Mémoire d'habilitation à diriger des recherches

Automata, Logics and Games for Infinite Trees

Arnaud Carayol

Patricia Bouyer	*reviewer*
Didier Caucal	*examiner*
Jérôme Leroux	*examiner*
Jean-Marc Talbot	*examiner*
Wolfgang Thomas	*reviewer*
Sophie Tison	*reviewer*

LABORATOIRE D'INFORMATIQUE GASPARD-MONGE
Sous la co-tutelle de :
CNRS
ÉCOLE DES PONTS PARISTECH
ESIEE PARIS
UPEM • UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE

École Doctorale 532
Mathématiques et STIC

UNIVERSITÉ PARIS-EST

Mémoire d'habilitation à diriger des recherches

Automata, Logics and Games for Infinite Trees

Arnaud Carayol

Noisy-Champs, 2019

Contents

Acknowledgments

This document is dedicated to the memory of Zoltán Ésik.

The full version of the acknowledgments will appear in the printed version of this document.

The LaTeX style for this document is shamelessly copied with permission from Fabian Reiter (see his PhD thesis [125]).

1 Introduction

This document presents a selection of the contributions obtained since my PhD thesis in 2006 at IRISA in Rennes. This research was conducted during my post-doc in RWTH Aachen under the guidance of Wolfgang Thomas (September 2006 - September 2007), my very brief stay in LIAFA (now IRIF) (September 2007 - October 2007) and since November 2007 at LIGM in the *Modèles et algorithmes* team.

All my work was done with colleagues and friends. My main co-authors are Zoltan Ésik, Matthew Hague, Christof Loeding and Olivier Serre with whom I wrote the majority of the material presented here. I also collaborated with Luca Aceto, Laurent Braud, Christopher Broadbent, Stefan Göller, Axel Haddad, Anna Ingólfsdóttir, Antoine Meyer, Cyril Nicaud, Damian Niwiński, Luke Ong, Mikhaela Slaats and Igor Walukiewicz. They will be more properly acknowledge in each chapter. After this introduction, I will drop the "I" for the more honest "We".

My research is in algorithmic model theory. This domain studies the algorithmic properties of infinite structures (graphs, trees, linear orders, relational structures, ...) that arise in theoretical computer science. As these infinite structures are described by finite objects (automata, functional programs, systems of equations, ...), they are amenable to automated treatment. Logic provides a formalism to express properties of these structures. My line of research is to understand what type of properties can be checked on these structures, how these different structures relate to one another and how they can automatically be transformed or generated. Algorithmic model theory has a strong connection with program verification which is the most natural sources of models and of problems.

The tools of algorithmic model theory come from automata theory, logic and games. In the case of the monadic second order logic which is the logic of choice in my work, these three areas are strongly connected providing different angles of attack for each problem.

To give some structure to this document, I organized my contributions into two parts. Chapter 3 presents my work on recursion schemes which are an abstract model for functional programs. Chapter 4 contains my contributions related to the model of tree automaton on infinite trees and its extensions. Tree automata on infinite trees are a central tool in algorithmic model theory and are at the heart of the

above mention connexion between automata, logic and games. Each chapter contain an introduction providing the motivation and the summary of my contributions. Chapter 5 presents some directions for future research.

The references of articles I co-authored appear in red in this document.

The results published in [51, 38, 47, 37, 36, 39, 50, 40] did not find a place in this presentation. However I would like to briefly present two of these results which I particularly like although they are not in my field of expertize. I am fortunate to be part of a research team at LIGM covering a large spectrum of topics including analytic combinatorics, bio-informatics, combinatorics on words, algorithmic geometry, theory of databases, ... This proximity with other fields has been and continues to be a tremendous asset. These two results certainly would not have seen the day of light in a more standard environment.

The first result obtained with Cyril Nicaud in [51] gives an algorithm to uniformly generate a deterministic and complete finite automaton (DFA) in which all states are accessible from the initial state. The algorithm has an expected run time in $\mathcal{O}(n^{3/2})$ to generate an accessible automaton of size n. It is easy to generate uniformly a DFA with n states over an alphabet with k letters. The difficulty comes from the accessibility constraint which is not local.

The idea of our algorithm is to randomly generate a DFA $\mathcal{A}$ with $\beta_k n$ states where $\beta_k > 1$ is a well-chosen constant depending on the size of the k of the alphabet. Then we extract the sub-DFA $\overline{\mathcal{A}}$ containing the states reachable from the initial state. It is easy to see that $\overline{\mathcal{A}}$ is an accessible DFA which is uniformly chosen amongst the accessible DFA with the same number of states. We reiterate this process until the number of states of $\overline{\mathcal{A}}$ is exactly n. We showed that if we take $\beta_k = \left(1 + \frac{1}{k} W_0(-ke^{-k})\right)^{-1}$, the expected number of tries is in $\mathcal{O}(\sqrt{n})$. Indeed, as hinted by Figure 1.1, the distribution of the size of $\overline{\mathcal{A}}$ resemble a Gaussian law which we proved, in the appropriate probabilistic setting. Also, we proved that expected value of the size of $\overline{\mathcal{A}}$ for a random DFA $\mathcal{A}$ of size m is equivalent to $\beta_k^{-1} m$ with a standard deviation equivalent to $\sigma_k \sqrt{m}$ where σ_k is a constant depending on the size of the alphabet.

The classical Lambert function W_0 is implicitly defined by $W_0(x)e^{W_0(x)} = x$ and $W_0(x) \geqslant -1$ for all $x \geqslant -e^{-1}$

n	1000	10000
$\mathbb{E}[X_n]$(k=2)	796.663	7967.41
$\mathbb{E}[X_n]$(k=3)	940.489	9404.40
$\mathbb{E}[X_n]$(k=4)	980.137	9801.89

k	2	3	4
β_k^{-1}	0.79681	0.94047	0.98017

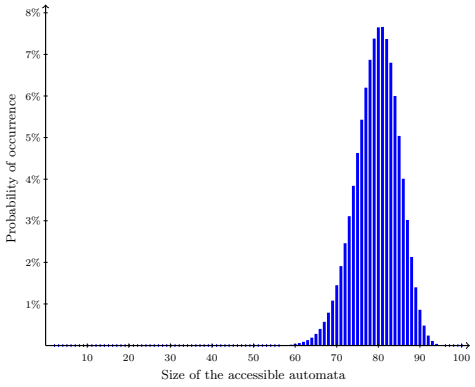


Figure 1.1. On the top left, an approximation of the average size X_n of the accessible automaton based on 10000 randomly generated structures DFAs over an alphabet of size k. On the bottom left, the values of the constant $\beta_k^{-1} = 1 + \frac{1}{k} W_0(-ke^{-k})$ for different values of k. On the right, the graphical representation of X_{100}.

The second result obtained with Stefan Göller in [53, 40] is in word combinatorics. This result concerns unavoidable patterns in infinite words. More precisely it con-

cerns patterns that must appear in every sufficiently long word over an arbitrary alphabet.

A pattern is a finite word over some set of pattern variables. A pattern matches a word if the word can be obtained by substituting each variable appearing in the pattern by a non-empty word. The pattern xx matches the word *nana* when x is replaced by the word *na*. A word encounters a pattern if the pattern matches some infix of the word. For example, the word $banana$ encounters the pattern xx (as the word *nana* is one of its infixes).

Unavoidable patterns have been characterized using the Zimin patterns $(Z_n)_{n\geqslant 0}$ defined by:

$$Z_1 = x_1 \quad \text{and} \quad Z_{n+1} = Z_n x_{n+1} Z_n \quad \text{for all } n \geqslant 1.$$

A pattern over n distinct pattern variables is unavoidable if, and only if, the pattern itself is encountered in the n-th Zimin pattern Z_n. Zimin patterns can therefore be viewed as the canonical patterns for unavoidability.

A natural question is how long can a word over a k-letters alphabet be while not encountering the n-th Zimin pattern. The associated length is denoted $f(n,k)$ for $n \geqslant 1$ and $k \geqslant 2$ and has been the subject of several recent articles. However before [53], the best lower-bound for $f(n,k)$ was doubly exponential. In [53], we show that for $n \geqslant 4$:

$$f(n,2) \;>\; 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \Bigg\} n-3 \text{ times}$$

We use Stockmeyer's yardstick construction [136] to construct for each $n \geqslant 1$, a family of words of length at least $\text{Tower}(n-1,2)$ that does not encounter Z_n (if $n \geqslant 3$). As these words are over an alphabet of size $2n-1$, this immediately establishes that $f(n,2n-1) > \text{Tower}(n-1,2)$. By using a carefully chosen encoding in binary we were able to prove the announced lower bound for $f(n,2)$. Independently, a similar (but slightly worst) bound was obtained using the probabilistic method in [67].

2
Preliminaries

This chapter collects some definitions which are used throughout the document and can safely be skipped at first. The notes in the margin highlight non-standard definitions.

2.1 Orders

We only recall basic definitions on partial and total orders. For a more detailed presentation, we refer the reader to [76, 126].

2.1.1 Partial orders

A *partial order* $\leqslant$ over a set D is a binary relation in $D \times D$ which is reflexive, antisymmetric and transitive. We write $<$ for the strict version of $\leqslant$ (i.e., $x < y$ if and only if $x \leqslant y$ and $x \neq y$). Two elements x and y of D are *incomparable* if neither $x \leqslant y$ nor $y \leqslant x$ hold. An *ω-chain* (for $\leqslant$) is an infinite sequence $(d_i)_{i \geqslant 0}$ such that $d_0 \leqslant d_1 \leqslant d_2 \leqslant \dots$ A *directed subset* of D is a subset X of D such that every finite subset of X has an upper bound. For example, ω-chains are directed subsets.

The *product of two partial orders* C and D is denoted $C \times D$. The order $<$ on pairs of elements of $C \times D$ is defined by $(c_1, d_1) \leqslant (c_2, d_2)$ if and only if $c_1 \leqslant_C c_2$ and $d_1 \leqslant_D d_2$.

A function $f : C \mapsto D$ between two partial orders is *monotonous* (or *order-preserving*) if $x \leqslant_C y$ implies that $f(x) \leqslant_D f(y)$.

A *complete partial order* (or *cpo* for short) is a pair $(D, \leqslant)$ where $\leqslant$ is a partial order on D with a least element, usually denoted $\bot$, and such that every ω-chain has a least upper bound (or supremum).

What we call cpo is sometimes called directed ω-cpo.

The product of two cpos is a cpo.

A function $f : C \mapsto D$ between two cpos is *continuous* if it is monotonous and its preserves the least upper-bounds of ω-chains.

A *directed complete partial order* (or *dcpo* for short) is a pair $(D, \leqslant)$ where $\leqslant$ is a partial order on D and such that every finite subset has a least upper bound (or supremum).

2.1.2 Linear orders

A *linear order* (or *total order*) $\leqslant$ over a set D is a partial order on D which is total (i.e., for all $x, y \in D$, either $x \leqslant y$ or $y \leqslant x$).

A *sub-order* of a linear order $(D, \leqslant)$ is a subset of D with the order induced by $\leqslant$ on D.

Given two linear orders $(A, \leqslant_A)$ and $(B, \leqslant_B)$ with $A \cap B = \varnothing$, we define their *sum* $A + B$ as the linear order $(A \uplus B, \leqslant)$ where for all $x, y \in A \uplus B$,

$$x \leqslant y \quad \Leftrightarrow \quad \begin{array}{ll} & x, y \in A \text{ and } x \leqslant_A y \\ \text{or} & x, y \in B \text{ and } x \leqslant_B y \\ \text{or} & x \in A \text{ and } y \in B \end{array}$$

Intuitively we make all the elements of A smaller than the elements of B. The sum generalises to a family $(L_i)_{i \in I}$ of linear orders indexed by elements of a linear order $(I, <_I)$ and is denoted $\sum_{i \in I} L_i$.

Given two linear orders $(A, \leqslant_A)$ and $(B, \leqslant_B)$, we define their *product* $A \cdot B$ as the linear order $(A \times B, \leqslant)$ where $\leqslant$ is the **reversed** lexicographic order $(x_1, y_1) \leqslant (x_2, y_2)$ if $y_1 <_B y_2$ or $y_1 = y_2$ and $x_1 \leqslant_A x_2$. Intuitively, the product is obtained by replacing each element of B by a copy of A.

2.1.3 Ordinals

A linear order $(D, \leqslant)$ is *well-founded* if every subset of D has a least element.

Ordinals are the isomorphism classes of well-founded linear orders. For all $n \geqslant 0$, the ordinal corresponding to the finite linear order with n elements is denoted by $\mathbf{n}$. The isomorphism class of $(\mathbb{N}, \leqslant)$ is denoted by ω. The isomorphism class of $(\mathbb{Z}, \leqslant)$ is denoted by ζ.

The class of ordinals is closed under sum and product. Note that sum (and the product) are not commutative. For instance $1 + \omega = \omega$ but $\omega + 1 \neq \omega$.

An initial segment of well-founded order $(D, \leqslant)$ is a suborder induced by a set of the form $\{ y \mid y \leqslant x \}$ for some $x \in D$.

Ordinals can be totally ordered by taking $\alpha \leqslant \beta$ if α is isomorphic to an initial segment of β.

An ordinal of the form $\alpha + 1$ is called a *successor ordinal*. Non-successor ordinals are called limit ordinals. A limit ordinal β satisfy that $\beta = (\{\alpha \mid \alpha < \beta\}, <)$.

Ordinal induction can be used to prove that a property $\mathcal{P}$ holds for all ordinals. It is enough to show that it holds for $\mathbf{0}$, if it holds for α, it also holds for $\alpha + 1$ and if it holds for all $\alpha < \beta$ for some limit ordinal β then it holds for β.

For every ordinal α, we define ω^α by taking:

$$\begin{aligned} \omega^{\alpha+1} &= \omega^\alpha \cdot \omega \\ \omega^\beta &= \sup(\{\omega^\alpha \mid \alpha < \beta\}) \qquad \text{for } \beta \text{ a limit ordinal.} \end{aligned}$$

The Cantor normal form can be defined for all ordinals. However it becomes less informative above ε_0 as we lose $\alpha > \beta_1$ which is replace by $\alpha \geqslant \beta_1$.

Let ε_0 be the smallest ordinal such that $\varepsilon_0 = \omega^{\varepsilon_0}$. Every ordinal $\alpha < \varepsilon_0$ can uniquely be written as:

$$\alpha = \omega^{\beta_1} \cdot c_1 + \cdots + \omega^{\beta_k} \cdot c_k$$

with $c_1, \cdots, c_k < \omega$ and $\alpha > \beta_1 > \cdots > \beta_k$. This presentation is called the *Cantor normal form* of the ordinal.

2.1.4 Scattered linear orders

A *dense linear order* $\leqslant$ over a set D is such that for all $x < y$, there exists z such that $x < z < y$.

A *scattered order* $\leqslant$ over a set D is a linear order which does not contain any dense sub-order.

For countable scattered orders, a more constructive characterisation is provided by Hausdorff's theorem which also gives a measure of the *complexity* of such orders.

Theorem 2.1 (Hausdorff [Hausdorff08])
A countable linear order is scattered if and only if it belongs to

$$\mathcal{S} = \bigcup_{\alpha} V_\alpha$$

where

$$V_0 = \{\mathbf{0}, \mathbf{1}\} \text{ and } V_\beta = \left\{ \sum_{i \in \zeta} L_i \mid \forall i, L_i \in \bigcup_{\alpha < \beta} V_\alpha \right\}.$$

The Hausdorff rank of a scattered order L, written $r_H(L)$, is the smallest α such that L can be expressed as a finite sum of elements of V_α. For instance, we have $r_H(\zeta) = r_H(\omega) = 1$ and $r_H(\omega^2) = 2$.

