\rangle$ as similar to *promotion* in linear logic while the operation $(\cdot)^{\partial}$ can be seen as *dereliction*. Again, as the model we treat performs some convergence test, we do not know yet whether it has decomposition into a model of linear logic, in particular, we do not know how to interpret the (`strat`) condition in linear logic. The model we propose depends in its very shape on the way the automaton associates ranks to states. This dependence can be seen in two places, the definition of $\mathcal{R}^{ms}_{o}$ and in the (`strat`) condition. On the contrary, the construction of Grellois and Melliès is independent from the rank that the automaton associates to the states. In a certain sense, their construction is more generic, but this is at the cost of Ω-blindness.

The value $\mathrm{fix}(h, l)$ is the fixpoint of h in contexts where the maximal rank met is l. The computation is started with $\mathrm{fix}(h, m)$ which follows the alternation of least and greatest fixpoint used to solve parity games with formulae of the μ-calculus (see [43]). By definition $\mathrm{fix}(h, m) = \sigma f_m \dots \mu f_1.\nu f_0.(h\lfloor_m)^{\partial}(\bigvee_{i=0}^{m} f_i \cdot i)$, recall that $\mathrm{fix}(h, m)$ is in $\mathcal{D}_A$ while h is in $\mathcal{R}^{ms}_{A\to A}$, so as to build a value in $\mathcal{D}_A$, we take $(h\lfloor_m)^{\partial}$ which is representing the meaning conveyed by h in a context where the maximal rank met is m. Then, each f_i represent the value of the fixpoint of $h\lfloor_m$ when, locally to the computation of the fixpoint, the maximal rank met is i. Then once $\mathrm{fix}(h, m)$ is computed, $\mathrm{fix}(h, m-1)$ is computed similarly except that when locally the rank m is met we can now use $\mathrm{fix}(h, m)$. This way, in following the decreasing order of ranks, we can compute every $\mathrm{fix}(h, l)$.

It is not obvious from the above clauses that all the elements have the required properties, it is even not clear that the right hand sides define elements of the model. Nevertheless, this can be proven (c.f. [S26]) and moreover that indeed, the meaning of terms is invariant under $\beta\delta$-conversion.

Theorem 63 *For every M, N and υ, if $M =_{\beta\delta} N$, then $[\![M, \upsilon]\!] = [\![N, \upsilon]\!]$ and $\langle\!\langle M, \upsilon\rangle\!\rangle = \langle\!\langle N, \upsilon\rangle\!\rangle$.*

Another interesting property of the model that we can prove is that:

Lemma 64 For every f in $\mathcal{D}_{B\to A\to A}$, and every $l \in [0, m]$, we have:

$$\boldsymbol{\lambda}y.fix^{A}(f(y), l) = fix^{B\to A}(\boldsymbol{\lambda}zy.f(y)(z\,y), l)\ .$$

This identity is called *the abstraction identity* by Bloom and Esik. It is a rather natural identity for fixpoints to satisfy and they propose it as an axiom of what they call a Conway CCC [56] whose aim is to be a minimal axiomatization of models of the λY-calculus.

Now that we have a model, it can be established that this model has the same recognizing power as the automaton $\mathcal{A}$.

Theorem 65 *For every closed term M of type 0, $q \in [\![M]\!]$ iff $\mathcal{A}$ accepts $BT(M)$ from state q.*

5.6 Adequacy of the model via parity games

We are going to briefly explain the main lines of the proof of Theorem 65. As we already explained the proof consists mainly in transforming an infinite game into a finite one. In the context of infinitary λY-calculus, i.e. an extension of λY-calculus where the terms are allowed to be infinite, under certain hypothesis, the smaller game (which is that case may not be finite) may be defined using an MSOL transduction inside the original λY-term leading to a transfer theorem that generalizes (on tree structures) already known logical transfer theorems such as unfolding [93], Müchnik iteration [221, 290] and a result by Courcelle and Knapik showing that first order substitution is MSOL compatible [92].

The main idea of the proof is following the general guideline we outlined in the beginning of this section. More precisely the idea consists in constructing an infinite game that ties the execution of the Krivine machine when it computes $KTree(M)$ and the possible executions of the automaton $\mathcal{A}$ on that very tree. The invariants of the Krivine machine and in particular Lemma 57 ensure that Eve has a winning strategy in that game iff $\mathcal{A}$ accepts $BT(M)$.

So as to make the presentation of the game simpler, it is convenient to make a distinction between variables that are bound by λ and those that are bound by Y combinators. Later, this distinction will be important so as to present the transfer theorem. Typographically, this distinction will be represented by writing variables bound by fixpoint combinators using boldface fonts as in $Y\mathbf{x}.M$, while we will keep normal fonts for λ-variables as in $\lambda x.M$. We shall call variables bound by Y combinators *recursive variables*. In a term M, we assume that the naming of recursive variables is so that their names are pairwise distinct allowing us to have a function $term_M$ that maps those variables to the term where they are bound in a term M. For example, if $M = C[Y\mathbf{x}.N]$, $term_M(\mathbf{x})$ should be $Y\mathbf{x}.N$. In the sequel, as M is always clear from the context, we write $term(\mathbf{x})$ in the place of $term_M(\mathbf{x})$.

To this convention about recursive variables we add another one which requires that subterms which are built with Y combinators should be closed. It is always possible to transform a term M into a term M' that satisfy this convention and so that $BT(M) = BT(M')$. For this, it suffices to replace each subterm of the form $Y\mathbf{x}.N$ whose free variables are $x_1, \ldots, x_n$ by $Px_1 \ldots x_n$ where $P = Y\mathbf{z}.\lambda x_1 \ldots x_n.N[\mathbf{x} \leftarrow \mathbf{z}x_1 \ldots x_n]$. Using Lemma 64, we easily prove that for every valuation v, $[\![Px_1 \ldots x_n, v]\!] = [\![Y\mathbf{x}.N, v]\!]$ so that we can prove the correctness of the model under this restriction without loss of generality.

The game is defined on top of the structure $RT(\mathcal{A}, M)$, the runs of the automaton $\mathcal{A}$ on the graph of configurations of the Krivine Machine computing $BT(M)$. The actual runs of $\mathcal{A}$ on $BT(M)$ can easily be read off $RT(\mathcal{A}, M)$.

The labels of the tree $RT(\mathcal{A}, M)$ will be of the form $(N,\rho)\geq C$ where N is a term, ρ is an environment, and C is a *closure expression*. The latter is either just a state q or has the form (u, K, ρ') where u is a node of $RT(\mathcal{A}, M)$, and (K, ρ') is a closure. We will also have labels with indices $(N,\rho)\geq_{ind} C$, where ind is a pair of states or a node of $RT(\mathcal{A}, M)$.

Definition 66 For a given closed term M of type o, and a parity automaton $\mathcal{A}$ we define the tree of runs $RT(\mathcal{A}, M)$ of $\mathcal{A}$ on the graph of configurations of the execution of the Krivine machine on M:

1. The root of the tree is labelled with $(M, \emptyset)\geq q^0$.
2. A node labelled $(a,\rho)\geq C_0 \mapsto C_1 \mapsto q$ has a successor $(a,\rho)\geq_{q_0,q_1} C_0 \mapsto C_1 \mapsto q$ for every $(q_0, q_1) \in \delta(q, a)$.
3. A node labelled $(a,\rho)\geq_{q_0,q_1} C_0 \mapsto C_1 \mapsto q$, where $C_i = (u_i, N_i, \rho_i)$, has successors $(N_i, \rho_i) \geq_{u_i} q_i$ for $i = 0, 1$.
4. A node labelled $(\lambda x.N, \rho) \geq C \mapsto D$ has a unique successor labelled $(N, \rho[x \mapsto C]) \geq D$
5. A node u labelled with $(Y\mathbf{x}.N, \rho) \geq C$ has a unique successor $(N, \rho)\geq C$.
6. A node labelled $(\mathbf{x}, \rho)\geq C$, for $\mathbf{x}$ a recursive variable, has a unique successor $(term(\mathbf{x}), \emptyset)\geq C$.
7. A node u labelled $(NK, \rho)\geq C$ has a unique successor labelled $(N, \rho)\geq (u, K, \rho) \mapsto C$. We say that here a *$u$-closure is created.*
8. A node labelled $(x, \rho)\geq C$, for x a λ-variable and $\rho(x) = (u', N, \rho')$, has a unique successor labelled $(N, \rho') \geq_{u'} C$.
9. A node labelled $(N, \rho) \geq_u C$ has a unique successor labelled $(N, \rho)\geq C$.

We will say that in the nodes of the form $(N, \rho) \geq_u C$ the closure (u, N, ρ) is *used.*

The definition is as expected but for the fact that in the rule for the application we store the current node in the closure. When we use the closure in the variable rule or constant rule (rules 8 and 3), the stored node does not influence the result. This technical detail plays an important in the proof: it allows us to give a precise account of the contexts in which closures are used. Notice also that the convention that we have taken to use Y-combinators only on closed terms has the consequence that, when a recursive variable is to be evaluated, we do not need to retrieve the environment in which that term was to be evaluated. Actually, the transformation of terms into terms that satisfy our convention that we have outlined amounts to store the environment on the stack of the machine.

Notice also that the rules 2,3,4 rely on the typing properties of the Krivine machine ensured by the definition of its configurations. Indeed, when the machine reaches a configuration of the form $(a, \rho) \geq C$ then, since we are working with a tree signature, a is either of type o or of type $o \to o \to o$. In consequence, C is of the form $D_0 \mapsto D_1 \mapsto q$ where D_0, D_1 are two closures of type o. The environment ρ plays no role in such a configuration as a is a constant. Also from the typing invariant we get that, when the machine is in a configuration like $(\lambda x.N, \rho) \geq C$ then C is of the form $C' \mapsto D$.

Definition 67 We use the tree $RT(\mathcal{A}, M)$ to define a parity game between two players[5]: Eve chooses a successor in nodes of the form $(a, \rho) \geq C$, and Adam in nodes $(a, \rho) \geq_{q_0, q_1} C$. The rank of a node $(N, \rho) \geq \vec{D} \mapsto q$ is $\mathtt{rk}(q)$ (and similarly nodes with indices). The max parity condition decides who wins an infinite play. Let us call the resulting game $\mathcal{K}(\mathcal{A}, M)$.

In the game $\mathcal{K}(\mathcal{A}, M)$, when the Krivine machine enters a diverging computation, there is a state q so that every position of the play has a label of the form $(N, \rho) \geq \vec{D} \mapsto q$. So, in that situation, Eve wins only in the case where the rank of q is even. This is consistent with the fact that the automaton $\mathcal{A}$ is Ω-even, and that it accepts nodes labeled Ω only with states of even rank. Together with this remark and Lemma 57, the following is a direct consequence of the definitions.

Proposition 2 *For every parity automaton $\mathcal{A}$ and concrete canonical term M. Eve has a strategy from the root position in $\mathcal{K}(\mathcal{A}, M)$ iff $\mathcal{A}$ accepts $BT(M)$.*

The above proposition reduce the problem whether $BT(M)$ is accepted by $\mathcal{A}$ to deciding who has a winning strategy from the root of $\mathcal{K}(\mathcal{A}, M)$. We now introduce a finite game $G(\mathcal{A}, M)$, and show that the winner in the two games is the same.

The positions of the game are of the form $(N, \upsilon) \geq S$ where N is a subterm of M, υ is a valuation in $\mathcal{R}^{ms}$, and S is join irreducible element of $\mathcal{D}$, we will write S using the step function notation. The fact that S is join-irreducible implies that S must be of the form $R_1 \mapsto \cdots \mapsto R_n \mapsto q$ for some $R_1, \ldots, R_n$ in $\mathcal{R}^{ms}$ and a state q. We will also have positions with indices $(N, \upsilon) \geq_{ind} S$, where *ind* is a pair of states, a rank, or a residual.

Definition 68 The game $\mathcal{G}(\mathcal{A}, M)$ is as follows:

1. The initial position is $(M, \emptyset) \geq q^0$

2. A node $(a, \upsilon) \geq R_0 \mapsto R_1 \mapsto q$ has a successor $(a, \upsilon) \geq_{q_0, q_1} R_0 \mapsto R_1 \mapsto q$ for every $(q_0, q_1) \in \delta(q, a)$.

[5]We only specify to whom positions with several successors belong. For positions from which there is a unique possible move, they can belong to any of the players without changing anything to the outcomes of the game.

3. A node $(Y\mathbf{x}.N, \upsilon) \geq S$ has a successor $(N, \upsilon) \geq S$.

4. A node $(\mathbf{x}, \upsilon) \geq S$, for $\mathbf{x}$ a recursive variable, has a successor $(\mathit{term}(\mathbf{x}), \upsilon) \geq S$.

5. A node $(NK, \upsilon) \geq S$ has a successor $(NK, \upsilon) \geq_R S$ for every R residual of the type of K.

6. A node $(NK, \upsilon) \geq_R \vec{S} \mapsto q$ has two types of successors

   - one successor $(N, \upsilon) \geq R\lfloor_q \mapsto \vec{S} \mapsto q$, and
   - for every $\vec{P}$ and every $(q', r') \in R(\vec{P})$ with $r' \geq \max(rk(q'), rk(q))$ a successor $(K, \upsilon) \geq_{r'} \vec{P} \mapsto q'$. Actually the restriction $r' \geq rk(q')$ is not needed since we work only with pairs (q', r') with $r' \geq rk(q')$. The restriction $r' \geq rk(q)$ is not needed too since Eve has no incentive to play R with $(q', i) \in R(\vec{P})$ for some $i < rk(q)$. Anyway this pair will be removed in $R\lfloor_q$ so it does not help in the left branch. It is just in the proof that we should note that indeed the residual of a closure has this property.

7. A node $(K, \upsilon) \geq_{r'} \vec{P} \mapsto q'$ has a unique successor $(K, \upsilon\lfloor_{r'}) \geq \vec{P} \mapsto q'$.

The rank of a node labelled $(N, \upsilon) \geq \vec{S} \mapsto q$ is the rank of q. The rank of a node labelled $(N, \upsilon) \geq_r \vec{S} \mapsto q$ is r, while the rank of a node labelled $(N, \upsilon) \geq_R \vec{S} \mapsto q$ is 0 when R is a residual.

A position $(a, \upsilon) \geq_{q_0, q_1} R_0 \mapsto R_1 \mapsto q$ is winning for Eve iff $(q_i, rk(q_i)) \in R_i\lfloor_{\max(rk(q), rk(q_i))}$ for $i = 0, 1$.

A position $(x, \upsilon) \geq \vec{S} \mapsto q$ is winning for Eve iff $(q, rk(q)) \in \upsilon(x)(\vec{S})$.

The main property of $\mathcal{G}(\mathcal{A}, M)$ is that it is equivalent to $\mathcal{K}(\mathcal{A}, M)$ in the following sense:

Theorem 69 *Eve has a winning strategy in $\mathcal{K}(\mathcal{A}, M)$ iff she has a winning strategy in $\mathcal{G}(\mathcal{A}, M)$.*

After that, it suffices to remark that for any term N, a given valuation υ and a join-irreducible element S of $\mathcal{D}$, we may define a game similarly to $\mathcal{G}(\mathcal{A}, M)$ whose initial node is $(N, \upsilon) \geq S$. Then, we can prove that Eve has a winning strategy in that game iff $[\![N, \upsilon]\!] \geq S$. This final remark entails the correctness theorem of the model. The main technical part of the proof of that theorem is contained in Proposition 69. We will here give a presentation the proof of that theorem.

Residuals in $\mathcal{K}(\mathcal{A}, M)$

The key notion of the proof of equivalence of the games $\mathcal{K}(\mathcal{A}, M)$ and $\mathcal{G}(\mathcal{A}, M)$, the notion of *residuals of nodes*. Given a subtree $\mathcal{T}$ of $\mathcal{K}(\mathcal{A}, M)$, i.e. a tree

obtained from $\mathcal{K}(\mathcal{A}, M)$ by pruning some of its branches, we calculate the residuals $R_{\mathcal{T}}(u)$ and $res_{\mathcal{T}}(u, u')$ for some nodes u and pair of nodes (u, u') of $\mathcal{T}$, where u' is a descendant of u. In particular, $\mathcal{T}$ may be taken as being a strategy of Eve or a strategy of Adam. When $\mathcal{T}$ is clear from the context we will simply write $R(u)$ and $res(u, u')$. The value of $R(u)$ is simply recording the contexts in which the closure created at u is used in $\mathcal{T}$. The value $res(u, u')$ is simply an update of u with respect to the maximal rank that has been seen on the path from u to u'.

Recall that a node v in $\mathcal{K}(\mathcal{A}, M)$ is an application node when its label is of the form $(NK, \rho) \geq C$. In such node a closure (u, K, ρ) is created. We will define a residual $R(u)$ for such a closure. This is done by induction on types. We also define a variation of this notion: a residual $R(u)$ seen from a node u', denoted $res(u, u')$. The two notions are the main technical tools used in the proof of the theorem.

In the sequel, when u is an ancestor of u' in $\mathcal{T}$ then we write $\max(u, u')$ for the maximal rank appearing on the path between u and u', including both ends, and we let $res(u, u')$ be defined by $R(u){\lfloor}_{\max(u,u')}$.

Consider an application node u in $\mathcal{T}$. It means that u has a label of the form $(NK, \rho) \geq C$, and its unique successor has the label $(N, \rho) \geq (u, K, \rho) \mapsto C$. That is the closure (u, K, ρ) is created in u. We will look at all the places where this closure is used and summarize the information about them in $R(u)$. We write $\mathcal{U}(u)$ to denote the set of nodes where u is used, i.e. the set of nodes u' of the form $(K, \rho) \geq_u C_1 \mapsto \cdots \mapsto C_k \mapsto q$. In that case, supposing that for $i \in [k]$ $C_i = (u_i, N_i, \rho_i)$, we let

$$F[u, u'] = res(u_1, u') \mapsto \cdots \mapsto res(u_k, u') \mapsto (q, \max(u, u')) \ .$$

Notice, that the closure (K, ρ) has a type of the form $A_1 \to \cdots \to A_k \to o$ and thus, the closures $C_1, \ldots, C_k$ respectively have type $A_1, \ldots, A_k$. Therefore, the definition of $F[u, u']$ depends only on the definition of residuals of nodes with smaller types. We now define $R(u)$ as:

$$R(u) = \bigvee \{F[u, u'] \mid u' \in \mathcal{U}(u)\} \ .$$

Using the definitions of $R(u)$, we are now in position to prove that if Eve has a winning strategy in $\mathcal{K}(\mathcal{A}, M)$, then she has a winning strategy in $G(\mathcal{A}, M)$ and if Adam has a winning strategy in $\mathcal{K}(\mathcal{A}, M)$, then he has one in $\mathcal{G}(\mathcal{A}, M)$. As $G(\mathcal{A}, M)$ is a parity game, it is determined and either Eve or Adam has a winning strategy. Therefore, this proves that Eve has a winning strategy in $\mathcal{K}(\mathcal{A}, M)$ iff she has one in $G(\mathcal{A}, M)$.

The idea behind the construction of $G(\mathcal{A}, M)$ is based on an essential technical idea of Walukiewicz [290]. Here, this idea consists in representing finite part of plays with residuals. At the level of application, Eve has to choose a residual that is supposed to represent at least all the contexts in which the arguement is to be used in the sequel of the game. Then Adam may test whether this choice of a residual exhausitvely represents all the situations in which the

argument is actually used by continuing the game in the applicand. He may also choose one of the possible situations described in Eve's residual and continue the game from this position. The best interest of Eve is to play at each application node a residual of a node that is in her strategy in $\mathcal{K}(\mathcal{A}, M)$. In that case, when Adam chooses one of the situations described by the residual, at the level of $\mathcal{K}(\mathcal{A}, M)$, this amount for him to jump from a node u to a node u' where $F[u, u']$ is the description of that situation. When Eve has a winning strategy, she must win whatever the choice of Adam is, so her choice of a residual needs to lead to a winning position no matter what Adam chooses to do: either checking the exhaustivity of the residuals in the applicand, or jumping to a particular situation it describes. The proof of this result can be found in [S26].

The advantage of this methods that uses two kinds of games, a large game built on executions of Krivine machines which has a transparent relationship with runs of automata on the Böhm trees; and a small game that summarizes Krivine machines configurations by means of valuations can be generalized to the infinitary λY-calculus[6]. Indeed Krivine machines are able to compute the Böhm trees associated to infinitary λ-terms, and then under certain conditions the small game can be defined within the big one by MSOL transduction. As in the small game, determining the winner can be done within MSOL, this allows us to reduce the MSOL theory of Böhm trees to the MSOL theory of the infinitary terms that generate them. In other words, we obtain a *logical transfer theorem.*

The theorem relates logical theories of (infinitary) λY.terms and Böhm trees. We will consider monadic-second order logic (MSOL) on such objects. For this, it will be essential to restrict to some finite set of λ-variables: both free and bound. On the other hand, we will be able to handle infinitely many recursive variables. Once we make it clear how to represent terms and Böhm trees as logical structures, we will also state our main theorem.

Let us fix a tree signature Σ with finitely many constants other than Ω. We would like to consider terms as models of formulas of monadic second-order logic. We will work with terms over some arbitrary but finite vocabulary. We take a finite set of typed λ-variables $\mathcal{X} = \{x_1^{\alpha_1}, \ldots, x_k^{\alpha_k}\}$, and a finite set of types $\mathcal{T}$. We denote by $Terms(\Sigma, \mathcal{T}, \mathcal{X})$ the set of *infinite closed concrete terms* M over the signature Σ such that M uses only λ-variables from $\mathcal{X}$, and every subterm of M has a type in $\mathcal{T}$. We call *concrete terms*, terms that are not considered up to α-conversion, so it makes sense to say that bound λ-variables in M should come from $\mathcal{X}$. Observe that we do not put restrictions on the number of recursive variables used in terms.

A term from $Terms(\Sigma, \mathcal{T}, \mathcal{X})$ is a labeled tree where the labels come from a finite alphabet, but for the recursive variables. We will now eliminate the possible source of infiniteness of labels related to recursive variables. Take a

[6] In the context of infinitary λ-calculus, one may forget about the fixpoint operators as they can be defined by means of infinite λ-terms of the form $\lambda f.f(f(f(\ldots)))$. But here we will use certain restrictions that will make a distinction between variables introduced with fixpoints and variables introduced by λ's.

closed term M considered as a tree. For every node of this tree labeled by a recursive variable $\mathbf{x}^A$ we put an edge from the node to the node labeled $Y^A\mathbf{x}^A$ where it is bound. Since M is closed, such a node is an ancestor of the node labeled by $\mathbf{x}$. In the next step we use a fresh symbol $\Rsh^A$ as a replacement of

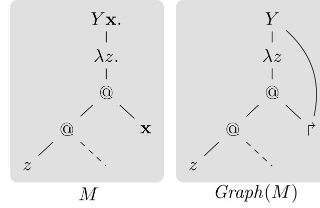


Figure 5.3: Y binding

the labels of recursive variables of type A (see Figure 5.3). Finally, we replace all labels of the form $Y^A\mathbf{x}^A$ by just Y^A. This way we have eliminated all occurrences of recursive variables from labels, but now a term is represented not as a labeled tree but as a labeled graph. Let us denote it by $Graph(M)$. Observe that the nodes of this graph have labels from a finite set, call it $Talph(\Sigma, \mathcal{T}, \mathcal{X})$.