The Hausdorff rank of the ordinal ω^α is equal to α. In particular if α is written $\sum_{i=1}^{k} \omega^{\alpha_i}$ with $\alpha_1 \geqslant \ldots \geqslant \alpha_k$ in Cantor's normal form then $r_H(\alpha) = \alpha_1$.

The standard definition considers the smallest α such that L belongs to V_α. It is easy to see that this two ranks can only differ by at most one.

2.2 Terms and trees

2.2.1 Trees

Let A be a finite alphabet. We let A^* denote the set of finite words over A, and we refer to a subset of A^* as a language over A. A tree t with directions in A (or simply a *tree* if A is clear from the context) is a non-empty prefix-closed subset of A^*. Elements of t are called *nodes* and the empty word ε is called the *root* of t. For a node $u \in t$, the subtree of t rooted at u, denoted t_u, is the tree $\{v \in A^* \mid u \cdot v \in t\}$. We let $\mathrm{Trees}^\infty(A)$ denote the set of trees with directions in A.

2.2.2 Terms

A *ranked alphabet* is a set of symbols together with an arity $\rho(a) \geqslant 0$ for each symbol a. Symbols of arity 0 are called *constants*. The set of finite terms Terms_Σ build over a ranked alphabet Σ is the smallest set satisfying that:

- $a \in \mathrm{Terms}_\Sigma$ for all constant $a \in \Sigma$,
- if $t_1, \ldots, t_k \in \mathrm{Terms}_\Sigma$ then $f(t_1, \ldots, t_k) \in \mathrm{Terms}_\Sigma$ for all $f \in \Sigma$ with arity $k > 0$.

The *infinite terms* built over a ranked alphabet A are those trees with directions

$$\overrightarrow{A} \stackrel{\text{def}}{=} \bigcup_{f \in A} \overrightarrow{f} \text{ where } \overrightarrow{f} = \{f_1, \ldots, f_{\rho(f)}\} \text{ if } \rho(f) > 0 \text{ and } \overrightarrow{f} = \{f\} \text{ if } \rho(f) = 0.$$

For a tree $t \in \mathrm{Trees}^\infty(\overrightarrow{A})$ to be a term, we require, for all nodes u, that the set $A_u = \{d \in \overrightarrow{A} \mid ud \in t\}$ is empty if and only if u ends with some $f \in A$ (hence $\rho(f) = 0$)

The infinite terms could be defined co-inductively in a similar way as the finite terms. For our purpose, it is more convenient to define them as a sub-class of infinite trees.

and if A_u is non-empty, then it is equal to some $\overrightarrow{f}$ for some $f \in A$. We let $\mathrm{Terms}(A)$ denote the set of terms over A.

For $c \in A$ of arity 0, we let c denote the term $\{\varepsilon, c\}$. For $f \in A$ of arity $n > 0$ and for terms $t_1, \ldots, t_n$, we let $f(t_1, \ldots, t_n)$ denote the term $\{\varepsilon\} \cup \bigcup_{i \in [1,n]} \{f_i\} \cdot t_i$. These notions are illustrated in Figure 2.1.

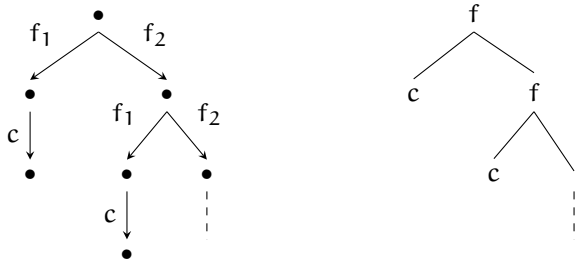


Figure 2.1. Two representations of the infinite term $f_2^*\{f_1 c, f_1, \varepsilon\} = f(c, f(c, f(\cdots)))$ over the ranked alphabet $\{f, c\}$, assuming that $\rho(f) = 2$ and $\rho(c) = 0$.

2.3 Labeled transition systems

A rooted labelled transition system is an edge-labelled directed graph with a distinguished vertex, called the root. Formally, a *rooted labelled transition system* $\mathcal{L}$ (LTS for short) is a tuple $(D, r, \Sigma, (\xrightarrow{a})_{a \in \Sigma})$, where D is a finite or countable set called the *domain*, $r \in D$ is a distinguished element called the *root*, Σ is a finite set of *labels*, and for all $a \in \Sigma$, $\xrightarrow{a} \subseteq D \times D$ is a binary relation on D.

For any $a \in \Sigma$ and any pair $(s, t) \in D^2$ we write $s \xrightarrow{a} t$ to indicate that $(s, t) \in \xrightarrow{a}$, and we refer to it as an a-*transition* with *source* s and *target* t. For a word $w = a_1 \cdots a_n \in \Sigma^*$, we define a binary relation $\xrightarrow{w}$ on D by letting $s \xrightarrow{w} t$ (meaning that $(s, t) \in \xrightarrow{w}$) if there exists a sequence $s_0, \ldots, s_n$ of elements in D such that $s_0 = s$, $s_n = t$, and for all $i \in [1, n]$, $s_{i-1} \xrightarrow{a_i} s_i$. These definitions are extended to languages over Σ by taking, for all $L \subseteq \Sigma^*$, the relation $\xrightarrow{L}$ to be the union of all $\xrightarrow{w}$ for $w \in L$.

When considering LTS associated with computational models, it is usual to allow silent (or internal) transitions. The symbol for silent transitions is usually ε but here, to avoid confusion with the empty word, we will use λ instead. Following [135, p. 31], we forbid a vertex to be the source of both a silent transition and a non-silent transition. Formally, an *LTS with silent transitions* is an LTS $(D, r, \Sigma, (\xrightarrow{a})_{a \in \Sigma})$ whose set of labels contains a distinguished symbol, denoted $\lambda \in \Sigma$ and such that for all $s \in D$, if s is the source of a λ-transition, then s is not the source of any a-transition with $a \neq \lambda$. We let Σ_λ denote the set $\Sigma \setminus \{\lambda\}$ of non-silent transition labels. For all words $w = a_1 \cdots a_n \in \Sigma_\lambda^*$, we let $\stackrel{w}{\Longrightarrow}$ denote the relation $\xrightarrow{L_w}$, where $L_w \stackrel{\text{def}}{=} \lambda^* a_1 \lambda^* \cdots \lambda^* a_n \lambda^*$ is the set of words over Σ obtained by inserting arbitrarily many occurrences of λ in w.

An LTS (with silent transitions) is said to be *deterministic* if for all s, t_1 and t_2 in D and all a in Σ, if $s \xrightarrow{a} t_1$ and $s \xrightarrow{a} t_2$, then $t_1 = t_2$.

2.4 Logics

2.4.1 Monadic second order logic

Monadic second order logic (MSO logic for short) is the extension of first-order logic with the ability to quantify over sets of elements. Here we define MSO logic over LTS with a label set Σ. In general it is defined for any relational structure and hence for any structures such as trees, partial orders, ... that can be encoded in relational structures.

Let $\mathcal{V}_1$ be an infinite set of first-order variables and $\mathcal{V}_2$ be an infinite set of second-order variables. We use lowercase letters such as x, y and z for first-order variables and uppercase letters such as X, Y and Z for second-order variables. *MSO formulas* are build from the atomic formula $x \xrightarrow{a} y$ with $a \in \Sigma$, $x = y$ and $x \in X$ for all $x, y \in \mathcal{V}_1$ and $X \in \mathcal{V}_2$ using the boolean connectors $\wedge$ and $\neg$ and the existential quantifiers $\exists x$ and $\exists X$ for all $x \in \mathcal{V}_1$ and $X \in \mathcal{V}_2$. We allows ourself to use syntactic sugar like $\vee$, $\forall X, \rightarrow, \ldots$

We write $\varphi(X_1, \ldots, X_n, y_1, \ldots, y_m)$ to denote that the free variables of the formula φ are among $X_1, \ldots, X_n$ and $y_1, \ldots, y_m$ respectively. A formula without free variables is called a *sentence*. For an LTS $\mathcal{L}$ and a sentence φ, we write $\mathcal{L} \models \varphi$ if $\mathcal{L}$ satisfies the formula φ. The *MSO-theory of $\mathcal{L}$* is the set of sentences satisfied by $\mathcal{L}$. We say that an LTS $\mathcal{L}$ has a *decidable MSO-theory* if its MSO-theory is recursive.

For every formula $\varphi(X_1, \ldots, X_n, y_1, \ldots, y_m)$, all subsets $U_1, \ldots, U_n$ of vertices of $\mathcal{L}$ and all vertices $v_1, \ldots, v_m$ of $\mathcal{L}$, we write $\mathcal{L} \models \varphi[U_1, \ldots, U_n, v_1, \ldots, v_m]$ to express that φ holds in $\mathcal{L}$ when X_i is interpreted as U_i for all $i \in [1, n]$ and y_j is interpreted as v_j for all $j \in [1, m]$.

A set U of vertices of $\mathcal{L}$ is *MSO-definable* in $\mathcal{L}$ if there exists a formula $\varphi(x)$ such that:

$$U = \{u \mid \mathcal{L} \models \varphi[u]\}$$

The definition of an MSO-definable vertex is similar.

2.4.2 Modal μ-calculus

The *modal μ-calculus* over LTS labelled by Σ is defined using greatest and least fix-point operators, an existential modality $\Diamond_a$ and a universal modality $\Box_a$ for each $a \in \Sigma$. Every μ-calculus formula φ defines a set of vertices of the LTS denoted by $[\![\varphi]\!]$.

In this manuscript, we will never use the precise definition of the μ-calculus and we will rely only on the properties given below. We refer the reader to [19] for a detailed presentation and for further references.

Every μ-calculus formula ψ can easily be translated into an MSO formula $\varphi(X)$ defining $[\![\psi]\!]$. In general, it is not true that sets definable in MSO logic are definable in μ-calculus. However for deterministic tree, we have equi-expressivity in the following sense.

Proposition 2.2 ([114])
The modal μ-calculus and MSO logic are equi-expressive over deterministic trees. More precisely for every MSO formula φ, there (effectively) exists a μ-calculus formula ψ, such that for all deterministic tree t, $t \models \varphi$ if and only if the root of t belongs to $[\![\psi]\!]$.

Over non-deterministic trees, the μ-calculus captures the bisimulation-invariant properties expressible in MSO logic [97]. The following proposition expresses the equivalence with alternating parity tree automata (see Section ??).

Proposition 2.3 ([77])
For every μ-calculus formula φ, there exists an alternating tree automaton $\mathcal{A}_\varphi$ such that for all deterministic tree t and node u, $\mathcal{A}_\varphi$ accepts t_u, the subtree rooted at u, if and only if $u \in [\![\varphi]\!]$. Furthermore $\mathcal{A}_\varphi$ can be constructed in linear time.

2.5 Transformations of labelled transitions systems

In this section, we recall the definitions of the transformation of LTS used throughout this document.

An *MSO-interpretation* I from LTS labelled by Σ to LTS labelled by Γ is given by a tuple of formulas $(\delta(x), \varphi_a(x,y))_{a\in\Gamma}$ where these formulas use the alphabet Σ. Applying the MSO-interpretation I to an LTS $\mathcal{L}$ labelled by Σ produces an LTS labelled by Γ denoted by $\mathcal{I}(\mathcal{L})$. The set of vertices of $I(\mathcal{L})$ is the set of vertices of $\mathcal{L}$ satisfying $\delta(x)$. For $a \in \Gamma$, there is a transition $u \xrightarrow{a} v$ in $I(\mathcal{L})$ if and only if $\mathcal{L} \models \varphi_a[u,v]$.

Proposition 2.4
For all MSO-interpretation I and all $\mathcal{L}$, if the LTS $\mathcal{L}$ has a decidable MSO-theory then the LTS $I(\mathcal{L})$ also has a decidable MSO-theory.

The unfolding of an LTS $\mathcal{L}$ from one of its vertices u is the tree, denoted $\mathrm{Unf}(\mathcal{L}, u)$ formed by the path from u in $\mathcal{L}$.

Theorem 2.5 ([73])
For an LTS $\mathcal{L}$ and an MSO-definable vertex u of $\mathcal{L}$, if $\mathcal{L}$ has a decidable MSO-theory then $\mathrm{Unf}(\mathcal{L}, u)$ also has a decidable MSO-theory.

2.6 Games

2.6.1 Two-players infinite duration games

A *graph* is a pair $G = (V, E)$ where V is a (possibly infinite) set of *vertices* and $E \subseteq V \times V$ is a set of *edges*. For a vertex v we let $E(v) = \{v' \mid (v,v') \in E\}$ and in the rest of the document (hence, this is implicit from now on), we only consider graphs that have no dead-end, i.e., such that $E(v) \neq \varnothing$ for all v.

An *arena* is a triple $\mathcal{G} = (G, V_{\mathbf{E}}, V_{\mathbf{A}})$ where $G = (V, E)$ is a graph and $V = V_{\mathbf{E}} \uplus V_{\mathbf{A}}$ is a partition of the vertices among two players, Éloise and Abelard.

Éloise and Abelard play in $\mathcal{G}$ by moving a pebble along edges. A *play* from an initial vertex v_0 proceeds as follows: the player owning v_0 (i.e., Éloise if $v_0 \in V_{\mathbf{E}}$, Abelard otherwise) moves the pebble to a vertex $v_1 \in E(v_0)$. Then the player owning v_1 chooses a successor $v_2 \in E(v_1)$ and so on. As we assumed that there is no dead-end, a play is an infinite word $v_0v_1v_2\cdots \in V^\omega$ such that for all $0 \leqslant i$ one has $v_{i+1} \in E(v_i)$. A *partial play* is a prefix of a play, i.e., it is a finite word $v_0v_1\cdots v_\ell \in V^*$ such that for all $0 \leqslant i < \ell$ one has $v_{i+1} \in E(v_i)$.

A *strategy* for Éloise is a function $\varphi : V^*V_{\mathbf{E}} \to V$ assigning, to every partial play

ending in some vertex $v \in V_{\mathbf{E}}$, a vertex $v' \in E(v)$. Strategies of Abelard are defined likewise, and usually denoted ψ. In a given play $\lambda = v_0 v_1 \cdots$ we say that Éloise (resp. Abelard) *respects a strategy* φ (resp. ψ) if whenever $v_i \in V_{\mathbf{E}}$ (resp. $v_i \in V_{\mathbf{A}}$) one has $v_{i+1} = \varphi(v_0 \cdots v_i)$ (resp. $v_{i+1} = \psi(v_0 \cdots v_i)$).

A *winning condition* is a subset $\Omega \subseteq V^\omega$ and a (two-player perfect information) *game* is a pair $\mathbb{G} = (\mathcal{G}, \Omega)$ consisting of an arena and a winning condition.

A play λ is *won* by Éloise iff $\lambda \in \Omega$; otherwise λ is won by Abelard. A strategy φ is *winning* for Éloise in $\mathbb{G}$ from a vertex v_0 if any play starting from v_0 where Éloise respects φ is won by her. Finally a vertex v_0 is *winning* for Éloise in $\mathbb{G}$ if she has a winning strategy φ from v_0. Winning strategies and winning vertices for Abelard are defined likewise.

We now define some classical winning conditions.

- A *reachability* winning condition is of the form $V^* F V^\omega$ for a set $F \subseteq V$ of target vertices, i.e., winning plays are those that eventually visit a vertex in F.
- A *Büchi* winning condition is of the form $(V^* F)^\omega$ for a set $F \subseteq V$ of final vertices, i.e., winning plays are those that infinitely often visit vertices in F.
- A *co-Büchi* condition is of the form $V^* (V \setminus F)^\omega$ for a set $F \subseteq V$ of forbidden vertices,
- A *parity* winning condition is defined by a *colouring* function Col that is a mapping $\mathrm{Col} : V \to C \subset \mathbb{N}$ where C is a *finite* set of *colours*. The parity winning condition associated with Col is the set $\Omega_{\mathrm{Col}} = \{v_0 v_1 \cdots \in V^\omega \mid \liminf(\mathrm{Col}(v_i))_{i \geqslant 0} \text{ is even}\}$, i.e., a play is winning if and only if the smallest colour infinitely often visited is even.

Finally a reachability (resp. Büchi, co-Büchi, parity) game is one equipped with a reachability (resp. Büchi, co-Büchi, parity) winning condition. For notation of such games we often replace the winning condition by the object that is used to define it (i.e., F or Col).

2.6.2 Markov Decision Process

Perfect Information Setting

A *probability distribution* over a countable set X is a mapping $d : X \to [0, 1]$ such that $\sum_{x \in X} d(x) = 1$. In the sequel we denote by $\mathcal{D}(X)$ the set of probability distributions over X. In this document, all probabilities will be rational numbers, which will be described in binary when dealing with encoding.

An *arena* is a tuple $\mathcal{G} = \langle S, s_{\mathrm{ini}}, \Sigma, \zeta \rangle$ where S is a countable set of states, s_{ini} is an initial state, Σ is a finite set of *actions* and $\zeta : S \times \Sigma \to \mathcal{D}(S)$ is the transition (total) function.

A play in such an arena proceeds as follows. It starts in state s_{ini} and Éloise picks an action σ, and a successor state is chosen according to the probability distribution $\zeta(s_{\mathrm{ini}}, \sigma)$. Then Éloise chooses a new action and the state is updated and so on forever. Hence, a *play* is an infinite sequence $s_0 s_1 s_2 \cdots \in S^\omega$ such that $s_0 = s_{\mathrm{ini}}$ and for every $i \geqslant 0$, there exists a $\sigma \in \Sigma$ with $\zeta(s_i, \sigma)(s_{i+1}) > 0$. In the sequel we refer to a prefix of a play as a *partial play* and we denote by Plays the set of all plays.

A *(pure) strategy* for Éloise is a function $\varphi : S^* \to \Sigma$ assigning to every partial play an action. Of special interest are those strategies that do not require memory: a

We do not consider here randomised strategies as in the setting of this paper they are useless. Note that for finite MDP, optimal strategies — when exists — can always be chosen to be pure.

strategy φ is *memoryless* if $\varphi(\lambda \cdot s) = \varphi(\lambda' \cdot s)$ for all partial play λ, λ' and all states s (*i.e.* φ only depends on the current state). A play $\lambda = s_0 s_1 s_2 \cdots$ is *consistent* with a strategy φ if $\zeta(s_i, \varphi(v_0 \cdots v_i))(s_{i+1}) > 0$, for all $i \geqslant 0$.

Now, for any partial play λ, the *cylinder* for λ is the set $\mathrm{Cyl}(\lambda) = \lambda S^\omega \cap \mathrm{Plays}$. Let $\mathcal{F}_P$ be the σ-algebra generated by the set of cylinders. Then, $(\mathrm{Plays}, \mathcal{F}_P)$ is a measurable space.

A strategy φ induces a probability space over $(\mathrm{Plays}, \mathcal{F}_P)$ as follows: one defines a measure μ_φ on cylinders and then uniquely extends it to a probability measure on $\mathcal{F}_P$ using the Carathéodory's unique extension theorem. For this, we first define inductively μ_φ on cylinders:

- as all plays start from s_{ini}, we let $\mu_\varphi(\mathrm{Cyl}(s_{\mathrm{ini}})) = 1$;
- for any partial play λ ending in some state s, we let $\mu_\varphi(\mathrm{Cyl}(\lambda \cdot s')) = \mu_\varphi(\mathrm{Cyl}(\lambda)) \cdot \zeta(s, \varphi(\lambda))(s')$.

We also denote by μ_φ the unique extension of μ_φ to a probability measure on $\mathcal{F}$. Then $(\mathrm{Plays}, \mathcal{F}_P, \mu_\varphi)$ is a probability space.

An *objective* is a measurable set $\mathcal{O} \subseteq \mathrm{Plays}$: a play is winning if it belongs to $\mathcal{O}$. A *Markov decision process (MDP, aka* one-and-half-player game*)* is a pair $\mathbb{G} = (\mathcal{G}, \mathcal{O})$ where $\mathcal{G}$ is an arena and $\mathcal{O}$ is an objective. In the sequel we should focus on ω-regular objectives (which are easily seen to be measurable), whose definitions are the same as for two-players games.

A strategy φ is *almost-surely winning* (*resp. positively winning*) if $\mu_\varphi(\mathcal{O}) = 1$ (*resp.* $\mu_\varphi(\mathcal{O}) > 0$). If such a strategy exists, we say that Éloise *almost-surely wins* (*resp. positively wins*) $\mathbb{G}$. The *value* of $\mathbb{G}$ is defined as $\mathrm{Val}(\mathbb{G}) = \sup_\varphi \mu_\varphi(\mathcal{O})$, and a strategy φ is *optimal* if $\mathrm{Val}(\mathbb{G}) = \mu_\varphi(\mathcal{O})$.

When the set of actions Σ is reduced to one element, the MDP $(\mathcal{G}, \mathcal{O})$ is called a *Markov chain* and we omit the unique action from all the definitions. The set Plays is called the set of *traces* of the Markov chain and is denoted Traces. We write $\mu_\mathcal{G}$ the probability measure associated with the unique strategy. We say that the Markov chain *almost-surely* fulfils its objective if $\mu_\mathcal{G}(\mathcal{O}) = 1$.

Theorem 2.6 ([74],[65])
Let $\mathbb{G}$ be an MDP over a finite arena with a parity objective. Then, one can decide in polynomial time whether Éloise almost-surely (*resp.* positively) wins. Moreover, Éloise always has an optimal memoryless strategy.

Imperfect Information Setting

Now we consider the case where Éloise has imperfect information about the current state. For this, we consider an equivalence relation $\sim$ over S. We let $[s]_\sim$ be the equivalence class of s for $\sim$ and $S/_\sim$ be the set of equivalence classes of $\sim$ over S.

The intuitive meaning of $\sim$ is that two states $s_1 \sim s_2$ cannot be distinguished by Éloise. We easily extend $\sim$ to partial plays: $s_0 s_1 \cdots s_n \sim s'_0 s'_1 \cdots s'_n$ if and only if $s_i \sim s'_i$ for all $i = 0, \cdots, n$. As two equivalent plays $\lambda_1 \sim \lambda_2$ cannot be distinguished by Éloise she should therefore behave the same in both of them.

Hence, we should only consider so-called observation-based strategies. An *observation-based (pure) strategy* is a function $\varphi : (S/_\sim)^* \to \Sigma$, *i.e.*, to choose her next action, Éloise considers the sequence of observations she got so far[1]. In particular,

[1]By abuse of notation, we shall write $\varphi(s_0 \cdots s_n)$ to mean $\varphi([s_0]_\sim \cdots [s_n]_\sim)$

an observation-based strategy φ is such that $\varphi(\lambda) = \varphi(\lambda')$ whenever $\lambda \sim \lambda'$. In this context, a *memoryless strategy* is a function from $S/_\sim \to \mathcal{D}(\Sigma)$, *i.e.* it only depend on the current equivalence class.

A *partial observation Markov decision process (POMDP, aka* one-and-half-player imperfect information game*)* is a triple $(\mathcal{G}, \sim, \mathcal{O})$ where $\mathcal{G}$ is an arena, $\sim$ is an equivalence relation over states and $\mathcal{O}$ is an objective. We say that Éloise *almost-surely wins* (*resp.* positively wins) $\mathbb{G}$ if she has an almost-surely (*resp.* positively) winning *observation-based* strategy. Finally, the *value* of $\mathbb{G}$ is defined as $\mathrm{Val}(\mathbb{G}) = \sup_\varphi \mu_\varphi(\mathcal{O})$ where φ ranges over observation-based strategies; optimality is defined as previously.

The following decidability results are known for POMDP:

Theorem 2.7 ([8])
In a POMDP with a Büchi (*resp.* co-Büchi) objective, deciding whether Éloise almost-surely (*resp.* positively) wins is EXPTIME-complete. Moreover if Éloise has an almost-surely (*resp.* positively) winning strategy, she has an almost-surely (*resp.* positively) winning strategy with finite memory.

In a POMDP with a co-Büchi (*resp.* Büchi) objective, it is undecidable whether Éloise almost-surely (*resp.* positively) wins.

The results in Theorem 2.6 and 2.7 do not depend on the encoding of probability distributions, as the only relevant information is which probabilities are non zero.

2.7 Automata on infinite words and trees

An infinite word over the alphabet A is an infinite sequence of letter of A. We denote by A^ω the set of all infinite words over A.

A *non-deterministic Büchi ω-word automata* is given by a tuple $(Q, q_{\mathrm{in}}, \Delta, F)$ where Q is the set of states, $q_{\mathrm{in}} \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\Delta \subseteq Q \times A \times Q$ is the set of transitions. A run of $\mathcal{A}$ on an ω-word $w = a_1 a_2 \cdots$ is an infinite sequence of states $q_0 q_1 \cdots$ such that $q_0 = q_{\mathrm{in}}$ and for all $i \geqslant 0$, (q_i, a, q_{i+1}) is a transitions in Δ. A run is accepting if it contains infinitely many final states. An ω-word w is accepted by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ over w. We denote by $L(\mathcal{A})$ the set of ω-words accepted by $\mathcal{A}$.

Parity ω-word automata are defined similarly but with the parity condition instead of the Büchi condition. Deterministic parity ω-word automata accept the same languages as Büchi ω-word automata.

For simplicity, we only define on tree automata on full binary trees whose nodes are labelled by a finite alphabet Σ (i.e. mappings from $\{0,1\}^* \mapsto \Sigma$). Of course, the definition and results are easily adapted to arbitrary deterministic non-necessarily complete trees.

A *(non-deterministic) parity tree automaton* on Σ-labelled (full binary) trees is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \Delta, \Omega)$ with a finite set Q of states, initial state $q_0 \in Q$, transition relation $\Delta \subseteq Q \times \Sigma \times Q \times Q$, and a priority function $\Omega : Q \to \mathbb{N}$. A *run* of $\mathcal{A}$ on a Σ-labelled tree t from a state $q \in Q$ is a tree ρ labelled by Q such that $\rho(\varepsilon) = q$, and for each $u \in \{0,1\}^*$ we have $(\rho(u), t(u), \rho(u0), \rho(u1)) \in \Delta$. We say that ρ is accepting if on each path the minimal priority appearing infinitely often is even. A tree is accepted by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ for it. We denote by $L(\mathcal{A})$ the set of infinite trees accepted by $\mathcal{A}$.

Tree automata on infinite trees can be defined with other acceptance conditions

such Büchi and co-Büchi in a similar way.

So far, we defined non-deterministic parity tree automata. An *alternating parity tree automaton* can propagate more than one state to each child of the current node. For a precise definition, we refer to [98]. We will mainly use that fact that for the parity acceptance condition non-deterministic and alternating automata accept the same languages.

A famous result due to Rabin is that the languages accepted by parity tree automata are the languages definable in MSO logic.

Theorem 2.8 (Rabin's theorem [124])
For every MSO sentence φ, there exists a parity tree automaton $\mathcal{A}_\varphi$ such that for every tree t, we have:

$$t \vDash \varphi \text{ if and only if } \mathcal{A}_\varphi \text{ accepts } t$$

This chapter is based on [45] co-authored with Matthew Hague, Antoine Meyer, Luke Ong and Olivier Serre, [58] co-authored with Michaela Slaats, [21] co-authored with Laurent Braud, [26] co-authored with Christopher Broadbent, Luke Ong and Olivier Serre, [35] co-authored with Zoltán Ésik, [52] co-authored with Olivier Serre, [2] with Luca Aceto, Zoltán Ésik and Anna Ingólfsdóttir, [38] co-authored with Zoltán Ésik and [54] co-authored with Olivier Serre. The presentation follows the definitions and notations of [54].

3 Recursion schemes

3.1 Introduction

Often, we write recursion scheme or even scheme for brevity without any implication on its order.

This chapter presents my contributions to the study of higher-order recursion schemes. It is not intended as a detailed presentation of the state of the art but it should offer the necessary background to understand my contributions to the field. Before proceeding with the necessary formal definitions, I would like to attempt an informal presentation of recursion schemes that should be accessible to any fellow computer scientist that enjoys programming languages as I do.

Recursion schemes are an abstract model for programs written in a *pure* functional language such as Haskell, Ocaml, ML, Scala, ... By abstract model, we mean that some but not all features are captured. The main restriction of this model compared to full fledge functional programs is that it can only handle data (integers, lists, ...) in a very limited fashion.

In this informal introduction rather than defining recursion schemes as a mathematical model, we will define them as a syntactic subclass of Haskell programs. Here the choice of Haskell is not arbitrary. Indeed Haskell is a *lazy* functional language. It means that expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations. In consequence, arguments are not evaluated before they are passed to a function, but only when their values are actually used.[1] This allows Haskell programs to represent and work with *infinite* objects. For instance, the program below defines an infinite list `integers` containing all positive integers. Of course attempting to print the list does not terminate but thanks to lazy evaluation it is possible to retrieve its n-th element. In a non-lazy language, such an attempt would not terminate as the language would first attempt to fully evaluate `integers` where Haskell will only compute its n-th first elements.

```
integers = 0 : map (\x -> x+1) integers -- list of all integers
integers !! 5  -- 6-th element
```

[1] This definition of lazy evaluation is taken verbatim from the Haskell wiki https://wiki.haskell.org/Lazy_evaluation.

```
even = map (\x -> 2*x) integers -- list of even numbers
```

For the rest of this introduction, recursion schemes are Haskell programs that can only manipulate data-type representing terms and that do not perform any form of pattern matching on these terms. In particular, a function is never allowed to inspect or deconstruct its arguments.

The Haskell program below defines a data-type `Term` to represent the terms over the ranked alphabet $a : 0$, $b : 1$ and $c : 2$. The variable `finite` denotes the finite term $c(b(a), a)$. More interestingly, the variable `infinite` represents the infinite term $b(b(b(b(\cdots))))$.

```
data Term = A | B Term | C Term Term

finite :: Term
finite = C (B A) A
infinite :: Term
infinite = B infinite
```

The function `approx` is not a recursion scheme *as it uses a parameter of type `Int` and also performs* pattern-matching. *It is only presented here to convince the reader that infinite term can effectively be handled in Haskell.*

As we mentioned previously, it is possible to manipulate infinite terms in a meaningful way thanks to lazy-evaluation. In particular the function `approx` below computes the finite term obtained by *pruning* the infinite term at some given depth. For this, we need to add some new constant (`Void` in our case) that will be substituted at the cutting points. For instance, `approx infinite 2` returns the term `B (B Void)`.

```
data Term = Void | A | B Term | C Term Term

approx :: Term -> Int -> Term
approx _ 0 =  Void
approx A _ = A
approx (B t) n = B (approx t (n-1))
approx (C t1 t2) n = C (approx t1 (n-1)) (approx t2 (n-1))
```

Note that auxiliary functions are allowed to take functions as parameters as long as the only data-type they use is `Term`.

To sum up, a *recursion scheme* is an Haskell program of type `Term` which can, of course, use auxiliary recursive functions as long as they do not use other data type than `Term` or any form of pattern-matching. As a result, a scheme represents one, possibly infinite, term.