Since $Graph(M)$ is a labeled graph over a finite alphabet, it makes sense to talk about satisfiability of an MSOL formula in this graph. We will just write $M \vDash \varphi$ instead of $Graph(M) \vDash \varphi$. The first, easy but important, observation is that for fixed Σ, $\mathcal{T}$, $\mathcal{X}$, there is an MSOL formula determining if a graph is of the form $Graph(M)$ for some $M \in Terms(\Sigma, \mathcal{T}, \mathcal{X})$. For this it suffices to use a formula that expresses the local constraints imposed by typing and which are expressible in MSOL.

For a closed term $M \in Terms(\Sigma, \mathcal{T}, \mathcal{X})$ of type o, its Böhm tree is a tree with nodes labeled by symbols from Σ. Hence one can talk about satisfiability of MSOL formulas in $BT(M)$. The Transfer Theorem says that evaluation, that is the function assigning to a term its Böhm tree, is MSOL compatible.

Theorem 70 (Transfer Theorem) *Let Σ be a finite tree signature, $\mathcal{X}$ a finite set of typed variables, and $\mathcal{T}$ a finite set of types. For every MSOL formula φ one can effectively construct an MSOL formula $\widehat{\varphi}$ such that for every λY-term $M \in Terms(\Sigma, \mathcal{T}, X)$ of type 0:*

$$BT(M) \vDash \varphi \qquad \textit{iff} \qquad M \vDash \widehat{\varphi}.$$

Notice that the formula $\widehat{\varphi}$ is independent from M; if it were dependent on M then we would simply obtain Ong's theorem since we could take $\widehat{\varphi}$ to be either *tt* or *ff*.Notice also that the formula $\widehat{\varphi}$ is constructed for an infinite family of terms provided they use only the lambda-variables from $\mathcal{X}$.

In [S27] we show that, a variant of global model-checking [66, 65] which was so far considered as a genuine extension of Ong's Theorem follows directly from the Transfer Theorem.

5.7 Conclusion and perspectives

In this chapter we have exposed a model approach to the verification of behavioral properties of higher-order programs. In the setting of verification, it extends the verification of MSOL properties on the behavior of programs to a much wider class. Trying to stick to a denotational approach may seem a bit too demanding for this kind of problems, nonetheless we have seen some of its virtues. For example, the relationship between trivial automata and monotone models not only gives a nice framework to design algorithms for verifying safety and reachability properties, but it also makes explicit the Ω-blindness property of these automata and the relation is proved by means of very standard methods. This considerably simplifies existing proofs such as the one of Aehlig [36] or of Kobayashi [184]. Moreover, it nicely relates this method to older research on strictness analysis [72], and the verification of safety properties for higher-order programs [159] and then to many of the algorithmic proposals that have been made in that context. In the case of weak MSOL, the model approach has led us to propose an extension of the operation of wreath product to applicative structures. Moreover, the duality underlying this construction can also be exploited algorithmically as it gives two dual procedures on that proves and the other that disproves facts about programs. This construction may have other applications outside the area of model checking, for example, in the semantics of natural language, where it can be used to combine several analyses which depend on each other. Finally the model for MSOL shows that finite models can capture complex infinitary properties. The connection with ideas from linear logic shows how this approach can serve as a bridge between automata theory and denotational semantics. Moreover, the central role in our approach that the Krivine machine is having is yet another example of how methods coming from the study of λ-calculus can successfully be used to cope with formal language problems.

Our line of work puts forward the problem of the evaluation of λY-terms in models. As the constructions we have exhibited show, evaluation methods can be applied to many verification problems. In particular, this connects the verification of behavioral properties with abstract interpretation which has developed a wide range of methods for the efficient evaluation of programs in particular domains. The abstraction/refinement methods and fixpoint acceleration techniques seem promising for this particular problem.

Another advantage of the explicit construction of models is that we can use them in combination with other models. An example would be the verification of a program that is to be executed in a complex environment. The internal state of the program may be modeled by means of usual denotational semantics. Then the messages it exchanges with its environment may be modeled by means of a possibly infinite tree which is required to verify an MSOL specification. As the internal state and the external behavior of the program are connected, we may combine the two semantic models so as to verify whether the specification is met. Here abstract interpretation methods can be used so as to compute approximations of the internal state and also executions of a parity automaton

on the behavior of the program.

In turn, this work raises the question of the expressiveness of finite models of λY-calculus about syntactic properties of Böhm trees. This means models of λY-calculus which give the same interpretation, in other words, equate, terms that have the same Böhm trees. From now on, we will call these models *Böhm models*. What kind of properties about infinitary terms can they capture? Or more precisely, do they capture a wider class of properties than MSOL properties? These questions about the frontiers of recognizability are related to fundamental questions that have remained open for a long time now. For example, a natural class of algebras that have been proposed by Esik [124] under the name of *iteration algebras* axiomatize precisely those algebras that equate regular trees. The question whether finite iteration algebras capture a wider class than MSOL properties has remained open since their definition at the beginning of the 80s. It may be the case that answering this question is easier in the context of λY-calculus as higher-order brings a wider class of operations.

An interesting aspect of the semantics of λY-calculus, is that the full abstraction problem of PCF has produced a wide variety of constructions from which we can get inspiration. In particular intentional models have been proposed rather early. In general these models are better understood from the point of view of linear logic which also brings a large set of tools. These kinds of models come naturally with a rich information about the flow of programs. The question then amounts to seeing whether this information can be rich enough so that the interpretation of fixpoints can exploit it to go beyond MSOL properties. A good example of a property that is not MSOL definable is boundedness. The simplest instance of this kind of properties is on trees built with only a unary operation a and a binary one b is the following: there is a bound on the number of a in each branch of the tree. The set of trees that have this property cannot be recognized with a parity tree automaton. The set of terms whose Böhm trees have this property can be recognized using a least fixpoint interpretation of terms where the atomic type is interpreted as the infinite domain $0 < 1 < 2 \cdots k < k+1 \cdots < \omega$ and where $b\,x\,y$ is interpreted as $\max(x, y)$ and $a\,x$ is interpreted as $x+1$. For that particular case, is it possible to construct a finite model which recognizes the set of trees whose semantics in that models is different from ω? The idea here would be to use an intentional semantics which gives a precise account about the occurrences of a so as to make fixpoints able to see when their number increases unboundedly in branches; rather quickly we remark that we also need to treat the variables and relations between them in a similar way. Interestingly answering positively to that problem leaves open the problem we mentioned above about iteration algebras. Indeed, even though a syntactic model of λY-calculus naturally induces an iteration algebras by only considering second order functions, the fact that this algebra would recognize regular trees with a bounded number of a's in any branch does not entail that iteration algebras capture properties that are not MSOL definable. This comes from the fact that on regular trees, this boundedness property can be observed by the absence of branch with infinitely

many a's; which is a property that is MSOL definable. The equivalence of these properties becomes false as soon as we consider algebraic trees. Ultimately, if we are able to construct a finite model for this particular property, our long term goal would be to construct a finite model for the logic weak MSOL plus the Unbounding quantifier that has been proposed and studied by Bojańczyk [61]. This would extend verification procedure to more qualitative properties in particular concerning bounded usage of resources, reactivity etc...

Another interest that we have in constructing a finite model of λY-calculus that captures boundedness properties is that it may shed some light on algebras of infinite trees. So to understand why, it is best to take a look at the situation on infinite words. Courcelle [87] is pointing the fact that, at that moment, sets of infinite words recognized by MSOL formulae "[are] not algebraic since the corresponding sets of infinite [...] words are not recognized with respect to any (known) algebraic structure". Wilke [295] proposed a class of finitary algebras, nowadays called *Wilke algebras*, that recognizes exactly regular infinite words that satisfy MSOL properties. The main interest of Wilke algebras, is that they naturally induce a *unique* interpretation of any infinite words. This means that if we are to extend Wilke algebras into a model of λY-calculus, then it may be the case that there is a unique way of defining higher-order fixpoints. While iteration algebras have been defined a decade before Wilke algebras, and while, when they are restricted to unary trees, they coincide with Wilke algebras, it is unknown whether they induce a unique interpretation of any infinite trees. The construction of a model for the boundedness property we discussed above would shed some light about this problem. Indeed, as we explained, such a model would induce, by restricting it to regular terms, an iteration algebra that would recognize an MSOL property while the model itself would recognize a property that is not MSOL. This would suggest that this particular iteration algebra may well induce two different interpretations of arbitrary infinite trees that coincide on the class of regular trees (it may also be the case that as the order of type increases the model gets refined and that we just obtain a refinement of this MSOL property). As a consequence, we may underline this way a deep difference between infinite trees and infinite words. On the other hand, this would leave open the question of the expressivity of iteration algebras about the classes of regular trees they recognize. Nonetheless, this would give arguments in favor of the more restrictive approach of Blumensath [58] who proposed made the first variety of algebras for infinite trees that capture exactly MSOL properties and induce a unique interpretation of infinite trees.

More abstractly, as we are interested in describing a class of Böhm models of λY-calculus, we would like to have axioms describing such a class. This problem is nevertheless quite challenging or even hopeless. Indeed, such axioms or axioms schemes would give a semi-decision procedure to check when two λY-terms have the same Böhm tree. Showing the problem of deciding that two terms have the same Böhm tree is recursively enumerable. But this problem is naturally co-recursively enumerable as the computation of Böhm tree and Theorem 17 (for non-convergence) give a semi-algorithm for deciding whether

two terms have different Böhm trees. Thus, such a definition would entail the decidability of Böhm tree equivalence. Using a result of Courcelle [84] and the result of Sénizergues [267], it is known that the equivalence of algebraic trees, i.e. trees defined by λY-terms so that fixpoints are applied only to second order functions, is decidable. This result is very difficult to prove and its conjecture has remained open very long. Nonetheless, to the best of our knowledge, there has not been any extensions of iteration algebras to algebraic trees and there is no axiomatic system that describes algebras that equate algebraic trees.

A way to avoid this hard question is to focus on particular well understood classes of models and then try to understand the fixpoints that are definable in those classes. The most obvious choice is that of models constructed on top of monotone applicative structures. In these models, there are many possible definitions of fixpoints. We should try to classify those fixpoints and then try to understand when they give rise to Böhm models. Another class of interest is that of stable models. Here a problem arises that the applicative structures they use are not based on lattices but on semi-lattices and thus, there is no obvious notion of greatest fixpoints. Berry has proposed a notion of bi-domain which combines the extensional order of monotone functions and the stable order. In that case, starting with lattices, as in the case of monotone functions, the extensional order preserves the lattice structure which allows one to describe complex fixpoints based on the interleaving of least and greatest fixpoints with other bi-domain operations. One interest of stable models and then of bi-domains is that they admit interesting decompositions as models of linear logic respectively as Girard's coherence spaces [134] and Curien et al's bi-structures [97]. Linear logic then gives a good understanding of how parts of a program are used during a computation. In particular, at the level of stable models, this property translates into the possibility of computing the minimal semantic parts of a term that contributes to producing a result. For example, a stable model that interpret atomic type in the two elements lattice makes it easy to remove syntactically useless parts of λY-terms using least fixpoint computations [S20]. This construction does not extend to the infinitary λ-calculus and thus to greatest fixpoints. Technically the problem is a bit difficult to describe. Conceptually, the difficulty comes from the definition of a notion similar to Girard's notion of coherence but in an impredicative setting. A possible solution could be to distinguish two kinds of resources, the ones that justify the convergence and the actual usage in infinitary computation. This problem is related to the boundedness problem we mentioned above and seems to be a preliminary step towards understanding boundedness from a finitary perspective.