Before proceeding, let us give some examples of the expressivity of recursion schemes. Our first non-trivial example `comb_n`, given below, defines the infinite term depicted in Figure 3.1. This term, denoted Comb_n, is comb-shaped and the n-th teeth of the comb is the term $\underbrace{b(b(\cdots(b\ a)\cdots))}_{n \text{ times}}$ which we denote $\mathbf{n}$ in the following. More generally, for any function $f : \mathbb{N} \mapsto \mathbb{N}$, Comb_f is the comb-shaped term where the n-th teeth has length $f(n)$. This term is also depicted in Figure 3.1.

```
comb_n :: Term
comb_n = comb_n' (B A)
comb_n' :: Term -> Term
comb_n' x = C x (comb_n' (B x))
```

To better understand how comb_n produces this term, we can use a little syntactic sugar and remark that the argument x of comb_n' will always be some term of the form **n** for some $n \geqslant 1$. Hence the program can be rewritten as:

```
comb_n = comb_n' 1
comb_n' n = C n (comb_n' n+1)
```

Unfolding the recursive definition, we obtain:

$$\texttt{comb\_n} \rightsquigarrow \texttt{comb\_n'}\,\mathbf{1} \rightsquigarrow \mathbf{C}(\mathbf{1}, \texttt{comb\_n'}\,\mathbf{2}) \rightsquigarrow \mathbf{C}(\mathbf{1}, \mathbf{C}(\mathbf{2}, \texttt{comb\_n'}\,\mathbf{3})) \rightsquigarrow \cdots$$

which at the limit produces the term presented in Figure 3.1.

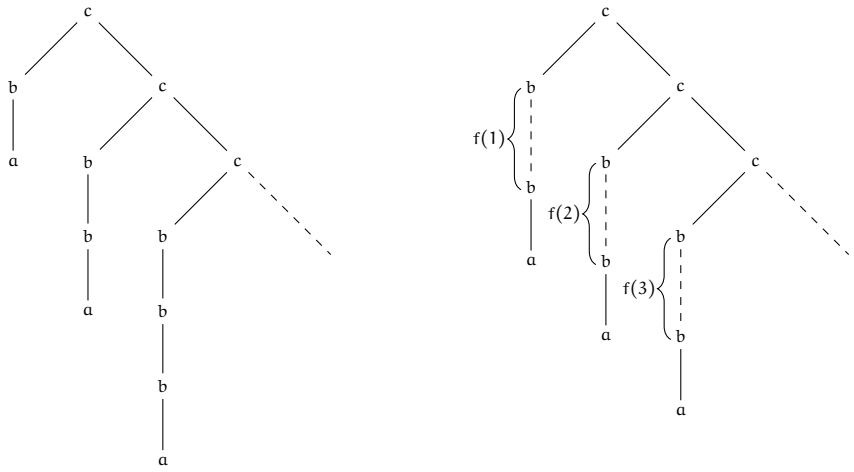


Figure 3.1. The term Comb_n produced by the scheme comb_n (left) and the general shape Comb_f for some function $f : \mathbb{N} \to \mathbb{N}$ (right).

To better understand the expressiveness of recursion schemes[2],we give examples of recursion schemes constructing the term Comb_f (see Figure 3.1) for more *complicated* functions $f : \mathbb{N} \mapsto \mathbb{N}$. Let us start with the Fibonacci sequence $\mathrm{fib} : \mathbb{N} \mapsto \mathbb{N}$ defined by $\mathrm{fib}(1) = \mathrm{fib}(2) = 1$ and $\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$ for all $n \geqslant 3$.

The scheme comb_fib given below compute the infinite term $\mathrm{Comb}_{\mathrm{fib}}$.

```
comb_fib :: Term
comb_fib = comb_fib' B B
comb_fib' :: (Term -> Term) -> (Term -> Term) -> Term
comb_fib' fst snd = C (fst A) (comb_fib' snd (add fst snd))
add :: (Term -> Term) -> (Term -> Term) -> (Term -> Term)
add n m x = n (m x)
```

Because the recursive definition of fib involves an addition, it is no longer possible to represent the integer $n \geqslant 0$ directly by the term **n**. Rather the integer $n \geqslant 0$ is represented by the function $\underline{\mathbf{n}}$ of type `Number = Term -> Term` defined by:

$$\underline{\mathbf{n}}(t) = \underbrace{b(b(\cdots(b\,t)\cdots))}_{n \text{ times}}.$$

As our definition of recursion scheme forbids the deconstruction of arguments using pattern-matching, it is not possible to construct a function taking the terms **n** *and* **m** *as arguments and producing the term* **n** + **m**.

[2] All the code presented in this document can be downloaded at www-igm.univ-mlv.fr/~carayol/HDR/schemes.hs.

By applying this function to the term a, we obtain the term corresponding to this integer (i.e., $\underline{\mathbf{n}}(a) = \mathbf{n}$ for all $n \geqslant 0$). With this representation, the addition operation simply becomes the composition of the functions (i.e., $\underline{\mathbf{n}} \circ \underline{\mathbf{m}} = \underline{\mathbf{n}+\mathbf{m}}$).

Now the code of `comb_fib` can be rewritten to become more readable.

```
type Number = Term -> Term
comb_fib :: Term
comb_fib = comb_fib' 1 1
comb_fib' :: Number -> Number -> Term
comb_fib' fib_{n-1} fib_n  = C fib_{n-1} (comb_fib' fib_n (add fib_{n-1} fib_n))
add :: Number -> Number -> Number
add n m  = n+m
```

It is now easy to see that the n-th call tp`comb_fib'` has $\underline{\mathbf{fib}(\mathbf{n})}$ and $\underline{\mathbf{fib}(\mathbf{n}+\mathbf{1})}$ as arguments.

This example illustrates two crucial features for the expressiveness of recursion schemes: the use of higher-order auxiliary functions, meaning functions that take functions as arguments and partial application of functions, meaning that a function is called with only part of its arguments as it is the case in `add fst snd`.

To further demonstrate the expressiveness of using auxiliary functions of higher order, we show how to construct $\mathrm{Comb}_{2^n}, \mathrm{Com}_{2^{2^n}}, \mathrm{Com}_{2^{2^{2^n}}}, \ldots$ which schemes of increasing order. In our context, the order of a function is 0 if it is of type `Term`, 1 if it has arguments which are of type `Term` and more generally the order of a function is one plus the maximal order of its arguments. For instance, the function `add` defined above is of order 2 as its arguments are of order 1, 1 and 0 respectively. The order of a scheme is the maximal order of a function it uses.

The scheme `comb_exp` of order 2 below defines the term Comb_{2^n}.

```
comb_exp :: Term
comb_exp = comb_exp' B
comb_exp' :: Number -> Term
comb_exp' n = C (n A) (comb_exp' (double n))
double :: Number -> Number
double n x = n (n x)
```

When rewritten using some syntactic sugar, we see that the n-th recursive call to `comb_exp'` is passed $\underline{\mathbf{2^n}}$ as argument.

```
comb_exp = comb_exp' (double 1)
comb_exp' n = C n (comb_exp' (double n))
double n = 2n
```

Let us move on to the construction of $\mathrm{Comb}_{2^{2^n}}$, we already know how to construct at the n-th recursive call the function $\underline{\mathbf{2^n}}$ that is to say, the function consisting in 2^n consecutive calls to the function `b`. Actually, we could have substituted the function `b` with any other function. In particular, we could have constructed the function D_n : `Number -> Number` consisting of 2^n consecutive call to `double`. When applying D_n to `B`, we obtain $\underline{\mathbf{2^{2^n}}}$. This is in essence the mechanism of the scheme `comb_exp_exp` given below, which describes $\mathrm{Comb}_{2^{2^n}}$.

```
comb_exp_exp :: Term
comb_exp_exp = comb_exp_exp' (double2 double)
comb_exp_exp' :: (Number -> Number) -> Term
comb_exp_exp' f = C (f B A) (comb_exp_exp' (double2 f))
double2 :: (Number -> Number) -> (Number -> Number)
double2 g x = g (g x)
```

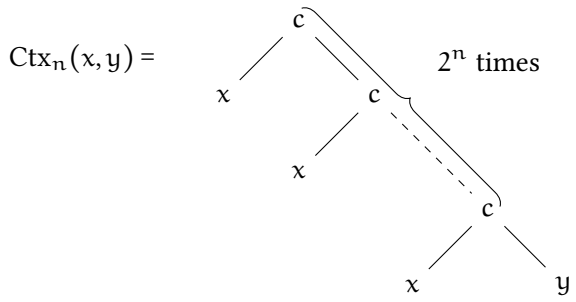
The function `double` and `double2` only differ by their type. It would be possible to replace both of them by the polymorphic version `double f = f . f` using the operator for composition of functions. However polymorphism is not allowed by the definition of recursion schemes.

It should be clear that this construction generalises to the tower of exponential of height k and results in a scheme of order $k+1$.

The examples given so far might give the false impression that the definition of the scheme always closely mimics natural recursive definition of the functions modulo some encoding of data as functions. The scheme `comb_log` given below, which defines the term $\mathrm{Comb}_{\lfloor \log_2(n) \rfloor}$ shows that it is not necessarily the case.

```
comb_log :: Term
comb_log = comb_log' (\x -> x) C
comb_log' :: (Term->Term->Term) -> Term -> Term
comb_log' ctx n = ctx n (comb_log' (B n) (dup context))
dup :: (Term->Term->Term) -> Term -> Term -> Term
dup ctx x y = ctx x (ctx x y)
```

Here the n-th recursive call to `comb_log'` is given the arguments Ctx_n and $\mathbf{n}$ where for $n \geqslant 1$, Ctx_n is the function type `Term ->Term ->Term` given by:



The correctness of the scheme `comb_log` uses the following properties:

- $\mathrm{Comb}_{\lfloor \log_2(n) \rfloor} = \mathrm{Ctx}_1(\mathbf{1}, \mathrm{Ctx}_2(\mathbf{2}, \mathrm{Ctx}_3(\mathbf{3}, \ldots)\ldots))$,
- $\mathrm{Ctx}_{n+1}(x, y) = \mathrm{Ctx}_n(x, \mathrm{Ctx}_n(x, y))$.

The main motivation to study recursion schemes is the automated verification of functional programs. In this setting, recursion schemes are used as an abstract model, an idea which dates back to the early 70s and the work of Nivat [113].

Even though, functional programming is not the most common paradigm encountered in the industry, it is worth noting that widely used languages such as Java, C#, python have functional programming features.

Nivat proposes to see a first-order recursive program as a recursion scheme simply by leaving all the base functions (i.e., the functions handling data), uninterpreted. This translation is purely syntactical and can therefore be automated. For example, consider the program `factorial_three`, given below, which computes the factorial of 3.

```
factorial_three :: Int
factorial_three = factorial 3
factorial :: Int -> Int
factorial n = if n==0 then 1 else n*(factorial n-1)
```

A possible recursion scheme corresponding to this program is given below. In **Term**, we introduced the constants **One** and **Three** to represent the numbers 0 and 3, a unary function symbol **Prec** to represent the function $n \mapsto n - 1$, a binary function **Times** for the function $n, m \mapsto n \cdot m$ and the ternary function **IfZero** to capture the conditional (i.e., $n, c_1, c_2 \mapsto c_1$ if $n = 0$ and c_2 otherwise).

```
data Term = Void | One | Three | Times Term Term
          | IfZero Term Term Term
scheme_ft :: Term
scheme_ft = scheme_f Three
scheme_f :: Term -> Term
scheme_f n = IfZero n One (Times n (scheme_f (Prec n)))
```

The infinite term produced by this recursion scheme is in some sens the free-interpretation of the program. As such, it contains a lot of meaningful information about the program and in particular it allows to compute the value of the program. To formally compute the value from the infinite term, we must give, for each function f in **Term** an interpretation which is a function of the same arity in some concrete domain. In order to guaranty that the value of the infinite term can be computed, it is usually required that the domain is a complete partial order $(D, \sqsubseteq)$ (see Section 2.1.1) with a least element $\bot$ and that the interpretations are component-wise monotonous. With these assumptions, the value of the infinite term can be defined as the least upper bound of the values of the approximants of the infinite term (as computed by the function approx). If we interpret **Void** (the constant introduced by approx) as $\bot$, this sequence is directed and hence as a least upper bound which coincide with the value of the program under the same interpretation.

Intuitively, the partial order $\sqsubseteq$ can be interpreted as "less defined than" *and the least element $\bot$ as the undefined element. As our example tries to demonstrate, the requirement that the domain is an cpo can always be meet by adding a fresh least element and keeping all the other elements incomparable.*

In our example, we can simply take as domain the set of integers with a least element $\bot$ and otherwise all elements are incomparable. The interpretations follow the Haskell semantics. For instance the interpretation of **IfZero** is the ternary function:

$$n, v_1, v_2 \mapsto \begin{cases} v_1 & \text{if } n = 0, \\ v_2 & \text{if } n > 0, \\ \bot & \text{otherwise.} \end{cases}$$

Nivat proposed to use this abstraction process to automatically prove equivalence between recursive programs. He defined two programs to be equivalent if their associated schemes produce the same infinite term. In particular, two equivalent programs compute the same value and this independently of the chosen interpretation. Following the work by Courcelle [68, 69] the equivalence problem for first-order schemes was reduced to the problem of the decidability of language equivalence between deterministic pushdown automata (DPDA). Research on the equivalence for program schemes was halted until Sénizergues [133, 132] established decidability of DPDA equivalence which therefore also solved the scheme equivalence problem. At first glance, this notion of equivalence might seem too restrictive as it fails to equate many *obviously* equivalent programs. However we must keep in mind that more permissive notions of equivalence (such as saying that two program are equivalent if they compute the same value) soon become undecidable.

The use of recursion scheme for program verification saw a complet regain of interest with the works of Knapik, Niwinski and Urzyczyn [100] and Ong [118]. Ong showed that given a recursion scheme generating an infinite term t and a property φ

expressed in monadic second order (MSO) logic, we can decide if the term t satisfies the property φ. This result is important as we have seen that the infinite term t captures a lot of information about the program. Consider, for instance, a program accessing some file on a computer via the base functions `open`, `read` and `close`. We might want to check that the file is never read before it is opened. This property corresponds, on the infinite term generated by the scheme, to saying that every node labelled by `read` has an ancestor labelled by `open`. As MSO logic can express the ancestor relation, this property can be checked using Ong's result.

For practical purpose, the complexity of the decision procedure seems untractable : for an order-k scheme, the problem is non-elementary for MSO logic, k-EXPTIME-complete for the μ-calculus and still $(k-1)$-EXPTIME-complete for properties expressed by trivial tree automata. Surprisingly, Kobayashi's TRecS tool [103], which checks properties expressible by a deterministic trivial Büchi automaton (all states accepting), manages to handle schemes of order 4 with tens of lines. Since then, a number of tools have been proposed to improve the scaleability of the model-checking problem. We refer the reader to [119] for a recent survey.

Another motivation to study recursion schemes is of a more fundamental nature: recursion schemes are the current frontier in our quest to find large classes of infinite structures with good algorithmic properties. By *good algorithmic* properties, we mean structures for which properties expressed in MSO logic can be decided. The choice of the MSO logic may seem arbitrary but up to now, it presents the best compromise between expressivity and decidability. This quest started with the proof by Büchi that the MSO theory of the natural with the successor is decidable [28]. He later extended this result to the decidability of theory of any countable ordinal [30]. Then Rabin established the decidability for the infinite full binary tree [124]. This seminal result implies the decidability of MSO logic for many interesting classes of infinite graphs that can be defined in this logic starting from the full binary tree such as the context-free graphs [111], the HR-equational graphs [72], the prefix-recognizable graphs [63]. Another approach to obtain infinite structures for which we can decide MSO logic is *via* transformations that preserves the decidability of MSO logic : MSO-interpretations and MSO-transductions [71], unfolding [73], Muchnik tree-iteration [131, 143] (see Section 2.5). In [60], Caucal introduced a hierarchy of infinite graphs which contains the finite graphs and is closed by the afore mentioned operations. He later proved that the infinite terms that can be built in this fashion coincide with a subclass of the higher-order recursion schemes (called *safe* recursion schemes). As a result, almost all known structures for which MSO logic is decidable can be defined in MSO in the infinite term defined by some recursion scheme.

Of course, it is possible to construct an ad hoc *structure with a decidable MSO-theory which cannot be interpreted into a recursion scheme. For instance, it is the case for* Comb_g *with* $g(1) = 2$ *and* $g(n+1) = 2^{g(n)}$ *for* $n \geqslant 1$ *which cannot be defined in MSO logic in a term produced by a recursion scheme. Also we conjecture that countable ordinals above* ε_0 *are not definable in this fashion.*

Contributions and outline of the section

Section 3.2 presents the formal definitions of (higher-order) recursion schemes and their main decidability properties.

In Section 3.3, our first contribution is moving from *deciding* properties on recursion schemes to *synthesizing* recursion schemes satisfying some properties. We have shown decidability of two *synthesis* problem for recursion schemes.

In [26], we showed that given a recursion scheme $\mathcal{S}$ generating a term t and an MSO formula $\varphi(x)$, we can construct a new scheme $\mathcal{S}'$ generating the term t in which the nodes satisfying $\varphi(x)$ are marked. From the point of view of program verification, this could, for instance, be use to automatically annotate unsafe function

calls in a program.

In [52], we showed that given a recursion scheme $\mathcal{S}$ generating a term t and an MSO formula $\varphi(X)$, if there exists a set of nodes of t satisfying $\varphi(X)$, we can construct a scheme $\mathcal{S}'$ generating the term t marked with such a set. A possible application is the synthesis of program satisfying a given property. Imagine for instance a program with a choice operator that is left unresolved by the programmer. Using this result, we could resolve the choices left by the programmer while ensuring that the resulting program satisfies a certain property.

We also present analogous results obtained for a subclass of recursion schemes call safe recursion schemes [45, 58].

Our approach is based on compiling a recursion scheme into an extension of the model of pushdown automaton called a collapsible pushdown automaton. In [52], we obtained a simplified proof of the correcness of this compilation process.

In Section 3.4, a second contribution is a saturation-based algorithm to checks properties expressible by a deterministic trivial Büchi automaton on recursion schemes [23]. This algorithm is at the core of the C-SHORe model-checker [24] which at time of its release rivaled with the state of the art model-checkers.

In Section 3.5, a third contribution is the characterization of ordinals defined by safe recursions scheme [21], as well as the synchronization trees [2] defined by order 0 and 1 recursion schemes.

3.2 Recursion schemes and their properties

3.2.1 Definitions

In this section, we formally define *recursion schemes* as grammars for simply typed terms. This requires quite a lot a notation as we need to formalise all the notions (such as type, application and evaluation) we had *for free* in Haskell. We start with some necessary definitions about simply typed terms.

Simply typed terms

Types are generated by the grammar $\tau ::= o \mid \tau \to \tau$. Every type $\tau \neq o$ can be uniquely written as $\tau_1 \to (\tau_2 \to \cdots(\tau_n \to o)\ldots)$ where $n \geqslant 0$ and $\tau_1, \ldots, \tau_n$ are types. The number n is the *arity* of the type and is denoted by $\rho(\tau)$. To simplify the notation, we take the convention that the arrow is associative to the right and we write $\tau_1 \to \cdots \to \tau_n \to o$ (or $(\tau_1, \ldots, \tau_n, o)$ to save space).

Intuitively, the base type o corresponds to base elements (such as `Term` in Haskell). An arrow type $\tau_1 \to \tau_2$ corresponds to a function taking an argument of type τ_1 and returning an element of type τ_2. Even if there are no specific types for functions taking more than one argument, those functions are represented in their curried form. Indeed, a function taking two arguments of type o and returning a value of type o, in its curried form, has the type $o \to o \to o = o \to (o \to o)$.

The *order* measures the nesting of a type. Formally one defines $\mathrm{ord}(o) = 0$ and $\mathrm{ord}(\tau_1 \to \tau_2) = \max(\mathrm{ord}(\tau_1) + 1, \mathrm{ord}(\tau_2))$. Alternatively for any type $\tau = (\tau_1, \ldots, \tau_n, o)$ of arity $n > 0$, the order of τ is the maximum of the orders of the arguments plus one, i.e., $\mathrm{ord}(\tau) = 1 + \max\{\mathrm{ord}(\tau_i) \mid 1 \leqslant i \leqslant n\}$. For example, the type $o \to (o \to (o \to o))$ as order 1 while the type $((o \to o) \to o) \to o$ has order 3.

Let X be a set of typed symbols. For every symbol $f \in X$, and every type τ, we write

$f : \tau$ to mean that f has type τ. The set of *applicative terms* of type τ generated from X, denoted $\mathrm{Terms}_\tau(X)$, is defined by induction over the following rules. If $f : \tau$ is an element of X then $f \in \mathrm{Terms}_\tau(X)$; if $s \in \mathrm{Terms}_{\tau_1 \to \tau_2}(X)$ and $t \in \mathrm{Terms}_{\tau_1}(X)$ then the applicative term obtained by applying t to s, denoted $s\,t$, belongs to $\mathrm{Terms}_{\tau_2}(X)$. For every applicative term t, and every type τ, we write $t : \tau$ to mean that t is an applicative term of type τ. By convention, the application is considered to be left-associative, thus we write $t_1 t_2 t_3$ instead of $(t_1 t_2) t_3$. For example, assuming that $f : (o \to o) \to o \to o$, $g : o \to o$ and $c : o$, we have $g\,c : o$, $f\,g : o \to o$, $f\,g\,c = (f\,g)\,c : o$ and $f\,(f\,g)\,c : o$.

The term $M[t/x]$ obtained by replacing a variable $x : \tau$ by a term $t : \tau$ over $A \cup N$ in a term M over $A \cup N \cup V$ is defined by induction on M by taking $\varphi[t/x] = \varphi$ for $\varphi \neq x \in A \cup N \cup V$, $x[t/x] = t$ and $(t_1\ t_2)[t/x] = t_1[t/x]\,t_2[t/x]$.

The set of *argument subterms* of t, denoted $\mathrm{ASubs}(t)$, only keeps those subterms that appear as an argument. The set $\mathrm{ASubs}(t)$ is inductively defined by letting $\mathrm{ASubs}(t_1 t_2) = \mathrm{ASubs}(t_1) \cup \mathrm{ASubs}(t_2) \cup \{t_2\}$ and $\mathrm{ASubs}(f) = \varnothing$ for $f \in X$. In particular if $t = F t_1 \cdots t_n$, $\mathrm{ASubs}(t) = \cup_{i=1}^{n} (\mathrm{ASubs}(t_i) \cup \{t_i\})$. The argument subterms of $f\,(f\,g)\,c : o$ are $f\,g$, c and g.

Recursion schemes

For each type τ, we assume an infinite set V_τ of variables of type τ, such that V_{τ_1} and V_{τ_2} are disjoint whenever $\tau_1 \neq \tau_2$, and we write V for the union of those sets V_τ as τ ranges over types. We use letters $x, y, \varphi, \psi, \chi, \xi, \ldots$ to range over variables.

A (deterministic) *recursion scheme* is a 5-tuple $\mathcal{S} = (A, N, \mathcal{R}, Z, \bot)$ where

- A is a ranked alphabet of *terminals*. A symbol $f \in A$ of arity k is assigned the type $\underbrace{o \to \cdots \to o}_{k \text{ times}} \to o$.
- $\bot$ is a distinguished terminal symbol of arity 0 (and hence of ground type) that does not appear in any production rule,
- N is a finite set of typed *non-terminals*; we use upper-case letters $F, G, H, \ldots$ to range over non-terminals,
- $Z \in N$ is a distinguished *initial symbol* of type o which does not appear in any right-hand side of a production rule,
- $\mathcal{R}$ is a finite set of *production rules*, one for each non-terminal $F : (\tau_1, \cdots, \tau_n, o)$, of the form

$$F\,x_1 \cdots x_n \ \to\ e$$

  where the x_i are distinct variables with $x_i : \tau_i$ for $i \in [1, n]$ and e is a ground term in $\mathrm{Terms}((A \setminus \{\bot\}) \cup (N \setminus \{Z\}) \cup \{x_1, \ldots, x_n\})$. Note that the expressions on either side of the arrow are terms of *ground* type.

The *order* of a recursion scheme is defined to be the highest order of (the types of) its non-terminals.

If we come back to the intuition provided in the introduction, the terminals in A correspond to the functions in `Term` with $\bot$ playing the same role as `Void`, the non-terminals in N correspond to recursive functions, Z is the name of the main

function and $\mathcal{R}$ is the code of the program, giving the definition of each recursive function.

For example, let us recall below the Haskell recursion scheme `comb_n` computing Comb_n.

```
comb_n :: Term
comb_n = comb_n' (B A)
comb_n' :: Term -> Term
comb_n' x = C x (comb_n' (B x))
```

The corresponding recursion scheme is:

$$\begin{array}{lcl} Z & \to & F\,(b\,a) \\ F\,x & \to & c\,x\,(F\,(b\,x)) \end{array}$$

where the non-terminals $Z : o$ and $F : o \to o$ play the role of `comb_n` and `comb_n'` and $a : o$,$b : o \to o$ and $c : o \to o \to o$ correspond to the constructors **A**,**B** and **C** of **Term**. The order of this scheme is 1 as $\mathrm{ord}(Z) = 0$ and $\mathrm{ord}(F) = 1$.

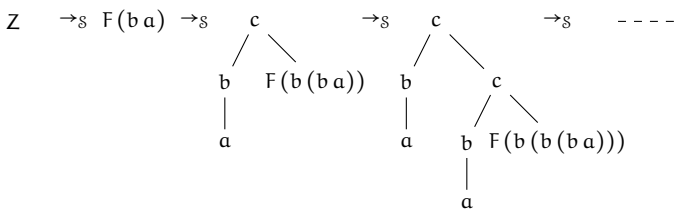
Rewriting system associated to a recursion scheme

A recursion scheme $\mathcal{S}$ induces a rewriting relation, denoted $\to_{\mathcal{S}}$, over $\mathrm{Terms}(\mathcal{A} \cup \mathcal{N})$. Informally, $\to_{\mathcal{S}}$ replaces any ground subterm $F\,t_1 \ldots t_{\rho(F)}$ starting with a non-terminal F by the right-hand side of the production rule $F\,x_1 \cdots x_n \to e$ in which the occurrences of the "formal parameter" x_i are replaced by the actual parameter t_i for $i \in [1, \rho(F)]$.

The rewriting system $\to_{\mathcal{S}}$ is defined by induction using the following rules:

- (*Substitution*) $F t_1 \cdots t_n \to_{\mathcal{S}} e[t_1/x_1, \cdots, t_n/x_n]$ where $F x_1 \cdots x_n \to e$ is a production rule of $\mathcal{S}$.
- (*Context*) If $t \to_{\mathcal{S}} t'$ then $(st) \to_{\mathcal{S}} (st')$ and $(ts) \to_{\mathcal{S}} (t's)$.

The figure below depicts the first rewriting steps of $\to_S$ starting from the initial symbol Z for our first scheme example.



In general, the rewriting relation $\to_{\mathcal{S}}$ is non-deterministic because there might be several possible independent locations to rewrite. However it can be proved to be *confluent*. That is to say that for all ground terms t,t_1 and t_2, if $t \to_{\mathcal{S}}^* t_1$ and $t \to_{\mathcal{S}}^* t_2$ (here $\to_{\mathcal{S}}^*$ denotes the transitive closure of $\to_{\mathcal{S}}$), then there exists t' such that $t_1 \to_{\mathcal{S}}^* t'$ and $t_2 \to_{\mathcal{S}}^* t'$.

Infinite term generated by a recursion scheme

Informally the *value tree* of (or the *tree generated* by) a recursion scheme $\mathcal{S}$, denoted $[\![\,\mathcal{S}\,]\!]$, is a (possibly infinite) term, *constructed from the terminals in* A, that is obtained as the *limit* of the set of all terms that can obtained by iterative rewriting from the initial symbol Z.

In the introduction, the infinite term was given for free *and we defined its approximants in order to inspect it. In the formal setting, we start by defining the set of its approximants and obtain the infinite term as its* limit.

To formally define this limit, we first introduce an operation associating with every term t over $A \cup N$ the term $t^\perp$ over $A \cup \{\perp\}$ by replacing all non-terminals ground sub-terms by the terminal symbol $\perp$.

$$a^\perp = a \text{ for } a \in A, \quad F^\perp = \perp \text{ for } F \in N \quad \text{and} \quad (st)^\perp = \begin{cases} \perp & \text{if } s^\perp = \perp, \\ (s^\perp t^\perp) & \text{otherwise.} \end{cases}$$

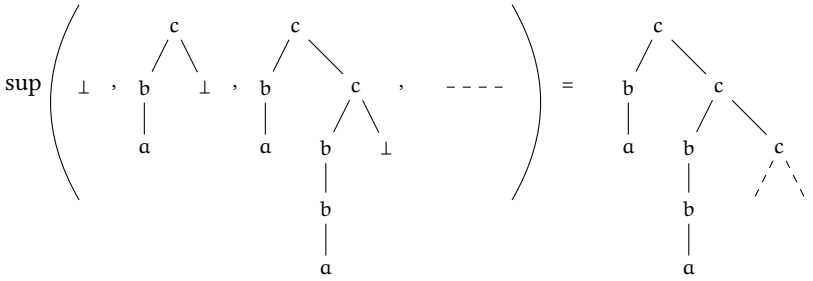
Terms built over A can be partially ordered by the *approximation order* $\preccurlyeq$ defined for all terms t and t' over A by $t \preccurlyeq t'$ if t' is obtained from t by substituting some occurrences of $\perp$ by arbitrary terms over A. For instance, $\perp \preccurlyeq a(\perp, b(\perp)) \preccurlyeq a(b(\perp), b(\perp))$.

The set of terms over A together with $\preccurlyeq$ form a directed complete partial order (see Sec. 2.1.1). Furthermore if $s \to_{\mathcal{S}} t$ then $s^\perp \preccurlyeq t^\perp$. The confluence of the relation $\to_{\mathcal{S}}$ implies that the set $\{t^\perp \mid Z \to_{\mathcal{S}}^* t\}$ is directed. Hence the *value tree* $[\![\,\mathcal{S}\,]\!]$ of $\mathcal{S}$ can be defined as its supremum,

$$[\![\,\mathcal{S}\,]\!] \;=\; \sup\{t^\perp \mid Z \to_{\mathcal{S}}^* t\}.$$

We write $\mathrm{RecTree}_n(A)$ for the class of value trees $[\![\,\mathcal{S}\,]\!]$, where $\mathcal{S}$ ranges over order-n recursion schemes.

The computation of the value tree of our example is shown below and is a expected the term Comb_n.



Remark 3.1 If the scheme contains non-productive parts, it is possible that its value tree contains the symbol $\perp$. For instance, consider the the scheme $Z \to A\,c$ and $Ax \to A(A\,x))$ with $Z : o$, $A : o \to o$ and $c : 0$. The set $\{t \mid Z \to_{\mathcal{S}}^* t\}$ consists of all terms of the form $\underbrace{A(A(\ldots A(}_{n \geqslant 2 \text{ times}} c)\ldots))$. Hence the set $\{t^\perp \mid Z \to_{\mathcal{S}}^* t\}$ is reduced to $\perp$ and the value tree is $\perp$.

3.2.2 Labelled recursion schemes

We now present labelled recursion schemes a formalism equivalent to recursion schemes but which generate (infinite) labelled transitions systems (i.e., a graph whose edges are labelled by a finite alphabet, see Sec. 2.3) instead of an infinite term. Our

main motivation to introduce this model was to the simplify the translation to finite state machines (such as pushdown automata, higher-order pushdown automata and collapsible pushdown automata, ...). As a bonus, we can naturally use labelled recursion schemes to generate trees, terms or language of words.