Chapter 6

Conclusion

This document has presented the main facets of my work over the past years. I have insisted on the unifying theme that recognizability has played in my research in that period of time. Recognizability allowed me to point at a generic problem that is faced both in natural language analysis and in verification: the evaluation of higher-order programs in a finite domain. This problem has a non-elementary complexity. Nevertheless, in the case of natural language, we may restrict our attention to fragments for which the computation can be performed in polynomial time. Concerning program verification, experiments [183, 282] show that there is some hope to verify non-trivial programs.

This general problem is also at the center of several research directions I would like to follow in the next future and that I mentioned in the document. I will now review these directions.

The structure of finite domains

When we evaluate a higher-order program in a finite domain, we need to understand the structure of the domain and how it may influence the evaluation.

Concerning parsing algorithms, the transformations of datalog programs that yield prefix correct algorithms are related to the sequential evaluation of the original grammar. It thus seems that sequential algorithms or strongly stable functions may be the tools of choice so as to describe prefix-correct parsing algorithms for grammars based on λ-calculus. Moreover, the decompositions of these models of λ-calculus as models of linear logic give us a gradual way of approaching this problem. We may indeed start with grammars that do not use copying and then try to generalize the approach to the copying case. This work can then be used in the problem of evaluating programs in certain domains and verify properties of programs. Here sequential algorithms and strongly stable functions may offer ways of accelerating the computation of programs in particular when we cope with safety or reachability properties. We mentioned the relationship between these kinds of models and top-down deterministic automata. This kinds of approach thus seem to be limited to

top-down deterministic specifications, but also to safety and reachability properties. It is unclear whether it can be used in a more general setting.

Related to the parsing problem, Statman syntactic model suggested us that a certain class of models we call top-down deterministic may have a decidable definability problem. If this happens, this may help us to define a new class of models that would be fully abstract for the λ-calculus. We hope that the studies of those models may give a simpler and semantic proof of the decidability of higher-order matching.

Finally, the complex fixpoints that we have defined so as to capture complex logical properties present some regularities in their definitions. Moreover, the structure of the models is closely related to the definitions of these fixpoints. A natural question then is whether the structure of the model and the definitions of fixpoints of low order fully determine the definition of higher-order fixpoints or whether there is some flexibility that can be exploited while making the definitions higher-order. Here the study of particular kinds of properties that are related to boundedness may help to understand this problem.

Efficient fixpoint computation

The work that we did on parsing algorithms based on datalog pushes to first look at generalizations of datalog to cope with non-linearity. The goal is to have a generic way of describing least fixpoint computations like the semi-naive bottom up algorithm, but then take benefit from the richness of the formalism so as to implement other strategies by means of program transformations. Another issue is to see whether it is possible to adapt datalog to the computations of fixpoints that are not the least one.

Another line of research related to the evaluation of fixpoint is to exploit the structure of domains so as to take benefit from approximation techniques. In particular, using abstraction/refinement methods and fixpoint acceleration techniques, we may make computation converge quicker. Nevertheless, this adaptation may reveal technical as these methods have been designed in relation to least fixpoint computation and may be hard to adapt to more general contexts.

The expressiveness of recognizability

Finite models of λY-calculus can capture all MSOL properties. It seems reasonable that they can also capture some qualitative properties related to boundedness. We have given a simple property we may try to model as a starting point. If we succeed in describing this property by means of a finite model, then it may help us to answer to some fundamental questions about algebras of trees. I will then open the way to model the class of properties that are captured by the logic weak MSOL plus the unbounding quantifier.

Experiments for logical approaches to linguistic models

The logical descriptions of language that we have proposed are concise and offer some modularity. They enter in a general architecture that consists in compiling linguistic descriptions to datalog programs. So as to make this architecture practical, we need to develop some heuristic of compilation and to tame the intrinsic complexity of the construction of automata that recognize logic descriptions. We also need enlarge the set of descriptions of linguistic phenomena and to test this approach on non-configurational languages.

Bibliography

[32] S. Abramsky. “Domain Theory in Logical Form”. In: *Ann. Pure Appl. Logic* 51.1-2 (1991), pp. 1–77.

[33] S. Abramsky and R. Jagadeesan. “Games and Full Completeness for Multiplicative Linear Logic”. In: *J. Symb. Log.* 59.2 (1994), pp. 543–574.

[34] J. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[35] J. Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 1996.

[36] K. Aehlig. “A Finite Semantics of Simply-Typed Lambda Terms for Infinite Runs of Automata”. In: *Logical Methods in Computer Science* 3.1 (2007), pp. 1–23.

[37] K. Aehlig, J. de Miranda, and C.-H. Ong. “Safety Is not a Restriction at Level 2 for String Languages”. In: *Foundations of Software Science and Computational Structures.* Ed. by V. Sassone. Vol. 3441. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, pp. 490–504.

[38] A. V. Aho. “Indexed Grammars - An Extension of Context-Free Grammars”. In: *J. ACM* 15.4 (1968), pp. 647–671.

[39] A. V. Aho. “Nested Stack Automata”. In: *J. ACM* 16.3 (1969), pp. 383–406.

[40] M. Amblard. “Calculs de représentations s’emantique et syntaxe générative : les grammaires minimalistes catégorielles”. PhD thesis. Université de Bordeaux 1, 2007.

[41] M. T. Aravind K. Joshi Leon S. Levy. “Tree Adjunct Grammars”. In: *Journal of Computer and System Sciences* 10.1 (1975), pp. 136–163.

[42] M. A. Arbib and Y. Give’on. “Algebra automata I: Parallel programming as a prolegomena to the categorical approach”. In: *Information and Control* 12.4 (1968), pp. 331–345.

[43] A. Arnold and D. Niwiski. *Rudiments of μ-calculus.* Vol. 146. Studies in Logic and the Foundations of Mathematics. Noth-Holland, 2001.

[44] A. Arnold and M. Dauchet. “Théorie des magmodes”. In: *ITA* 12.3 (1978).

[45] A. Arnold and M. Dauchet. “Théorie des Magmodes (II)”. In: *ITA* 13.2 (1979).

[46] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. “Magic sets and other strange ways to implement logic programs”. In: *PODS-5.* ACM. 1985, pp. 1–15.

[47] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Vol. 103. revised edition. Studies in Logic and the Foundations of Mathematics, North-Holland Amsterdam, 1984.

[48] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. “A Filter Lambda Model and the Completeness of Type Assignment”. In: *J. Symb. Log.* 48.4 (1983), pp. 931–940.

[49] T. Becker, O. Rambow, and M. Niv. *The Derivational Generative Power of Formal Systems or Scrambling is beyond LCFRS*. Tech. rep. IRCS 92-38. University of Pennsylvania, 1992.

[50] H. Beki. *Definable operations in general algebras*. Tech. rep. IBM Laboratories, Vienna, 1969.

[51] D. Bekki. “Representing Anaphora with Dependent Types”. In: *Logical Aspects of Computational Linguistics - 8th International Conference, LACL 2014, Toulouse, France, June 18-20, 2014. Proceedings*. Ed. by N. Asher and S. Soloviev. Vol. 8535. Lecture Notes in Computer Science. Springer, 2014, pp. 14–29.

[52] D. Bekki and E. McCready. “CI via DTS”. In: *New Frontiers in Artificial Intelligence - JSAI-isAI 2014 Workshops, LENLS, JURISIN, and GABA, Kanagawa, Japan, October 27-28, 2014, Revised Selected Papers*. Ed. by T. Murata, K. Mineshima, and D. Bekki. Vol. 9067. Lecture Notes in Computer Science. Springer, 2014, pp. 23–36.

[53] J. van Benthem. *Language in Action*. MIT Press, Elsevier Science Publishers B.V., 1995.

[54] G. Berry and P.-L. Curien. “Sequential algorithms on concrete data structures”. In: *Theoretical Computer Science* 20 (1982), pp. 265–321.

[55] G. Berry. “Stable Models of Typed lambda-Calculi”. In: *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*. London, UK, UK: Springer-Verlag, 1978, pp. 72–89.

[56] S. L. Bloom and Z. Ésik. “Fixed-Point Operations on ccc’s. Part I”. In: *Theor. Comput. Sci.* 155.1 (1996), pp. 1–38.

[57] W. Blum and C.-H. L. Ong. “The safe lambda calculus”. In: *Logical Methods in Computer Science* 5.1:3 (2009), pp. 1–38.

[58] A. Blumensath. “Recognisability for algebras of infinite trees”. In: *Theor. Comput. Sci.* 412.29 (2011), pp. 3463–3486.

[59] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 platform, version 0.86.1*. version 0.86.1. `http://why3.lri.fr/download/manual-0.86.1.pdf`. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay. 2015.

[60] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. “Why3: Shepherd Your Herd of Provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocaw, Poland, 2011, pp. 53–64.

[61] M. Bojaczyk. “Weak MSO with the Unbounding Quantifier”. In: *Theory Comput. Syst.* 48.3 (2011), pp. 554–576.

[62] W. S. Brainerd. “The minimalization of tree automata”. In: *Information and Control* 13.5 (1968), pp. 484–491.

[63] W. S. Brainerd. “Tree generating regular systems”. In: *Information and Control* 14.2 (1969), pp. 217–231.

[64] J. Bresnan. *Lexical-Functional Syntax*. Blackwell Textbooks in Linguistics. Blackwell Publishers, 2001.

[65] C. H. Broadbent, A. Carayol, C.-H. L. Ong, and O. Serre. “Recursion Schemes and Logical Reflection”. In: *Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science*. LICS ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 120–129.

[66] C. H. Broadbent and C.-H. L. Ong. “On Global Model Checking Trees Generated by Higher-Order Recursion Schemes”. In: *FOSSACS*. LNCS 5504. 2009, pp. 107–121.

[67] S. D. Brookes and S. Geva. “Sequential Functions on Indexed Domains and Full Abstraction for a Sub-Language of PCF”. In: *Mathematical Foundations of Programming Semantics, 9th International Conference, New Orleans, LA, USA, April 7-10, 1993, Proceedings*. Ed. by S. D. Brookes, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt. Vol. 802. Lecture Notes in Computer Science. Springer, 1993, pp. 320–332.

[68] A. Bucciarelli and T. Ehrhard. “Sequentiality in an Extensional Framework”. In: *Inf. Comput.* 110.2 (1994), pp. 265–296.

[69] A. Bucciarelli, T. Ehrhard, and G. Manzonetto. “A relational semantics for parallelism and non-determinism in a functional setting”. In: *Ann. Pure Appl. Logic* 163.7 (2012), pp. 918–934.

[70] J. R. Büchi. “Weak Second-Order Arithmetic and Finite Automata”. In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92.

[71] J. Buchi. “Using Determinancy of Games to Eliminate Quantifiers”. In: *Fundamentals of Computation Theory*. Ed. by M. Karpiski. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1977, pp. 367–378.

[72] G. L. Burn, C. Hankin, and S. Abramsky. “Strictness Analysis for Higher-Order Functions”. In: *Sci. Comput. Program.* 7.3 (1986), pp. 249–278.

[73] S. Chatzikyriakidis and Z. Luo. “Natural Language Inference in Coq”. In: *Journal of Logic, Language and Information* 23.4 (2014), pp. 441–480.

[74] N. Chomsky. “Aspects of the theory of syntax Cambridge”. In: *Multilingual Matters* (1965).

[75] N. Chomsky. *Lectures on government and binding: The Pisa lectures*. 9. Walter de Gruyter, 1993.

[76] N. Chomsky. *Syntactic Structures*. Mouton, de Gruyter, 1957.

[77] N. Chomsky. *The minimalist program.* Vol. 28. MIT Press, 1995.

[78] A. Church. *The calculi of Lambda Conversion.* Princeton University Press, 1941.