A deterministic *labelled recursion scheme* is a 5-tuple $\mathcal{S} = (\Sigma, N, \mathcal{R}, Z, \bot)$ where

- Σ is a finite set of labels containing two distinguished symbols $\bot$ and λ,
- N is a finite set of typed *non-terminals*,
- $Z : o \in N$ is a distinguished *initial symbol* which does not appear in any right-hand side,
- $\mathcal{R}$ is a finite set of *production rules* of the form

$$F\, x_1 \cdots x_n \xrightarrow{a} e$$

where $a \in \Sigma \setminus \{\bot\}$, $F : (\tau_1, \cdots, \tau_n, o) \in N$, the x_i are distinct variables, each x_i is of type τ_i, and e is a ground term over $(N \setminus \{Z\}) \cup \{x_1, \ldots, x_n\}$.

In addition, we require that there is at most one production rule starting with a given non-terminal and labelled by a given symbol.

We restrict here to deterministic labelled schemes to make the generated objects easier to represent. However this restriction is not crucial for most of our results.

The LTS (with silent transitions) associated with $\mathcal{S}$ has the set of ground terms over N as domain, the initial symbol Z as root, and, for all $a \in \Sigma \setminus \{\bot\}$, the relation $\xrightarrow{a}$ is defined by

$$F\, t_1 \cdots t_{\rho(F)} \xrightarrow{a} e[t_1/x_1, \ldots, t_{\rho(F)}/x_{\rho(F)}] \text{ if } F\, x_1 \cdots x_n \xrightarrow{a} e \text{ is a production rule.}$$

Tree of a labelled recursion scheme

We associate a tree with every deterministic LTS with silent transitions $\mathcal{L}$, denoted $\mathrm{Tree}(\mathcal{L})$, with directions in Σ_λ, reflecting the possible behaviours of $\mathcal{L}$ starting from the root. For this we let $\mathrm{Tree}(\mathcal{L}) \stackrel{\text{def}}{=} \{w \in \Sigma_\lambda^* \mid \exists s \in D, r \xRightarrow{w} s\}$ where $\Sigma_\lambda \stackrel{\text{def}}{=} \Sigma \setminus \{\lambda\}$. As $\mathcal{L}$ is deterministic, $\mathrm{Tree}(\mathcal{L})$ is obtained by unfolding the underlying graph of $\mathcal{L}$ from its root and contracting all λ-transitions. Figure 3.2 presents an LTS with silent transitions together with its associated tree $\mathrm{Tree}(\mathcal{L})$.

As illustrated in Fig 3.2, the tree $\mathrm{Tree}(\mathcal{L})$ does not reflect the diverging behaviours of $\mathcal{L}$ (i.e., the ability to perform an infinite sequence of silent transitions). A more informative tree can be defined in which diverging behaviours are indicated by a $\bot$-child for some fresh symbol $\bot$. This tree, denoted $\mathrm{Tree}^\bot(\mathcal{L})$, is defined by letting

Defining the tree of a rooted deterministic LTS with silent transitions as the λ-closure of the unfolding of the LTS from its root is rather standard. However, the definition of the $\mathrm{Tree}^\bot$ is non-standard.

$$\mathrm{Tree}^\bot(\mathcal{L}) \stackrel{\text{def}}{=} \mathrm{Tree}(\mathcal{L}) \cup \{w\bot \in \Sigma_\lambda^*\bot \mid \forall n \geqslant 0, r \xRightarrow{w\lambda^n} s_n \text{ for some } s_n\}.$$

The tree generated by a labelled recursion scheme $\mathcal{S}$, denoted $[\![\, \mathcal{S} \,]\!]$, is the tree $\mathrm{Tree}^\bot$ of its associated LTS.

Term of a labelled recursion scheme

Recall that we defined in Sec. 2.2.2 infinite terms over a ranked alphabet A as infinite tree over the set of directions $\vec{A}$ satisfying some local conditions. To use labelled recursion schemes to generate infinite terms over a ranked alphabet A, we use $\vec{A}$

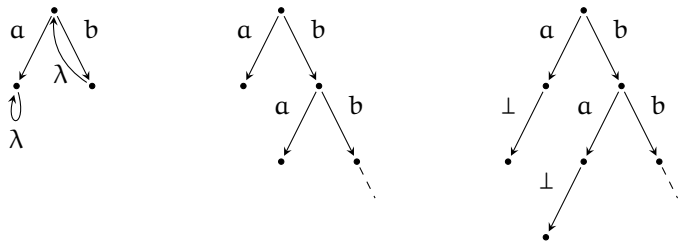


Figure 3.2. An example of labelled transition system $\mathcal{L}$ (left) together with its trees $\mathrm{Tree}(\mathcal{L})$ (center) and $\mathrm{Tree}^{\perp}(\mathcal{L})$ (right)

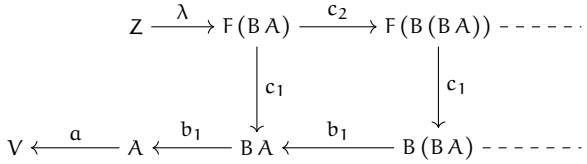
.

as set of labels and syntacticly enforce that the value tree represent a term over A. For instance, it is enough to ask that for every non-terminal $F \in N$, either there is a unique production starting with F which is labelled by λ, or there is a unique production starting with F which is labelled by some symbol c of arity 0 and whose right-hand side starts with a non-terminal that comes with no production rule in the scheme, or there exists a symbol $f \in A$ with $\rho(f) > 0$ such that the set of labels of production rules starting with F is exactly $\overrightarrow{f}$.

For example, if we want to define a labelled recursion scheme equivalent to the scheme in our running example, we introduce a non-terminal for each terminal with the same type: $C : o \to o \to o$ for c, $B : o \to o$ for b, and $A : o$ for a. We also need to add a non-terminal $V : o$ which will not appear in the left-hand side of a production rule to stop the rewriting.

$$Z \xrightarrow{\lambda} F\,(B\,A) \qquad F\,x \xrightarrow{c_1} x \qquad F\,x \xrightarrow{c_2} F\,(B\,x)$$
$$B\,x \xrightarrow{b_1} x \qquad A \xrightarrow{a} V$$

The LTS of this labelled recursion scheme is depicted below and its tree $\mathrm{Tree}^{\perp}$ is the term Comb_n.



This construction generalizes to show that recursion schemes and labelled recursion schemes are equi-expressive for generating terms.

Proposition 3.2 ([52])
The recursion schemes and the labelled recursion schemes generate the same terms. Moreover the translations are linear and preserves order and arity.

Language of a labelled recursion scheme

Given a language L over an alphabet Σ, we let $\mathrm{Pref}(L)$ denote the tree in $\mathrm{Trees}^{\infty}(\Sigma)$ containing all prefixes of words in L.

A language L over Σ is generated by a labelled recursion scheme $\mathcal{S}$ if $[\![\,\mathcal{S}\,]\!] = \mathrm{Pref}(L)$.

One could argue that the scheme generate the set of prefixes of the language and not the language. This could be remedied by adding a fresh symbol \$ and considering the language $L \cdot \$$.

In this sense the labelled scheme generating Comb_n defined previously also generate the language $\{c_2^n c_1 b_1^{n+1} a \mid n \geqslant 1\}$ over the alphabet $\{c_1, c_2, b_1, a\}$.

We present here a more involved example due to Urzyczyn. This language played an important role in demonstrating the restriction imposed by the safety constraint which we discuss in the next section.

The language U is over the alphabet $\{(,), \star\}$. A word over $\{(,)\}$ is *well bracketed* if it has as many opening brackets as closing brackets and if, for every prefix, the number of opening brackets is greater than the number of closing brackets.

The language U is defined as the set of words of the form $w\star^n$ where w is a prefix of a well-bracketed word and n is equal to $|w| - |u| + 1$, where u is the longest suffix of w that is well-bracketed. In other words, n equals 1 if w is well-bracketed, and otherwise it is equal to the index of the last unmatched opening bracket plus one.

For instance, the words $()((()) \star \star \star \star$ and $()()()\star$ belong to U. The following scheme $\mathcal{S}_U$ generates the language U.

$$\begin{array}{llll} Z \xrightarrow{\lambda} G\,(H\,X) & & F\,\varphi\,x\,y \xrightarrow{(} F\,(F\varphi x)\,y\,(Hy) \\ G\,z \xrightarrow{(} F\,G\,z\,(Hz) & & F\,\varphi\,x\,y \xrightarrow{)} \varphi\,(H\,y) \\ G\,z \xrightarrow{\star} X & & F\,\varphi\,x\,y \xrightarrow{\star} x \\ H\,u \xrightarrow{\star} u & & \end{array}$$

with $Z, X : o$, $G, H : o \to o$ and $F : (o \to o, o, o)$.

To better explain the inner workings of this scheme, let us introduce some syntactic sugar. Similarly to what we have done in the introduction, we represent every natural number by a ground term by letting $\mathbf{0} = X$ and, for all $n \geqslant 0$, $\mathbf{n+1} = H\,\mathbf{n}$. With every sequence $[\mathbf{n_1} \dots \mathbf{n_\ell}]$ of integers, we associate a term of type $o \to o$ by letting $[\,] = G$ and $[\mathbf{n_1} \dots \mathbf{n_\ell}\,\mathbf{n_{\ell+1}}] = F\,[\mathbf{n_1} \dots \mathbf{n_\ell}]\,\mathbf{n_{\ell+1}}$. Finally we write $([\mathbf{n_1} \dots \mathbf{n_\ell}], \mathbf{n})$ to denote the ground term $[\mathbf{n_1} \dots \mathbf{n_\ell}]\,\mathbf{n}$.

The scheme can be revisited as follows:

$$\begin{array}{llll} Z \xrightarrow{\lambda} ([\,], \mathbf{1}) & ([\,], \mathbf{n+1}) \xrightarrow{\star} \mathbf{0} & ([\mathbf{n_1} \dots \mathbf{n_\ell}], \mathbf{n}) \xrightarrow{\star} \mathbf{n_\ell} & \mathbf{n+1} \xrightarrow{\star} \mathbf{n} \\ ([\mathbf{n_1} \dots \mathbf{n_\ell}], \mathbf{n}) \xrightarrow{(} ([\mathbf{n_1} \dots \mathbf{n_\ell}\,\mathbf{n}], \mathbf{n+1}) & & & \\ ([\mathbf{n_1} \dots \mathbf{n_\ell}], \mathbf{n}) \xrightarrow{)} ([\mathbf{n_1} \dots \mathbf{n_{\ell-1}}], \mathbf{n+1}) & & & \end{array}$$

Let $w = w_0 \dots w_{|w|-1}$ be a prefix of a well-bracketed word. We have $Z \overset{w}{\Longrightarrow} ([\,\mathbf{n_1} \dots \mathbf{n_\ell}\,], |\mathbf{w}| + \mathbf{1})$ where $[\,n_1 \dots n_\ell\,]$ is the sequence (in increasing order) of those indices of unmatched opening brackets in w. In turn, $([\,\mathbf{n_1} \dots \mathbf{n_\ell}\,], |\mathbf{w}|) \xrightarrow{\star} \mathbf{n_\ell} \xrightarrow{\star^{n_\ell}} \mathbf{0}$. Hence, as expected, the number of $\star$ symbols is equal to 1 if w is well-bracketed (i.e., $\ell = 0$), and otherwise it is equal to the index of the last unmatched opening bracket plus one.

3.2.3 Safe schemes

Safety is a syntactic restriction, implicit in Damm's work [75], and made explicit by Knapik et al. [99, 100]. Safety restricts the use of partial application. The formal definition is rather technical as it is tailored to ensure that safe schemes can be generated by higher-order pushdown automata. Here we present a more restrictive but arguably simpler version that we called Damm-safety [52]. No generality is lost

as every safe scheme in the sense of [99, 100] can be transformed into a Damm-safe scheme of the same order and generating the same infinite term [52].

The first requirement of Damm-safety is for the scheme to be *homogeneous*. Homogeneity is a property of types. A type $(\tau_1, \dots, \tau_n, o)$ is homogeneous if $\mathrm{ord}(\tau_1) \geqslant \cdots \geqslant \mathrm{ord}(\tau_n)$ and the type τ_i is homogeneous for all $i \in [1, n]$. For instance, the type $(o \to o, o)$ is homogeneous but not the type $(o, o \to o, o)$. A scheme is *homogeneous* if the types of its non-terminals are homogeneous. Note that all our examples are homogenous. In fact, homogeneity does not restrict the expressivity of schemes. This was shown by Broadbent in his PhD thesis [22, Section 3.4] and more recently by Parys via a direct transformation on schemes [121].

A scheme is *Damm-safe* if it is homogenous and if every sub-term $X\, t_1 \dots t_\ell$ appearing as an argument in some right-hand side is such that $X : (\tau_1, \dots, \tau_n, o)$ and $t_i : \tau_i$ for $i \in [1, \ell]$ is such that $\mathrm{ord}(\tau_\ell) \neq \mathrm{ord}(\tau_{\ell+1})$. In clearer terms, if we provide one of the arguments of some order ℓ in a partial application then we have to provide all the arguments of order ℓ (and all the arguments of order greater than ℓ because of the homogeneity). We write $\mathrm{SafeRecTerm}_n(A)$ for the class of value terms $[\![\mathcal{S}]\!]$, where $\mathcal{S}$ ranges over order-n (Damm)-safe recursion scheme.

All the example given so far are Damm-safe except for the scheme presented in Sec. 3.2.2 which is unsafe. It is natural to wonder if safety is a restriction as far the infinite trees generated are considered. Parys showed that the language U (presented in Sec. 3.2.2) defined by an unsafe labelled scheme of order 2 cannot be defined by a safe labelled scheme of any order [123].

The trees of safe recursion schemes admit a very nice characterization in terms of graph transformations. As we will see in Section 3.5 this characterization is a very powerful tool which makes safe recursion schemes much easier to handle than unsafe schemes.

In Caucal [60], Caucal proposed to construct (edge-labelled) infinite graphs with decidable MSO-theories using two operations that preserve the decidability of this logic : unfolding and MSO-interpretations (see Sec.2.5). Caucal define this hierarchy using rational-inverse mappings instead of MSO-interpretations. Rational inverse mappings are a sub-class of MSO-interpretations and we proved in [59] that both definitions coincide. This lead him to introduce a hierarchy of classes of trees and graphs that bear his name. This hierarchy is composed a hierarchy of classes of (infinite) graphs $(\mathrm{Graph}_n)_{n \geqslant 0}$ and a hierarchy of classes of (infinite) trees $(\mathrm{Tree}_n)_{n \geqslant 0}$. At the first level, Tree_0 is the class of all finite trees, the rest of the hierarchy is defined as follows:

There are no differences between edge-labelled graphs and LTSs and we use both terminology interchangeably depending on the context.

$$\begin{aligned} \mathrm{Tree}_{n+1} &= [\{\mathrm{Unf}(G, s) \mid G \in \mathrm{Graph}_n \text{ and } s \text{ a vertex of } G\}] \\ \mathrm{Graph}_{n+1} &= [\{I(T) \mid T \in \mathrm{Tree}_{n+1} \text{ and } I \text{ an MSO-interpretation}\}] \end{aligned}$$

where $[X]$ denotes the class of graphs isomorphic to a graph in X.

As all finite tree have a decidable MSO-theory, all infinite graphs and trees in this hierarchy have a decidable MSO-theory.

To apply, Thm 2.5 unfoldings should be restricted to start from MSO-definable vertices but it can be shown that the two definitions lead to the same hierarchy [59].

Following the work of Knapik et al. [100], Caucal showed that deterministic trees in this hiearchy are precisely the trees generated by safe recursion schemes.

Theorem 3.3 ([62, 32])
For all $n \geqslant 1$, the deterministic trees in Tree_n are the trees generated by safe recursion schemes of order $n - 1$.

It easily follows that this hierarchy contains precisely those graphs that can be

defined in MSO logic in a safe term.

Corollary 3.4 ([59])
For all $n \geqslant 1$, the graphs in Graph_n are the graphs MSO-definable in a term generated by a safe recursion scheme of order $n-1$.

We illustrate Thm. 3.3 by constructing the term Comb_n in the Caucal hierarchy in Fig. 3.3.

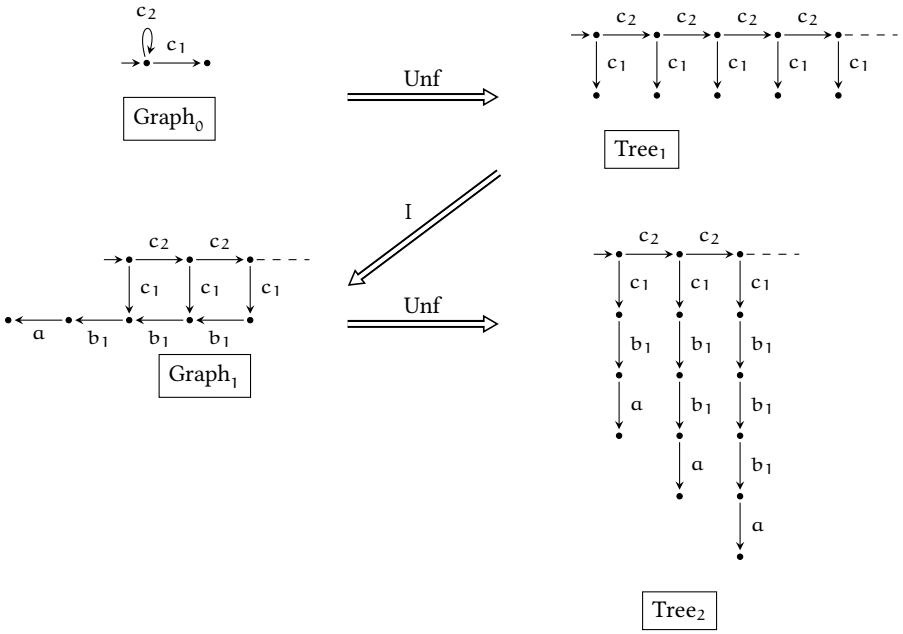


Figure 3.3. A construction of the term Comb_n in the Caucal hierarchy. The MSO-interpretation I remove the first two vertices source of a c_2-edge and otherwise leave the c_1-edges and c_2-edges unchanged. In addition, it adds b_1-edge between two consecutive vertices that are the target of a c_1-edge except for the first two where it adds an a-edge.

3.2.4 Decidability of MSO logic on recursion schemes

Since Ong's original proof of the decidability of the MSO-theory of terms generated by recursion schemes, there have been several alternative proofs of this result. In this section, we give a brief overview of the history of the result and of the different proof technics that have emerged since.

It is important to note that all the known proofs first translate the MSO-formula into an equivalent parity tree automaton. This translation is non-elementary already for finite words [137]. Hence all complexity results are given for the μ-calculus a logic which equivalent to MSO logic on (deterministic) trees but with a linear translation to alternating parity tree automata (see Section 2.4).

As we mentioned before, decidability of MSO was first proved in [99] for safe recursion schemes. They proceed by showing that safe recursion schemes can be generated by higher-order pushdown automata, an extension of pushdown automata with nested stack of stacks. The complexity of μ-calculus model-checking against

terms defined by safe schemes of order k was shown to be k-EXPTIME-complete by Cachat [31].

Decidability was then extended to order-2 unsafe schemes independently in [101] and [4]. Both proofs use a translation to what is now known as collapsible pushdown automata of order 2.

In 2006, Ong showed the decidability of MSO for arbitrary recursion schemes [118]. This result was obtained using tools from innocent game semantics (in the sense of Hyland and Ong [96]) and does not rely on an equivalent automata model for generating trees.

Theorem 3.5 ([118])
The MSO-theory of every term defined by a recursion scheme is decidable. Moreover given a recursion scheme $\mathcal{S}$ of order k and a μ-calculus formula φ, the problem of deciding if φ holds at the root of $[\![\mathcal{S}]\!]$ is k-EXPTIME-complete.

In [92, 93], Hague et al. provided an alternative proof by extending the equi-expressiveness with collapsible pushdown automata to all orders. In particular, they provide a translation of a scheme of order k into an equivalent collapsible pushdown automaton of the same order. This translation is based on *traversals* which give a very rich view of how a recursion scheme is evaluated [117]. This transformation reduces the initial problem to solving a parity game played on the LTS of a collapsible pushdown automata. To solve these games, they extended the techniques of Walukiewicz for pushdown games [144] and of [101] for order 2 collapsible pushdown automata.

Some years later, following initial ideas by Aehlig [3] and Kobayashi [103], Kobayashi and Ong [105] gave another proof of the decidability of MSO. The proof consists in showing that one can associate, with any scheme and formula, a typing system (based on intersection types) such that the scheme is typable in this system if and only if the formula holds. Typability is then reduced to solving a parity game.

As the λ-calculus is a model for functional programming, recursion schemes naturally correspond to the simply typed λ-calculus enriched with a fixpoint operator $Y^{\alpha\to\alpha}$ for each type α. This calculus is called the λY-calculus. This connection has proved very fruitfull.

Using the λY-calculus and Krivine Machines, Salvati and Walukiewicz proposed an alternative approach for the decidability of MSO [128].

As schemes can be seen as λY-terms, it is natural to wonder if the decidability of MSO can be reduced to a computation on a finite domain. For a fixed domain D, a *finitary model of the λY-calculus* gives a meaning for each terminal symbol of arity k as a function in $D^k \to D$ and a meaning for the fixpoint operator Y on functions based on D. In this model each λY-term of type τ defines a function of type τ on D. In particular, terms of ground type compute a value in D. Given an MSO-formula φ, it is possible to construct a finitary model $\mathcal{M}_\varphi$ of the λY-calculus such that for every ground λY-term $\mathcal{S}$ defining a term t, one can decide if t satisfies φ only based on value of $\mathcal{S}$ in $\mathcal{M}_\varphi$.

Under our joint-supervision with Olivier Serre, Haddad gave a first construction [91] in this spirit (based on [105]) which however works at the level of schemes and does not give an interpretation for the fixpoint operator. Two different constructions for a proper finitary model for MSO logic have independently be given by Grellois and Mélliès [88] on one side, and Salvati and Walukiewicz [127] on the other side.

The construction of Salvati and Walukiewicz was recently improved in Walukiewciz [142]. For more informations, we refer the reader to [87].

In [104], Kobayashi et al. have shown that the model-checking problem for the μ-calculus on recursion schemes is inter-reducible to the model-checking problem for higher-order modal fixpoint logic (HFL for short), an extension of the modal μ-calculus with fixpoint on higher-order functions [140]. As the model-checking of HFL is decidable, this provides another proof of Ong's result.

Recently, Parys improved Ong's result by considering an extension of MSO with an unboundedness quantifier U. The formula $\mathsf{U}X, \varphi(X)$ expresses that there are finite set of arbitrarily large size satisfying φ. Parys established the decidability of MSO+$\mathrm{U}^{\mathrm{fin}}$ on recursion schemes [121].

3.3 Synthesis of recursion schemes via automata

One of our main contribution is moving from *deciding* properties on recursion schemes to *synthesizing* recursion schemes satisfying some properties. We have shown decidability of two *synthesis* problems for recursion schemes: the marking by properties expressed in MSO logic and the effective selection of properties expressed in MSO logic.

For the marking problem, we are given a recursion scheme $\mathcal{S}$ generating an infinite term t and an MSO formula $\varphi(x)$ with one free first-order variable x and we are asked to compute a new scheme $\mathcal{S}_\varphi$ generating the infinite term t in which the set of nodes satisfying $\varphi(x)$ is *marked*.

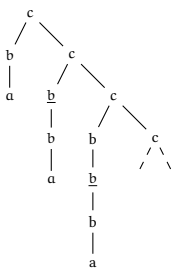
We need to precise what we mean by marking a term with a set of nodes. To mark a term over the ranked alphabet Σ, we introduced a new ranked alphabet $\underline{\Sigma}$ which contains for each symbol $f \in \Sigma$ of arity k, two symbols f and $\underline{f}$ of arity k. Intuitively, we use $\underline{f}$ instead of f when we want to mark the corresponding node.

For an infinite term t over the ranked alphabet Σ and a set U of nodes of t, the infinite term t_U obtained by marking in t the nodes is U is the unique infinite term over $\underline{\Sigma}$ such that:

- the term $\pi(t_U)$ obtained by erasing the marks (i.e., π is the morphism defined by $\pi(f) = \pi(\underline{f}) = f$ for all symbol $f \in \underline{\Sigma}$) is equal to t,
- a node u of t_U is labelled by a symbol of the form $\underline{f}$ for some f if and only if u belongs to U.

For an MSO-formula $\varphi(x)$ and a term t, we denote by t_φ the term t marked by the set of nodes satisfying $\varphi(x)$.

The term Comb_n with its nodes labelled by b which are at an even distance of the leaf marked is depicted below:



Problem 3.6 (*MSO-marking problem for recursion schemes*)
Given a recursion scheme $\mathcal{S}$ generating a tree t and an MSO-formula $\varphi(x)$, construct a recursion scheme $\mathcal{S}'$ of the same order generating t_φ.

The effective selection problem for recursion schemes is more general as it asks to mark a set satisfying a formula $\varphi(X)$ (if such a set exists).

Problem 3.7 (*Effective selection problem for recursion schemes*)
Given a recursion scheme $\mathcal{S}$ generating a tree t and an MSO-formula $\varphi(X)$ such that $t \vDash \exists X, \varphi(X)$ construct a recursion scheme $\mathcal{S}'$ generating t_U for some set U such that $t \vDash \varphi[U]$.

The effective selection problem subsumes the MSO-marking problem for a formula $\psi(x)$ by considering the formula $\varphi(X) = \forall x, x \in X \Leftrightarrow \psi(x)$. It might be tempting to think that a reduction in the other direction is also possible. One might (wrongly) assume that if the term generated by a recursion scheme satisfies a formula $\exists X, \varphi(X)$ then there exists a set U MSO-definable in t such that $t \vDash \varphi[U]$. However in Section 4.1, we present a term generated by a safe recursion scheme of order 3 and a formula φ for which this property fails.

We have shown that these two synthesis problems are decidable using the equi-expressivity between recursive schemes and collapsible pushdown automata. In Section 3.3.1, we present collapsible pushdown automata. In Section 3.3.2, we present a simplified translation of a (labelled) recursion scheme into an equivalent collapsible pushdown automaton [54]. In Section 3.3.3, we present the decidability of the MSO-marking problem. Finally in Section **??**, we present a stronger result namely the decidability of the effective selection problem for the MSO logic.

3.3.1 Collapsible pushdown automata

We first present the model of higher-order pushdown automaton which is an extension of the standard model of pushdown automaton with nested stacks of stacks. Then we introduce collapsible pushdown automata which again extend this model with links that are attached to symbol and refer to a position further down in the higher-order stack.

Higher-order stacks and their operations

Higher-order pushdown automata were introduced by Maslov [109] as a generalisation of pushdown automata. First, recall that a (order-1) pushdown automaton is a machine with a finite control together with an auxiliary storage given by a (order-1) stack whose symbols are taken from a finite alphabet. A higher-order pushdown automaton is defined in a similar way, except that it uses a higher-order stack as auxiliary storage. Intuitively, an order-n stack is a stack whose base symbols are order-$(n-1)$ stacks, with the convention that order-1 stacks are just stacks in the classical sense.

Fix a finite stack alphabet Γ and a distinguished *bottom-of-stack symbol* $\perp \notin \Gamma$. An order-1 stack is a sequence $\perp, a_1, \ldots, a_\ell \in \perp\Gamma^*$ which is denoted $[\perp a_1 \cdots a_\ell]_1$. An *order-$k$ stack* (or a *k-stack*), for $k > 1$, is a non-empty sequence $s_1, \ldots, s_\ell$ of order-$(k-1)$ stacks which is written $[s_1 \cdots s_\ell]_k$. For convenience, we may sometimes see an element $a \in \Gamma$ as an order-0 stack, denoted $[a]_0$. We let Stacks_k denote the set of

all order-k stacks and $\mathrm{Stacks} = \bigcup_{k \geqslant 1} \mathrm{Stacks}_k$ the set of all higher-order stacks. The height of the stack s denoted $|s|$ is simply the length of the sequence. We denote by $\mathrm{ord}(s)$ the order of the stack s.

For instance, the stack

$$s = [[[\bot baac]_1 [\bot bb]_1 [\bot bcc]_1 [\bot cba]_1]_2 [[\bot baa]_1 [\bot bc]_1 [\bot bab]_1]_2]_3$$

is an order-3 stack of height 2.

A *substack* of an order-1 stack $[\bot a_1 \cdots a_h]_1$ is a stack of the form $[\bot a_1 \cdots a_{h'}]_1$ for some $0 \leqslant h' \leqslant h$. A *substack* of an order-k stack $[s_1 \dots s_h]_k$, for $k > 1$, is either a stack of the form $[s_1 \dots s_{h'}]_k$ with $0 < h' \leqslant h$ or a stack of the form $[s_1 \dots s_{h'} s']_k$ with $0 \leqslant h' \leqslant h - 1$ and s' a substack of $s_{h'+1}$. We denote by $s \sqsubseteq s'$ the fact that s is a substack of s'.

In addition to the operations push_1^a and pop_1 that respectively pushes and pops a symbol in the topmost order-1 stack, one needs extra operations to deal with the higher-order stacks: the pop_k operation removes the topmost order-k stack, while the push_k duplicates it.

For an order-n stack $s = [s_1 \cdots s_\ell]_n$ and an order-k stack t with $0 \leqslant k < n$, we define $s +\!\!+ t$ as the order-n stack obtained by pushing t on top of s:

$$s +\!\!+ t = \begin{cases} [s_1 \cdots s_\ell\, t]_n & \text{if } k = n-1, \\ [s_1 \cdots (s_\ell +\!\!+ t)]_n & \text{otherwise.} \end{cases}$$

We first define the (partial) operations pop_i and top_i with $i \geqslant 1$: $\mathrm{top}_i(s)$ returns the top most $(i-1)$-stack of s, and $\mathrm{pop}_i(s)$ returns s with its top most $(i-1)$-stack removed. Formally, for an order-n stack $[s_1 \cdots s_{\ell+1}]_n$ with $\ell \geqslant 0$

$$\begin{aligned} \mathrm{top}_i([s_1 \cdots s_{\ell+1}]_n) &= \begin{cases} s_{\ell+1} & \text{if } i = n, \\ \mathrm{top}_i(s_{\ell+1}) & \text{if } i < n. \end{cases} \\ \mathrm{pop}_i([s_1 \cdots s_{\ell+1}]_n) &= \begin{cases} [s_1 \cdots s_\ell]_n & \text{if } i = n \text{ and } \ell \geqslant 1, \\ [s_1 \cdots s_\ell\, \mathrm{pop}_i(s_{\ell+1})]_n & \text{if } i < n. \end{cases} \end{aligned}$$

By abuse of notation, we let $\mathrm{top}_{\mathrm{ord}(s)+1}(s) = s$. Note that $\mathrm{pop}_i(s)$ is defined if and only if the height of $\mathrm{top}_{i+1}(s)$ is strictly greater than 1. For example, $\mathrm{pop}_2([[\bot\, a\, b]_1]_2)$ is undefined.