[79] A. Church. "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68.

[80] A. Church and J. B. Rosser. "Some Properties of Conversion". In: *Transactions of the American Mathematical Society* 39.3 (1936), pp. 472–482.

[81] P.-M. Cohn. *Universal Algebra.* Dordrecht, Netherlands: D.Reidel Publishing, 1981.

[82] H. Comon and Y. Jurski. "Higher-Order Matching and Tree Automata." In: *CSL.* 1997, pp. 157–176.

[83] D. J. Weir. "Characterizing mildly context-sensitive grammar formalisms". Supervisor-Aravind K. Joshi. PhD thesis. Philadephia, PA: University of Pennsylvania, 1988.

[84] B. Courcelle. "A Representation of Trees by Languages I". In: *Theor. Comput. Sci.* 6 (1978), pp. 255–279.

[85] B. Courcelle. "An axiomatic definition of context-free rewriting and its application to NLC graph grammars". In: *Theoretical Computer Science* 55.2-3 (1987), pp. 141–181.

[86] B. Courcelle. "Monadic Second-Order Definable Graph Transductions: A Survey". In: *Theoritical Computer Science* 126.1 (1994), pp. 53–75.

[87] B. Courcelle. "On recognizable sets and tree automata". In: *Resolution of equations in algebraic structures* 1 (1989), pp. 93–126.

[88] B. Courcelle. "The monadic second-order logic of graphs IX: Machines and their behaviours". In: *Theoretical Computer Science* 151.1 (1995). Selected Papers of the Workshop on Topology and Completion in Semantics, pp. 125–162.

[89] B. Courcelle and I. Durand. "Automata for the verification of monadic second-order graph properties". In: *J. Applied Logic* 10.4 (2012), pp. 368–409.

[90] B. Courcelle and J. Engelfriet. "A Logical Characterization of the Sets of Hypergraphs Defined by Hyperedge Replacement Grammars". In: *Mathematical Systems Theory* 28.6 (1995), pp. 515–552.

[91] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach.* Vol. 138. Encyclopedia of mathematics and its applications. Cambridge University Press, 2012.

[92] B. Courcelle and T. Knapik. "The evaluation of first-order substitution is monadic second-order compatible". In: *TCS* 281 (2002), pp. 177–206.

[93] B. Courcelle and I. Walukiewicz. "Monadic Second-Order Logic, Graphs and Unfoldings of Transition Systems". In: *Ann. of Pure and Appl. Log.* 92 (1998), pp. 35–62.

[94] P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL 1977*. ACM, 1977, pp. 238–252.

[95] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. “The ASTRÉE analyzer”. In: *Programming Languages and Systems*. Springer, 2005, pp. 21–30.

[96] A. B. Cremers and O. Mayer. “On vector languages”. In: *Journal of Computer and System Sciences* 8.2 (1974), pp. 158–166.

[97] P. Curien, G. D. Plotkin, and G. Winskel. “Bistructures, bidomains, and linear logic”. In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. Ed. by G. D. Plotkin, C. Stirling, and M. Tofte. The MIT Press, 2000, pp. 21–54.

[98] H. B. Curry. “Some Logical Aspects of Grammatical Structure”. In: *Structure of Language and Its Mathematical Aspects*. Ed. by R. Jakobson. AMS Bookstore, 1961, pp. 56–68.

[99] L. Damas. “Type Assignment in Programming Languages”. PhD thesis. University of Edinburgh, 1985.

[100] L. Damas and R. Milner. “Principal Type-Schemes for Functional Programs”. In: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*. Ed. by R. A. DeMillo. ACM Press, 1982, pp. 207–212.

[101] W. Damm. “The IO- and OI-Hierarchies.” In: *Theoretical Computer Science* 20 (1982), pp. 95–207.

[102] W. Damm and E. Fehr. “A schematalogical approach to the alalysis of the procedure concept in algol-languages”. In: *Proc. 5eme Colleque de Lille sur les Arbres en Algebre et en Programmation, Lilli, France, 21, 22 et 23 févier 1980*. Université de Lille 1, 1980, pp. 130–134.

[103] W. Damm and A. Goerdt. “An Automata-Theoretical Characterization of the OI-Hierarchy”. In: *Information and Control* 71.1-2 (1986), pp. 1–32.

[104] P. de Groote and S. Pogodalla. “On the expressive power of Abstract Categorial Grammars: Representing context-free formalisms”. In: *Journal of Logic, Language and Information* 13.4 (2005), pp. 421–438.

[105] M. Dezani-Ciancaglini, E. Giovannetti, and U. de’Liguoro. “Intersection Types, Lambda-models and Böhm Trees”. In: *MSJ-Memoir Vol. 2 “Theories of Types and Proofs”*. Vol. 2. Mathematical Society of Japan, 1998, pp. 45–97.

[106] L. E. Dickson. “Finiteness of the Odd Perfect and Primitive Abundant Numbers with n Distinct Prime Factors”. English. In: *American Journal of Mathematics* 35.4 (1913), pp. 413–422.

[107] J. Doner. “Tree acceptors and some of their applications”. In: *Journal of Computer and System Sciences* 4.5 (1970), pp. 406–451.

[108] J. E. Doner. “Decidability of the weak second-order theory of two successors”. In: *Notices Amer. Math. Soc.* 12 (1965), pp. 365–468.

[109] J. Earley. “An Efficient Context-Free Parsing Algorithm”. In: *Communications of the ACM* 13.2 (Feb. 1970), pp. 94–102.

[110] T. Ehrhard. “Hypercoherences: A Strongly Stable Model of Linear Logic”. In: *Mathematical Structures in Computer Science* 3.4 (1993), pp. 365–385.

[111] T. Ehrhard. “Projecting Sequential Algorithms on Strongly Stable Functions”. In: *Ann. Pure Appl. Logic* 77.3 (1996), pp. 201–244.

[112] S. Eilenberg and J. B. Wright. “Automata in general algebras”. In: *Information and Control* 11.4 (1967), pp. 452–470.

[113] C. C. Elgot. “Algebraic theories and program schemes”. In: *Symposium on Semantics of Algorithmic Languages*. Vol. 188. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 1971, pp. 71–88.

[114] C. C. Elgot. “Decision problems of finite automata design and related arithmetics”. In: *Transactions of the American Mathematical Society* (1961), pp. 21–51.

[115] E. A. Emerson and C. S. Jutla. “Tree automata, mu-calculus and determinacy”. In: *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*. IEEE. 1991, pp. 368–377.

[116] J. Engelfriet. “Bottom-up and Top-down Tree Transformations - A Comparison”. In: *Mathematical Systems Theory* 9.3 (1975), pp. 198–231.

[117] J. Engelfriet. “Iterated Stack Automata and Complexity Classes”. In: *Inf. Comput.* 95.1 (1991), pp. 21–75.

[118] J. Engelfriet. “The complexity of Languages Generated by Attribute Grammars”. In: *SIAM J. Comput.* 15.1 (1986), pp. 70–86.

[119] J. Engelfriet and L. Heyker. “The String generating Power of Context-free Hypergraph Grammars”. In: *Journal of Computer and System Sciences* 43 (1991), pp. 328–360.

[120] J. Engelfriet and E. M. Schmidt. “IO and OI. I”. In: *Journal of computer and system sciences* 15 (1977), pp. 328–353.

[121] J. Engelfriet and E. M. Schnidt. “IO and OI. II”. In: *Journal of computer and system sciences* 16 (1978), pp. 67–99.

[122] J. Engelfriet and S. Skyum. “Copying Theorems”. In: *Inf. Process. Lett.* 4.6 (1976), pp. 157–161.

[123] J. Engelfriet and H. Vogler. “Macro Tree Transducers”. In: *J. Comput. Syst. Sci.* 31.1 (1985), pp. 71–146.

[124] Z. Ésik. “Algebras of Iteration Theories”. In: *J. Comput. Syst. Sci.* 27.2 (1983), pp. 291–303.

[125] A. Ferguson and J. Hughes. "Fast Abstract Interpretation Using Sequential Algorithms". In: *Static Analysis, Third International Workshop, WSA'93, Padova, Italy, September 22-24, 1993, Proceedings*. Ed. by P. Cousot, M. Falaschi, G. Filé, and A. Rauzy. Vol. 724. Lecture Notes in Computer Science. Springer, 1993, pp. 45–59.

[126] M. J. Fischer. "Grammars with macro-like productions". PhD thesis. Harvard University, 1968.

[127] M. J. Fischer and A. L. Rosenberg. "Real-Time Solutions of the Origin-Crossing Problem". In: *Mathematical Systems Theory* 2.3 (1968), pp. 257–263.

[128] T. Freeman and F. Pfenning. "Refinement Types for ML". In: *SIGPLAN Not.* 26.6 (May 1991), pp. 268–277.

[129] H. Friedman. "Equality between functionals". English. In: *Logic Colloquium*. Ed. by R. Parikh. Vol. 453. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 1975, pp. 22–37.

[130] G. Gazdar, J. Klein, G. Pullum, and I. Sag. *Generalized Phrase Structure Grammar*. Oxford: Blackwell, 1985.

[131] G. Gazdar. "Phrase structure grammar". In: *The nature of syntactic representation*. Springer, 1982, pp. 131–186.

[132] S. Ginsburg, S. Greibach, and J. Hopcroft. *Studies in abstract families of languages*. Memoirs 87. American Mathematical Society, 1969.

[133] S. Ginsburg and E. H. Spanier. "AFL with the Semilinear Property". In: *J. Comput. Syst. Sci.* 5.4 (1971), pp. 365–396.

[134] J. Girard. "The System F of Variable Types, Fifteen Years Later". In: *Theor. Comput. Sci.* 45.2 (1986), pp. 159–192.

[135] J.-Y. Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur". Thèse de doctorat d'état. Université Paris VII, 1972.

[136] K. Gödel. "Die vollständigkeit der axiome des logischen funktionenkalküls". In: *Monatshefte für Mathematik* 37.1 (1930), pp. 349–360.

[137] K. Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für mathematik und physik* 38.1 (1931), pp. 173–198.

[138] C. Gómez-Rodrguez, M. Kuhlmann, and G. Satta. "Efficient Parsing of Well-Nested Linear Context-Free Rewriting Systems". In: *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 2-4, 2010, Los Angeles, California, USA*. The Association for Computational Linguistics, 2010, pp. 276–284.

[139] S. A. Greibach. "Hierarchy Theorems for Two-Way Finite State Transducers". In: *Acta Inf.* 11 (1978), pp. 80–101.

[140] S. A. Greibach. “One Way Finite Visit Automata”. In: *Theor. Comput. Sci.* 6 (1978), pp. 175–221.

[141] C. Grellois and P. Melliès. “An Infinitary Model of Linear Logic”. In: *FoSSaCS 2015*. Vol. 9034. Lecture Notes in Computer Science. Springer, 2015, pp. 41–55.

[142] C. Grellois and P. Melliès. “Finitary semantics of linear logic and higher-order model-checking”. In: *CoRR* abs/1502.05147 (2015).

[143] C. Grellois and P. Melliès. “Indexed linear logic and higher-order model checking”. In: *Proceedings Seventh Workshop on Intersection Types and Related Systems, ITRS 2014, Vienna, Austria, 18 July 2014*. 2015, pp. 43–52.

[144] A. V. Groenink. “Surface without Structure”. PhD thesis. University of Utrecht, 1997.

[145] P. de Groote. “Towards a Montagovian Account of Dynamics”. In: *Proceedings of Semantics in Linguistic Theory XVI*. CLC Publications, 2007.

[146] P. de Groote. “Towards Abstract Categorial Grammars”. In: *Proceedings 39th Annual Meeting and 10th Conference of the European Chapter*. Ed. by A. for Computational Linguistic. Morgan Kaufmann Publishers, 2001, pp. 148–155.

[147] P. de Groote and E. Lebedeva. *On the Dynamics of Proper Names*. Tech. rep. INRIA, 2010.

[148] Y. Gurevich and L. Harrington. “Trees, Automata, and Games”. In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. STOC ’82. San Francisco, California, USA: ACM, 1982, pp. 60–65.

[149] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. “Collapsible Pushdown Automata and Recursion Schemes”. In: *LICS*. IEEE Computer Society, 2008, pp. 452–461.

[150] J. Hajic, E. Hajicova, P. Pajas, J. Panevova, and P. Sgall. *Prague Dependency Treebank 1.0*. https://catalog.ldc.upenn.edu/LDC2001T10. 2001.

[151] L. Henkin. “Completeness in the Theory of Types”. In: *Journal of Symbolic Logic* 15.2 (1950), pp. 81–91.

[152] G. G. Hillebrand. “Finite Model Theory in the Simply Typed Lambda Calculus”. PhD thesis. Providence, Rhode Island 02912: Department of Computer Science, Brown University, 1994.

[153] R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60.

[154] G. Hotz and G. Pitsch. “On parsing coupled-context-free languages”. In: *Theoretical Computer Science* 161 (1996), pp. 205–253.

[155] G. Huet. “Résolution d'équations dans des langages d'ordre 1,2,...,ω”. Thèse de Doctorat es Sciences Mathématiques. Université Paris VII, 1976.

[156] R. Huybregts. “The weak inadequacy of context-free phrase structure grammars”. In: *Van periferie naar kern* (1984), pp. 81–99.

[157] J. M. E. Hyland and C. L. Ong. “On Full Abstraction for PCF: I, II, and III”. In: *Inf. Comput.* 163.2 (2000), pp. 285–408.

[158] D. Janin and I. Walukiewicz. “On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic”. In: *CONCUR '96: Concurrency Theory*. Ed. by U. Montanari and V. Sassone. Vol. 1119. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 263–277.

[159] T. Jensen. “Abstract Interpretation in Logical Form”. PhD thesis. Imperial College, University of London, 1992.

[160] A. Joshi and Y. Schabes. *Tree-Adjoining Grammars*. 1997.

[161] A. K. Joshi. “Tree-adjoining grammars: How much context sensitivity is required to provide reasonable structural descriptions?” In: *Natural Language Parsing*. Ed. by D. Dowty, L. Karttunen, and A. M. Zwicky. Cambridge University Press, 1985, pp. 206–250.

[162] A. K. Joshi, L. S. Levy, and M. Takahashi. “Tree Adjunct Grammars”. In: *J. Comput. Syst. Sci.* 10.1 (1975), pp. 136–163.

[163] A. K. Joshi, V. K. Shanker, and D. J. Weir. “The converence of Mildly Context-Sensitive Grammar Formalisms”. In: *Foundational Issues in Natural Language Processing*. Ed. by P. Sells, S. M. Shieber, and T. Wasow. The MIT Press, 1991, pp. 31–81.

[164] J. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. “A Formally-Verified C Static Analyzer”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by S. K. Rajamani and D. Walker. ACM, 2015, pp. 247–259.

[165] G. Kahn and G. D. Plotkin. “Concrete Domains”. In: *Theor. Comput. Sci.* 121.1&2 (1993), pp. 187–277.

[166] M. Kanazawa. “The failure of Ogden's lemma for well-nested multiple context-free languages”. http://research.nii.ac.jp/ kanazawa/publications/ogden.pdf. 2014.

[167] M. Kanazawa. “A prefix-correct Earley recognizer for multiple context-free grammars”. In: *Proceedings of the Ninth International Workshop on Tree Adjoining Grammars and Related Formalisms*. Tübingen, Germany, 2008, pp. 49–56.

[168] M. Kanazawa. “Parsing and generation as Datalog queries”. In: *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2007, pp. 176–183.

[169] M. Kanazawa. “Second-Order ACGs as Hyperedge Replacement Grammars”. In: *Workshop on New Directions in Type-theoretic Grammars (NDTTG)*. 2007.

[170] M. Kanazawa. “The convergence of well-nested mildly context-sensitive grammar formalisms”. Invited Talk at The 14th conference on Formal Grammar, FG 2009. Bordeaux, France, 2009.

[171] M. Kanazawa. “The Pumping Lemma for Well-Nested Multiple Context-Free Languages”. In: *proceedings of DLT 2009*. Ed. by V. Diekert and D. Nowotka. Vol. 5583. LNCS. Springer, 2009, pp. 312–325.

[172] M. Kandulski. “The equivalence of Nonassociative Lambek Categorial Grammars and Context-Free Grammars”. In: *Mathematical Logic Quarterly* 34.1 (1988), pp. 41–52.

[173] T. Kasami. *An efficient recognition and syntax-analysis algorithm for context-free languages*. Tech. rep. Science Report AFCRL-65-758. Air Force Cambridge Research Laboratory, 1965.

[174] J. P. Kimbal. “Formal Theory of Grammar”. In: Foundations of modern linguistics. Prentice-Hall, 1973. Chap. 6.

[175] J. Kirman and S. Salvati. “On the Complexity of Free Word Orders”. In: *Formal Grammar - 17th and 18th International Conferences, FG 2012, Opole, Poland, August 2012, Revised Selected Papers, FG 2013, Düsseldorf, Germany, August 2013. Proceedings*. 2013, pp. 209–224.

[176] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3. BRICS, Department of Computer Science, Aarhus University. 2001.

[177] S. C. Kleene. “Representation of Events in Nerve Nets and Finite Automata”. In: *Automata Studies*. Ed. by C. E. Shannon and J. McCarthy. Vol. 34. Annals of Mathematics Studies. Princeton University Press, 1956, pp. 3–42.

[178] S. C. Kleene. *Representation of events in nerve nets and finite automata*. Tech. rep. DTIC Document, 1951.

[179] T. Knapik, D. Niwinski, and P. Urzyczyn. “Deciding Monadic Theories of Hyperalgebraic Trees”. In: *TLCA*. 2001, pp. 253–267.

[180] T. Knapik, D. Niwinski, and P. Urzyczyn. “Higher-Order Pushdown Trees Are Easy”. In: *FOSSCA 2002*. Springer, 2002, pp. 205–222.

[181] T. Knapik, D. Niwinski, P. Urzycyzn, and I. Walukiewicz. “Unsafe Grammars and Panic Automata”. In: *ICALP 05*. Ed. by Springer. LNCS 3580. 2005, pp. 1450–1461.

[182] T. Knapik, D. Niwinski, and P. Urzyczyn. “Higher-order pushdown trees are easy”. In: *Proc. FoSSaCS’02*. Ed. by Springer. Vol. 2303. 2002, pp. 205–222.

[183] N. Kobayashi. “Model Checking Higher-Order Programs”. In: *J. ACM* 60.3 (2013), p. 20.

[184] N. Kobayashi. “Types and Recursion Schemes for Higher-Order Program Verification”. In: *APLAS*. Vol. 5904. LNCS. 2009, pp. 2–3.

[185] N. Kobayashi and L. Ong. “A Type System Equivalent to Modal Mu-Calculus Model Checking of Recursion Schemes”. In: *Proceedings of 24th Annual IEEE Symposium on Logic in Computer Science (LICS 2009), Los Angeles*. 2009, pp. 179–188.

[186] G. M. Kobele. “Idioms and extended transducers”. In: *Proceedings of the 11th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+11)*. 2012, pp. 153–161.

[187] G. M. Kobele. “Generating Copies: An Investigation into Structural Identity in Language and Grammar”. PhD thesis. University of California Los Angeles, 2006.

[188] H.-P. Kolb, J. Michaelis, U. Mönnich, and F. Morawietz. “An operational and denotational approach to non-context-freeness”. In: *Theoretical Computer Science* 293.2 (2003), pp. 261–289.

[189] M. Kracht. *The Mathematics of Language*. Vol. 63. Studies in Generative Grammar. Mouton De Gruyter, 2003.

[190] J.-L. Krivine. “A call-by-name lambda-calculus machine”. In: *Higher-Order and Symbolic Computation* 20.3 (2007), pp. 199–207.

[191] M. T. Kromann. “The Danish Dependency Treebank and the DTAG treebank tool”. In: *In Proceedings of the Second Workshop on Treebanks and Linguistic Theories (TLT 2003*. 2003, pp. 14–15.

[192] M. Kuhlmann and M. Möhl. “Mildly Context-Sensitive Dependency Languages”. In: *ACL 2007, Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, June 23-30, 2007, Prague, Czech Republic*. 2007.

[193] M. Kuhlmann and M. Möhl. “The string-generative capacity of regular dependency languages”. In: *FG 2007*. 2007.

[194] M. Kuhlmann and J. Nivre. “Mildly Non-Projective Dependency Structures”. In: *ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, Sydney, Australia, 17-21 July 2006*. Ed. by N. Calzolari, C. Cardie, and P. Isabelle. The Association for Computer Linguistics, 2006.

[195] J. Laird. “A Fully Abstract Bidomain Model of Unary FPC”. In: *Typed Lambda Calculi and Applications*. Ed. by M. Hofmann. Vol. 2701. Lecture Notes in Computer Science. Springer, 2003, pp. 211–225.

[196] J. Laird. *Games and Sequential Algorithms*. Unpublished note. 2001.

[197] F. Lamarche. "Sequentiality, games and linear logic". Manuscript. 1992.

[198] J. Lambek. "On the calculus of syntactic types". In: *Studies of Language and its Mathematical Aspects, Proceedings of the 12th Symposium of Applied Mathematics*. Ed. by R. Jakobson. 1961, pp. 166–178.

[199] J. Lambek. "The mathematics of sentence structure". In: *American Mathematical Monthly* 65 (1958), pp. 154–170.

[200] M. Latteux. "Cônes Rationnels Commutativement Clos". In: *ITA* 11.1 (1977), pp. 29–51.

[201] F. W. Lawvere. "Functorial Semantics of Algebraic Theories". PhD thesis. Columbia University, 1963.

[202] A. Lecomte. "A Computational Approach to Minimalism". In: *Proceedings of ICON-2003, International Conference on Natural Language*. Central Institute of Indian Languages, 2003, pp. 20–31.

[203] A. Lecomte. "Derivations as Proofs : a Logical Approach to Minimalism". In: *Proceedings of CG 2004*. 2004.

[204] A. Lecomte and C. Retoré. "Extending Lambek grammars: a logical account of minimalist grammars". In: *Proceedings of the 39th meeting of the Association for Computational Linguistics,ACL 2001*. 2001, pp. 354–361.

[205] A. Lecomte and C. Retoré. "Towards a Minimal Logic for Minimalist Grammars: a tranformational use of Lambek calculus". In: *Formal Grammar 99*. 1999.

[206] R. Loader. "Finitary PCF is not decidable". In: *Theor. Comput. Sci.* 266.1-2 (2001), pp. 341–364.

[207] R. Loader. "The Undecidability of λ-definability". In: *Logic, Meaning and Computation: Essays in memory of Alonzo Church*. Ed. by C. A. Anderson and M. Zeleny. Kluwer, 2001, pp. 331–342.

[208] G. M. M. Kobele. "Parsing Ellipsis". manuscript. 2007.

[209] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[210] Z. Manna. "The correctness of programs". In: *Journal of Computer and System Sciences* 3.2 (1969), pp. 119–127.

[211] A. N. Maslov. "Multilevel Stack Automata". In: *Probl. Peredachi Inf.* 12.1 (1976), pp. 55–62.

[212] A. Maslov. "The hierarchy of indexed languages of an arbitrary level". In: *Soviet Math. Dokl.* Vol. 15. 4. 1974, pp. 1170–1174.

[213] J. Mezei and J. Wright. "Algebraic Automata and Context-Free Sets". In: *Informcation and Control* 11 (1967), pp. 3–29.

[214] J. Michaelis and M. Kracht. “Semilinearity as a syntactic invariant”. English. In: *Logical Aspects of Computational Linguistics*. Ed. by C. Retoré. Vol. 1328. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 329–345.

[215] R. Milner. “LCF: A way of doing proofs with a machine”. English. In: *Mathematical Foundations of Computer Science 1979*. Ed. by J. Bevá. Vol. 74. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1979, pp. 146–159.

[216] R. Milner. *Models of LCF*. Tech. rep. Memo AIM-186. Stanford Artificial Intelligence Laboratory, 1973.

[217] R. Milner. “A Theory of Type Polymorphism in Programming”. In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375.

[218] J. G. de Miranda. “Structures Generated by Higher-Order Grammars and the Safety Constraint”. PhD thesis. Oxford University, 2006.

[219] R. Montague. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, CT, 1974.

[220] F. Morawietz. *Two-Step Approaches ot Natural Language Formalisms*. Studies in Generative Grammar. Berlin · New York: Mouton de Gruyter, 2003.

[221] A. A. Muchnik. “Games on Infinite Trees and Automata with Dead Ends”. In: *Semiotics and Information* 24 (1984). in Russian, pp. 17–44.

[222] D. E. Muller and P. E. Schupp. “The Theory of Ends, Pushdown Automata, and Second-Order Logic”. In: *Theor. Comput. Sci.* 37 (1985), pp. 51–75.

[223] D. E. Muller, A. Saoudi, and P. E. Schupp. “Alternating automata, the weak monadic theory of the tree, and its complexity”. In: *Automata, Languages and Programming*. Springer, 1986, pp. 275–283.

[224] D. E. Muller, A. Saoudi, and P. E. Schupp. “Alternating automata, the weak monadic theory of trees and its complexity”. In: *Theoretical Computer Science* 97.2 (1992), pp. 233–244.

[225] R. Muskens. “Lambda Grammars and the Syntax-Semantics Interface”. In: *Proceedings of the Thirteenth Amsterdam Colloquium*. Ed. by R. van Rooy and M. Stokhof. Amsterdam, 2001, pp. 150–155.

[226] J. Myhill. *Finite automata and the representation of events*. Tech. rep. WADC TR-57-624. Wright Patterson Air Force Base, Ohio, USA, 1957.

[227] M.-J. Nederhof. “Generalized left-corner parsing”. In: *Proceedings of the sixth conference on European chapter of the Association for Computational Linguistics*. Utrecht, The Netherlands: Association for Computational Linguistics, 1993, pp. 305–314.

[228] M.-J. Nederhof. “The computational complexity of the correct-prefix property for TAGs”. In: *Comput. Linguist.* 25.3 (1999), pp. 345–360.

[229] A. Nerode. “Linear Automaton Transformations”. In: *Proceedings of the American Mathematical Society*. Vol. 9. American Mathematical Society, 1958, pp. 541–544.

[230] M. Nivat. “On the interpretation of recursive polyadic program schemes”. In: *Symp. Mathematica*. 15. 1975, pp. 255–281.

[231] M. Nivat. “Transductions des langages de Chomsky”. Thèse d’état. Annales de l’institut Fourier, 1968.

[232] W. Ogden. “A helpful result for proving inherent ambiguity”. In: *Theory of Computing Systems* 2.3 (1968), pp. 191–194.

[233] C. L. Ong. “On Model-Checking Trees Generated by Higher-Order Recursion Schemes”. In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006, pp. 81–90.

[234] V. Padovani. “Decidability of All Minimal Models”. In: *Types for Proofs and Programs, International Workshop TYPES’95, Torino, Italy, June 5-8, 1995, Selected Papers*. Ed. by S. Berardi and M. Coppo. Vol. 1158. Lecture Notes in Computer Science. Springer, 1995, pp. 201–215.

[235] V. Padovani. “Decidability of fourth-order matching”. In: *Mathematical Structures in Computer Science* 10.3 (2000), pp. 361–372.

[236] V. Padovani. “Filtrage d’odre supérieur”. Thèse de doctorat. Université de Paris 7, 1994.

[237] M. A. Palis and S. M. Shende. “Pumping Lemmas for the Control Language Hierarchy”. In: *Mathematical System Theory* 28.3 (1995), pp. 199–213.

[238] B. H. Partee. “Richard Montague (1930 - 1971)”. In: *Encyclopedia of Language and Linguistics*. Ed. by K. Brown. Vol. 8. 2nd edition. Elsevier, 2006, pp. 255–257.

[239] P. Parys. “Collapse Operation Increases Expressive Power of Deterministic Higher Order Pushdown Automata”. In: *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011*. 2011, pp. 603–614.

[240] M. Pentus. “Lambek Grammars Are Context Free”. In: *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*. 1993, pp. 429–433.

[241] P. S. Peters and R. W. Ritchie. “On the generative power of transformational grammars”. In: *Information Sciences* 6 (1973), pp. 49–83.

[242] G. Plotkin. “Lambda-definability and logical relations”. In: *To H.B.Curry : essays on combinatory logic, lambda calculus, and formalism*. Ed. by J. Seldin and J. Hindley. Academic Press, 1980, pp. 363–373.

[243] G. D. Plotkin. “LCF Considered as a Programming Language”. In: *Theor. Comput. Sci.* 5.3 (1977), pp. 223–255.

[244] C. Pollard and I. A. Sag. *Head-driven phrase structure grammar*. University of Chicago Press, 1994.

[245] G. K. Pullum and G. G. "Natural languages and Context-Free Languages". In: *Linguistics and Philosophy* 4 (1982), pp. 471–504.

[246] P. C. R. Amadio. *Domains and lambda-calculi*. Cambridge Tracts in Theoretical Computer Science 46. Cambridge University Press, 1996.

[247] M. O. Rabin. "Decidability of Second-Order Theories and Automata on Infinite Trees". In: *Transactions of the AMS* 141 (1969), pp. 1–23.

[248] M. O. Rabin. "Decidaiblity of Second-Order Theories and Automata on Infinite Trees". In: *Transaction of the American Mathematical Society* 141 (1969), pp. 1–35.

[249] M. O. Rabin. *Automata on infinite objects and Church's problem*. Vol. 13. American Mathematical Soc., 1972.

[250] D. Radzinski. "Chinese number-names, tree adjoining languages, and mild context-sensitivity". In: *Comput. Linguist.* 17 (3 1991), pp. 277–299.

[251] O. Rambow and G. Satta. "A rewritting system for free word order syntax that is non-local and mildly context sensitive". In: *Current Issues in Mathematical Linguistics*. Ed. by C. Martín-Vide. Vol. 56. North-Holland Linguistics Series. Elsevier-North Holland, Amsterdam, 1994.

[252] O. Rambow and G. Satta. *A Two-Dimensional Hierarchy for Parallel Rewriting Systems*. Tech. rep. IRCS-94-02. University of Pennsylvania Institute for Research in Cognitive Science, 1994.

[253] O. Rambow and G. Satta. "Independent Parallelism in Finite Copying Parallel Rewriting Systems". In: *Theoretical Computer Science* 223.1-2 (1999), pp. 87–120.

[254] A. Ranta. *Type-theoretical Grammar*. Vol. 1. Indices. Oxford Science Publisher, 1994.

[255] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[256] J. Rogers. *A Descriptive Approach to Language-Theoretic Complexity*. Studies in Logic, Language & Information. distributed by the University of Chicago Press. CSLI publications, 1998.

[257] J. Rogers. "A Model-Theoretic Framework for Theories of Syntax". In: *34th Annual Meeting of the Association for Computational Linguistics, 24-27 June 1996, University of California, Santa Cruz, California, USA, Proceedings*. Ed. by A. K. Joshi and M. Palmer. Morgan Kaufmann Publishers / ACL, 1996, pp. 10–16.

[258] J. Rogers. "wMSO theories as grammar formalisms". In: *Theor. Comput. Sci.* 293.2 (2003), pp. 291–320.

[259] J. Rogers. “Wrapping of Trees”. In: *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics, 21-26 July, 2004, Barcelona, Spain.* Ed. by D. Scott, W. Daelemans, and M. A. Walker. ACL, 2004, pp. 558–565.

[260] B. Rozoy. “The Dyck language D'^*_1 is not generated by any matrix grammar of finite index”. In: *Information and Computation* 74.1 (1987), pp. 64–89.

[261] Y. Schabes and A. K. Joshi. “An Earley-type parsing algorithm for Tree Adjoining Grammars”. In: *Proceedings of the 26th annual meeting on Association for Computational Linguistics.* Buffalo, New York: Association for Computational Linguistics, 1988, pp. 258–269.

[262] M. Schmidt-SchauSS. “Decidability of Behavioural Equivalence in Unary PCF”. In: *Theor. Comput. Sci.* 216.1-2 (1999), pp. 363–373.

[263] S. Schmitz. “On the Computational Complexity of Dominance Links in Grammatical Formalisms”. In: *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden.* Ed. by J. Hajic, S. Carberry, and S. Clark. The Association for Computer Linguistics, 2010, pp. 514–524.

[264] H. Schwichtenberg. “Definierbare funktionen im lambda-kalkul mit typen”. In: *Archiv Logik Grundlagenforsch* 17 (1976), pp. 113–114.

[265] D. Scott. “Continuous lattices”. In: *Toposes, Algebraic Geometry and Logic.* Ed. by F. Lawvere. Vol. 274. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 1972, pp. 97–136.

[266] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. “On multiple context free grammars”. In: *Theoretical Computer Science* 88.2 (1991), pp. 191–229.

[267] G. Sénizergues. “L(A)=L(B)? decidability results from complete formal systems”. In: *Theor. Comput. Sci.* 251.1-2 (2001), pp. 1–166.

[268] S. M. Shieber. “Evidence Against the Context-Freeness of Natural Language”. In: *Linguistics and Philosophy* 8 (1985). Reprinted in Walter J. Savitch, Emmon Bach, William Marsh, and Gila Safran-Navah, eds., The Formal Complexity of Natural Language, pages 320–334, Dordrecht, Holland: D. Reidel Publishing Company, 1987. Reprinted in Jack Kulas, James H. Fetzer, and Terry L. Rankin, eds., Philosophy, Language, and Artificial Intelligence, pages 79–92, Dordrecht, Holland: Kluwer Academic Publishers, 1988, pp. 333–343.

[269] A. K. Simpson and G. D. Plotkin. “Complete Axioms for Categorical Fixed-Point Operators”. In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000.* IEEE Computer Society, 2000, pp. 30–41.

[270] A. Sorokin. "Pumping Lemma and Ogden Lemma for Displacement Context-Free Grammars". In: *Developments in Language Theory - 18th International Conference, DLT 2014, Ekaterinburg, Russia, August 26-29, 2014. Proceedings*. Ed. by A. M. Shur and M. V. Volkov. Vol. 8633. Lecture Notes in Computer Science. Springer, 2014, pp. 154–165.

[271] E. H. Spanier. *Algebraic Topology*. Corrected reprint. Springer-Verlag, 1981.

[272] E. P. Stabler. "Derivational Minimalism". In: *Logical Aspects of Computational Linguistics, First International Conference, LACL '96, Nancy, France, September 23-25, 1996, Selected Papers*. Ed. by C. Retoré. Vol. 1328. Lecture Notes in Computer Science. Springer, 1996, pp. 68–95.

[273] E. P. Stabler. "Varieties of crossing dependencies: structure dependence and mild context sensitivity". In: *Cognitive Sience* 28 (2004), pp. 699–720.

[274] R. Statman. "Completeness, Invariance and λ-definability". In: *Journal of Symbolic Logic* 47.1 (1982), pp. 17–26.

[275] R. Statman and G. Dowek. *On Statman's finite completeness theorem*. Tech. rep. CMU-CS-92-152. University of Carnegie Mellon, 1992.

[276] R. Statman. "On the [lambda]Y calculus". In: *Annals of Pure and Applied Logic* 130.1-3 (2004). Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS), pp. 325–337.

[277] P. Staudacher. "New Frontiers Beyond Context-Freeness: DI-Grammars And DI-Automata". In: *Sixth Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, 21-23 April 1993, Utrecht, The Netherlands*. Ed. by S. Krauwer, M. Moortgat, and L. des Tombe. The Association for Computer Linguistics, 1993, pp. 358–367.

[278] C. Stirling. "A Game-Theoretic Approach to Deciding Higher-Order Matching". In: *ICALP (2)*. Ed. by M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener. Vol. 4052. Lecture Notes in Computer Science. Springer, 2006, pp. 348–359.

[279] C. Stirling. "Decidability of higher order matching". In: *Logical Methods in Computer Science* 5.3 (2009).

[280] W. W. Tait. "Intensional Interpretations of Functionals of Finite Type I". In: *The Journal of Symbolic Logic* 32.2 (1967), pp. 198–212.

[281] T. C. development team. *The Coq proof assistant reference manual*. Version 8.4. INRIA. 2012.

[282] T. Terao and N. Kobayashi. "A ZDD-Based Efficient Higher-Order Model Checking Algorithm". In: *APLAS 2014*. Vol. 8858. Lecture Notes in Computer Science. Springer, 2014, pp. 354–371.

[283] Terese. *Term Rewriting Systems*. Vol. 55. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[284] J. W. Thatcher and J. B. Wright. "Generalized finite automata". In: *Notices Amer. Math. Soc.* (1965). Abstract No 65T-649.

[285] J. Thatcher and J. Wright. "Generalized finite automata theory with an application to a decision problem of second-order logic". English. In: *Mathematical systems theory* 34.1 (1968), pp. 57–81.

[286] B. A. Trakhtenbrot. "Finite automata and monadic second order logic". In: *Siberian Math. J* 3 (1962), pp. 101–131.

[287] P. Urzyczyn. "The Emptiness Problem for Intersection Types." In: *J. Symb. Log.* 64.3 (1999), pp. 1195–1215.

[288] K. N. Verma and J. Goubault-Larrecq. "Karp-Miller Trees for a Branching Extension of VASS". In: *Discrete Mathematics & Theoretical Computer Science* 7.1 (2005), pp. 217–230.

[289] K. Vijay-Shanker, D. J. Weir, and A. K. Joshi. "Characterizing Structural Descriptions produced by Various Grammatical Formalisms". In: *25th Annual Meeting of the Association for Computational Linguistics, Stanford University, Stanford, California, USA, July 6-9, 1987.* 1987, pp. 104–111.

[290] I. Walukiewicz. "Monadic second-order logic on tree-like structures". In: *Theor. Comput. Sci.* 275.1-2 (2002), pp. 311–346.

[291] M. Wand. "A Concrete Approach to Abstract Recursion Definitions". In: *ICALP*. 1972, pp. 331–341.

[292] M. Wand. "Final Algebra Semantics and Data Type Extensions". In: *J. Comput. Syst. Sci.* 19.1 (1979), pp. 27–44.

[293] D. J. Weir. "A geometric hierarchy beyond context-free languages". In: *Theoretical Computer Science* 104.2 (1992), pp. 235–261.

[294] D. J. Weir. "Linear Context-Free Rewriting Systems and Deterministic Tree-Walking Transducers." In: *ACL*. 1992, pp. 136–143.

[295] T. Wilke. "An Eilenberg Theorem for Infinity-Languages". In: *ICALP91*. 1991, pp. 588–599.

[296] E. S. Y. Bar-Hillel M. Perles. "On formal properties of simple phrase structure grammars". In: *Zeitschrift für Phonetik Sprachwissenschaft, und Kommunikationsforschung* 14 (1961).

[297] D. Younger. "Recognition and parsing of context-free languages in time n^3". In: *Information and Control* 10 (1967), pp. 572–597.

[298] W. Zielonka. "Infinite games on finitely coloured graphs with applications to automata on infinite trees". In: *Theoretical Computer Science* 200.12 (1998), pp. 135–183.

Personal bibliography

[S1] A. Ball, P. Bourreau, É. Kien, and S. Salvati. "Building PMCFG Parsers as Datalog Program Transformations". In: *the 8th International Conference on Logical Aspects of Computational Linguistics (LACL 2014)*. Vol. 7351. FOLLI-LNCS. Springer, 2014.

[S2] P. Bourreau and S. Salvati. "A Datalog Recognizer for Almost Affine λ-CFGs". In: *The Mathematics of Language 12*. Ed. by M. Kanazawa, M. Kracht, and H. Seki. Vol. 6878. Lecture Notes in Computer Science. Springer, 2011, pp. 21–38.

[S3] P. Bourreau and S. Salvati. "Game Semantics and Uniqueness of Type Inhabitance in the Simply-Typed lambda-Calculus". In: *10th International Conference on Typed Lambda Calculi and Applications (TLCA 2011)*. Vol. 6690. LNCS. Springer, 2011, pp. 61–75.

[S4] L. Clément, J. Kirman, and S. Salvati. "A logical approach to grammar description". In: *Journal of Language Modelling* 3.1 (2015).

[S5] P. de Groote, B. Guillaume, and S. Salvati. "Vector Addition Tree Automata". In: *19-th IEEE symposium on Logic in Computer Science* (2004), pp. 63–74.

[S6] M. Kanazawa, G. M. Kobele, J. Michaelis, S. Salvati, and R. Yoshinaka. "The Failure of the Strong Pumping Lemma for Multiple Context-Free Languages". In: *Theory Comput. Syst.* 55.1 (2014), pp. 250–278.

[S7] M. Kanazawa and S. Salvati. "Generating control languages with Abstract Categorial Grammars". In: *Proceedings of the 12th conference on Formal Grammar (FG 2007), Dublin, Ireland*. Ed. by L. Kallmeyer, P. Monachesi, G. Penn, and G. Satta. To appear, ISSN 1935-1569. CSLI Publications, 2007.

[S8] M. Kanazawa and S. Salvati. "MIX is not a tree-adjoining language". In: *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2012, pp. 666–674.

[S9] M. Kanazawa and S. Salvati. "The Copying Power of Well-Nested Multiple Context-Free Grammars". In: *4th International Conference on Language and Automata Theory and Applications (LATA 2010)*. Ed. by A.-H. Dediu, H. Fernau, and C. Martn-Vide. Vol. 6031. Lecture Notes in Computer Science, 2010, pp. 344–355.

[S10] M. Kanazawa and S. Salvati. "The String-Meaning Relations Definable by Lambek Grammars and Context-Free Grammars". In: *18th conference on Formal Grammar (FG 2013)*. Vol. 8036. LNCS. Springer, 2013, pp. 191–208.

[S11] J. Kirman and S. Salvati. "On the Complexity of Free Word Orders". In: *18th conference on Formal Grammar (FG 2013)*. Vol. 8036. LNCS. Springer, 2013, pp. 209–224.

[S12] G. M. Kobele and S. Salvati. “The IO and OI hierarchies revisited”. In: *Information and Computation* 243 (2015), pp. 205–221.

[S13] C. Retoré and S. Salvati. “A Faithful Representation of Non-Associative Lambek Grammars in Abstract Categorial Grammars”. In: *Journal of Logic Language and Information* 19.2 (2010), pp. 185–200.

[S14] S. Salvati and I. Walukiewicz. “Krivine Machines and Higher-Order Schemes”. In: *ICALP (2)*. Vol. 6756. LNCS. 2011, pp. 162–173.

[S15] S. Salvati and I. Walukiewicz. “Typing Weak MSOL Properties”. In: *FoSSaCS 2015*. Vol. 9034. Lecture Notes in Computer Science. Springer, 2015, pp. 343–357.

[S16] S. Salvati and I. Walukiewicz. “Using models to model-check recursive schemes”. In: *Logical Methods In Computer Science* (2015).

[S17] S. Salvati. “Encoding second order string ACG with Deterministic Tree Walking Transducers.” In: *Proceedings FG 2006: the 11th conference on Formal Grammars*. Ed. by S. Wintner. FG Online Proceedings. CSLI Publications, 2007, pp. 143–156.

[S18] S. Salvati. “Minimalist Grammars in the Light of Logic”. In: *Logic and grammar - Essays Dedicated to Alain Lecomte on the Occasion of His 60th Birthday*. Ed. by C. R. Sylvain Pogodalla Myriam Quatrini. Vol. 6700. LNCS/LNAI. Springer Berlin/Heidelberg, 2011, pp. 81–117.

[S19] S. Salvati. “MIX is a 2-MCFL and the word problem in is captured by the IO and the OI hierarchies”. In: *Journal of Computer and System Sciences* 81.7 (2015), pp. 1252–1277.

[S20] S. Salvati. “Non-linear Second Order Abstract Categorial Grammars and Deletion”. In: *NLCS’15. Third Workshop on Natural Language and Computer Science*. Ed. by M. Kanazawa, L. S. Moss, and V. de Paiva. Vol. 32. EasyChair Proceedings in Computing. EasyChair, 2015, pp. 64–72.

[S21] S. Salvati. “On the membership problem for non-linear ACGs”. In: *Journal of Logic Language and Information* 19.2 (2010), pp. 163–183.

[S22] S. Salvati. “Problèmes de filtrage et problèmes d’analyse pour les grammaires catégorielles abstraites”. PhD thesis. Institut National Polytechnique de Lorraine, 2005.

[S23] S. Salvati. “Recognizability in the Simply Typed Lambda-Calculus”. In: *16th Workshop on Logic, Language, Information and Computation*. Vol. 5514. Lecture Notes in Computer Science. Tokyo Japan: Springer, 2009, pp. 48–60.

[S24] S. Salvati. “Syntactic Descriptions: a Type System for Solving Matching Equations in the Linear λ-Calculus”. In: *proceedings of the 17th International Conference on Rewriting Techniques and Applications*. 2006, pp. 151–165.

[S25] S. Salvati, G. Manzonetto, M. Gehrke, and H. Barendregt. "Loader and Urzyczyn are Logically Related". In: *39th International Colloquium on Automata, Languages and Programming (ICALP (2) 2012)*. Lecture Notes in Computer Science. Springer, 2012, pp. 364–376.

[S26] S. Salvati and I. Walukiewicz. "A Model for Behavioural Properties of Higher-order Programs". In: *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*. Ed. by S. Kreutzer. Vol. 41. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 229–243.

[S27] S. Salvati and I. Walukiewicz. "Evaluation is MSOL-compatible". In: *33rd International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013)*. Vol. 24. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[S28] S. Salvati and I. Walukiewicz. "Krivine Machines and Higher-Order Schemes". In: *38th International Colloquium on Automata, Languages and Programming (ICALP (2) 2011)*. Ed. by L. Aceto, M. Henzinger, and J. Sgall. Vol. 6756. Lecture Notes in Computer Science. Springer, 2011, pp. 162–173.

[S29] S. Salvati and I. Walukiewicz. "Krivine machines and higher-order schemes". In: *Inf. Comput.* 239 (2014), pp. 340–355.

[S30] S. Salvati and I. Walukiewicz. "Simply typed fixpoint calculus and collapsible pushdown automata". In: *Mathematical Structures in Computer Science* FirstView (2015), pp. 1–47.

[S31] S. Salvati and I. Walukiewicz. "Using Models to Model-Check Recursive Schemes". In: *11th International Conference on Typed Lambda Calculi and Applications (TLCA 2013)*. Vol. 7941. LNCS. Springer, 2013, pp. 189–204.