We now introduce the operations push_i with $i \geqslant 2$ that duplicates the top most $(i-1)$-stack of a given stack. More precisely, for an order-n stack s and for $2 \leqslant i \leqslant n$, we let $\mathrm{push}_i(s) = s +\!\!+ \mathrm{top}_i(s)$.

The last operation, push_1^a pushes the symbol $a \in \Gamma$ on top of the top most 1-stack. More precisely, for an order-n stack s and for a symbol $a \in \Gamma$, we let $\mathrm{push}_1^a(s) = s +\!\!+ [a]_0$.

For instance, the stack

$$s = [[[\bot baac]_1 [\bot bb]_1 [\bot bcc]_1 [\bot cba]_1]_2 [[\bot baa]_1 [\bot bc]_1 [\bot bab]_1]_2]_3$$

we have:

$$\begin{aligned} \mathrm{top}_3(s) &= [[\bot baa]_1 [\bot bc]_1 [\bot bab]_1]_2, \\ \mathrm{pop}_3(s) &= [[[\bot baac]_1 [\bot bb]_1 [\bot bcc]_1 [\bot cba]_1]_2]_3. \end{aligned}$$

Note that $\mathrm{pop}_3(\mathrm{pop}_3(s))$ is undefined.

We also have that

$$\begin{aligned} \mathrm{push}_2(\mathrm{pop}_3(s)) &= [[[\bot baac]_1 [\bot bb]_1 [\bot bcc]_1 [\bot cba]_1 [\bot cba]_1]_2]_3, \\ \mathrm{push}_1^c(\mathrm{pop}_3(s)) &= [[[\bot baac]_1 [\bot bb]_1 [\bot bcc]_1 [\bot cbac]_1]_2]_3. \end{aligned}$$

Stacks with links and their operations

We define a richer structure of higher-order stacks where we allow links. Intuitively, a stack with links is a higher-order stack in which any symbol may have a link that points to an internal stack below it. This link may be used later to collapse part of the stack.

Order-n stacks with links are order-n stacks with a richer stack alphabet. Indeed, each symbol in the stack can be either an element $a \in \Gamma$ (i.e., not being the source of a link) or an element $(a, \ell, h) \in \Gamma \times \{2, \cdots, n\} \times \mathbb{N}$ (i.e., being the source of a link pointing to the h-th $(\ell - 1)$-stack inside the topmost ℓ-stack).

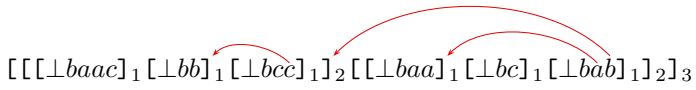
Formally, order-n stacks with links over the alphabet Γ are defined as order-n stacks over the alphabet $\Gamma \cup \Gamma \times \{2, \cdots, n\} \times \mathbb{N}$.

The stack s equals to

$$[\,[\,[\,\bot baac\,]_1\,[\,\bot bb\,]_1\,[\,\bot bc(c,2,2)\,]_1\,]_2\,[\,[\,\bot baa\,]_1\,[\,\bot bc\,]_1\,[\,\bot b(a,2,1)(b,3,1)\,]_1\,]_2\,]_3$$

is an order-3 stack with links.

To improve readability when displaying n-stacks in examples, we shall explicitly draw the links rather than using stacks symbols in $\Gamma \times \{2, \cdots, n\} \times \mathbb{N}$. For instance, we shall rather represent s as follows:



In addition to the previous operations pop_i, push_i and push_1^a, we introduce two extra operations: one to create links, and the other to collapse the stack by following a link.

Link creation is made when pushing a new stack symbol, and the target of an ℓ-link is always the $(\ell - 1)$-stack below the topmost one. Formally, we define $\mathrm{push}_1^{a,\ell}(s) = \mathrm{push}_1^{(a,\ell,h)}$ where we let $h = |\mathrm{top}_\ell(s)| - 1$ and require that $h > 1$.

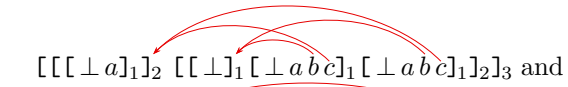
The collapse operation is defined only when the topmost symbol is the source of an ℓ-link, and results in truncating the topmost ℓ stack to only keep the component below the target of the link. Formally, if $\mathrm{top}_1(s) = (a, \ell, h)$ and $s = s' +\!\!+ [t_1 \cdots t_k]_\ell$ with $k > h$ we let $\mathrm{collapse}(s) = s' +\!\!+ [t_1 \cdots t_h]_\ell$.

For any n, we let $\mathrm{Op}_n(\Gamma)$ denote the set of all operations over order-n stacks with links.

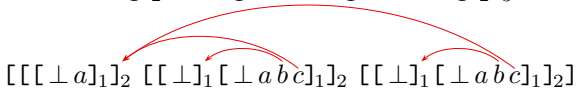
Take the 3-stack $s = [\,[\,[\,\bot a\,]_1\,]_2\ [\,[\,\bot\,]_1\,[\,\bot a\,]_1\,]_2\,]_3$. We have

$$\begin{aligned} \mathrm{push}_1^{b,2}(s) &= [\,[\,[\,\bot a\,]_1\,]_2\ [\,[\,\bot\,]_1\,[\,\bot a\, b\,]_1\,]_2\,]_3 \\ \mathrm{collapse}\,(\mathrm{push}_1^{b,2}(s)) &= [\,[\,[\,\bot a\,]_1\,]_2\ [\,[\,\bot\,]_1\,]_2\,]_3 \\ \theta = \mathrm{push}_1^{c,3}(\mathrm{push}_1^{b,2}(s)) &= [\,[\,[\,\bot a\,]_1\,]_2\ [\,[\,\bot\,]_1\,[\,\bot a\, b\, c\,]_1\,]_2\,]_3. \end{aligned}$$

Then $\mathrm{push}_2(\theta)$ and $\mathrm{push}_3(\theta)$ are respectively



and



We have $\mathrm{collapse}\,(\mathrm{push}_2(\theta)) = \mathrm{collapse}\,(\mathrm{push}_3(\theta)) = \mathrm{collapse}(\theta) = [\,[\,[\,\bot a\,]_1\,]_2\,]_3$.

Higher-order pushdown automata and collapsible pushdown automata

An *order-n (deterministic) collapsible pushdown automaton* (n-CPDA) is a 5-tuple $\mathcal{A} = (\Sigma, \Gamma, Q, \delta, q_0)$ where Σ is an input alphabet containing a distinguished symbol denoted λ, the set Γ is a stack alphabet, Q is a finite set of control states, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Gamma \times \Sigma \to Q \times \mathrm{Op}_n(\Gamma)$ is a (partial) transition function such that, for all $q \in Q$ and $\gamma \in \Gamma$, if $\delta(q, \gamma, \lambda)$ is defined then for all $a \neq \lambda$, the value $\delta(q, \gamma, a)$ is undefined, i.e., if some λ-transition can be taken, then no other transition is possible. We require δ to respect the convention that $\perp$ cannot be pushed onto or popped from the stack.

In the special case where $(p, \mathrm{collapse}) \notin \delta(q, \gamma, a)$ for all $p, q \in Q$, $\gamma \in \Gamma$ and $a \in \Sigma$, $\mathcal{A}$ is called a *higher-order pushdown automaton* (HPDA for short).

Let $\mathcal{A} = (\Sigma, \Gamma, Q, \delta, q_0)$ be an n-CPDA. A *configuration* of $\mathcal{A}$ is a pair of the form (q, s) where $q \in Q$ and s is an n-stack with link over Γ; we let $\mathrm{Config}(\mathcal{A})$ denote the set of configurations of $\mathcal{A}$ and we call $(q_0, [[\cdots[\perp]_1 \cdots]_{n-1}]_n)$ the *initial configuration*. It is then natural to associate with $\mathcal{A}$ a deterministic LTS denoted $\mathcal{L}_{\mathcal{A}} = (D, r, \Sigma, (\xrightarrow{a})_{a \in \Sigma})$ and defined as follows. We let D be the set of all configurations of $\mathcal{A}$ and r be the initial one. Then, for all $a \in \Sigma$ and all $(q, s), (q', s') \in D$ we have $(q, s) \xrightarrow{a} (q', s')$ if and only if $\delta(q, \mathrm{top}_1(s), a) = (q', \mathrm{op})$ and $s' = \mathrm{op}(s)$.

The tree generated by an n-CPDA $\mathcal{A}$, denoted $\mathrm{Tree}^{\perp}(\mathcal{A})$, is the tree $\mathrm{Tree}^{\perp}(\mathcal{L}_{\mathcal{A}})$ of its LTS.

3.3.2 Equivalence with recursion schemes

The equi-expressiveness between recursion schemes and collapsible pushdown automata was first proved in [92]. These translations generalize at all orders the order-2 translations from [101].

Theorem 3.8 ([92, 93])
For every collapsible pushdown automaton $\mathcal{A}$, there exists a recursion scheme of the same order $\mathcal{S}$ and of polynomial size such that $\mathrm{Tree}^{\perp}(\mathcal{A}) = \mathrm{Tree}^{\perp}(\mathcal{S})$.

For every recursion scheme $\mathcal{S}$, there exists a collapsible pushdown automaton $\mathcal{A}$ of the same order and of polynomial size such that $\mathrm{Tree}^{\perp}(\mathcal{S}) = \mathrm{Tree}^{\perp}(\mathcal{A})$.

In [92], the translation from collapsible to recursion scheme is quite syntactical. However the converse translation is more involved using notions of game semantics to prove its correctness. In [52, 54], we simplified the proof of this translation. The translation from [92] assumes a normal form for the schemes but up to these normalisations, but apart from this restriction, the CPDA obtained is essentially the same. Our contributions are to work direclty on schemes without normalisation and more importantly to prove the correctness of the translations without using game semantics as an intermediary tool. Independently in [129], Salvati and Walukiewicz also obtained a simplified proof of this translation using Krivine machines.

More precisely, we construct, for any labelled recursion scheme $\mathcal{S}$, a collapsible pushdown automaton $\mathcal{C}$ of the same order defining the same tree as $\mathcal{S}$ – i.e., $\mathrm{Tree}^{\perp}(\mathcal{S}) = \mathrm{Tree}^{\perp}(\mathcal{C})$. For the rest of this section, we fix a labelled recursion scheme $(\Sigma, N, \mathcal{R}, Z, \perp)$ of order $n \geqslant 1$ without silent transitions. .

If silent λ-transitions are present, we simply rename to a fresh temporary symbol and replace this temporary symbol by a λ in the resulting CPDA.

The automaton $\mathcal{C}$ has a distinguished state, denoted $q_\star$, and with the configurations of the form $(q_\star, s)$ we will associate a ground term over N denoted by $[\![s]\!]$.

Other configurations correspond to internal steps of the simulation and are only the source of silent transitions. To show that the two LTS define the same trees, we will establish that, for any reachable configuration of the form $(q_\star, s)$ and for any $a \in \Sigma$, the following holds:

- if $(q_\star, s) \xrightarrow[\mathcal{C}]{a\lambda^*} (q_\star, s')$ then $[\![s]\!] \xrightarrow[\mathcal{S}]{a} [\![s']\!]$;
- if $[\![s]\!] \xrightarrow[\mathcal{S}]{a} t$ then $(q_\star, s) \xrightarrow[\mathcal{C}]{a\lambda^*} (q_\star, s')$ and $[\![s']\!] = t$.

Hence, the main ingredient of the construction is the partial mapping $[\![\cdot]\!]$ associating with any order-n stack a ground term over N. The main difficulty is to guarantee that any rewriting rule of $\mathcal{S}$ applicable to the encoded term $[\![s]\!]$ can be simulated by applying a sequence of stack operations to s. Throughout this section, we will illustrate definitions and constructions using as a running example the order-2 scheme $\mathcal{S}_U$ defined on page 27.

Representing terms as stacks

To simplify the presentation we assume, without loss of generality, that all productions starting with a non-terminal A have the same left-hand side (i.e., they use the same variables in the same order) and that two productions starting with different non-terminals do not share any variables. Hence a variable $x \in V$ appears in a unique left-hand side $A\, x_1 \ldots, x_{\rho(A)}$ and we denote by $\mathrm{rk}(x)$ the index of x in the sequence $x_1 \cdots x_{\rho(A)}$ (i.e., $x = x_{\mathrm{rk}(x)}$).

The stack alphabet Γ consists of the initial symbol and of the right-hand sides of the rules in $\mathcal{R}$ and their argument subterms, i.e., $\Gamma \stackrel{\text{def}}{=} \{ Z \} \cup \bigcup_{F\, x_1 \cdots x_{\rho(x)} \xrightarrow{a} e} \{ e \} \cup \mathrm{ASubs}(e)$.

For the scheme $\mathcal{S}_U$ defined on page 27 (that we recall here for convinience),

$$
\begin{array}{llll}
Z & \xrightarrow{\lambda} & G\,(H\,X) & F\,\varphi\,x\,y \xrightarrow{(} F\,(F\varphi x)\,y\,(Hy) \\
G\,z & \xrightarrow{(} & F\,G\,z\,(Hz) & F\,\varphi\,x\,y \xrightarrow{)} \varphi\,(H\,y) \\
G\,z & \xrightarrow{\star} & X & F\,\varphi\,x\,y \xrightarrow{\star} x \\
H\,u & \xrightarrow{\star} & u &
\end{array}
$$

with $Z, X : o$, $G, H : o \to o$ and $F : (o \to o, o, o)$, we gets the alphabet:

$$
\begin{aligned}
\Gamma \;=\; & \{x, y, z, u, \varphi, Z\} \\
& \cup \{G\,(H\,X),\, H\,X,\, X,\, F\,(F\,\varphi\,x)\,y\,(Hy),\, F\,\varphi\,x,\, Hy,\, F\,G\,z\,(Hz),\, G,\, Hz,\, \varphi\,(Hy)\}
\end{aligned}
$$

For $\varphi \in V \cup N$, a *φ-stack* designates a stack whose top symbol starts with φ. By extension a stack s is said to be an *N-stack* (resp. a *V-stack*) if it is a φ-stack for some $\varphi \in N$ (resp. $\varphi \in V$).

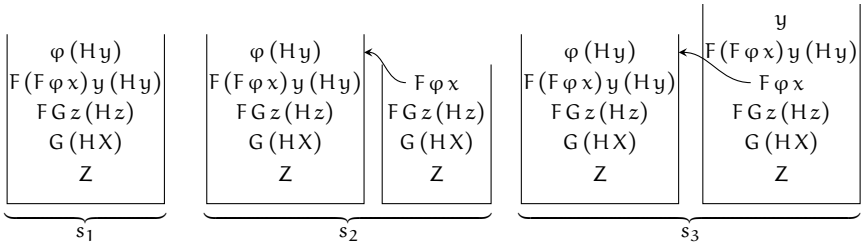
In order to represent a term in $\mathrm{Terms}(N)$, a stack over Γ must be *well-formed*, i.e., it must satisfy some syntactic conditions.

Definition 3.9 (Well-formed stack)
A non-empty stack of order-n over Γ is *well-formed* if every non-empty substack r of s satisfies the following two conditions:

- if $top_1(r)$ is not equal to Z nor to $\bot$ then $pop_1(r)$ is an A-stack for some $A \in N$ and $top_1(r)$ belongs to an A-production rule,
- if $top_1(r)$ is of type τ of order $k > 0$ then $top_1(r)$ is the source of an $(n-k+1)$-link and collapse(r) is a φ-stack for some variable $\varphi \in V$ of type τ.

We denote by WStacks the set of all well-formed stacks.

Example 3.10
For the scheme $\mathcal{S}_U$, the following order-2 stacks are well-formed.



We write $s :: t$ for $s \in$ WStacks and $t \in \Gamma$ to mean that if t belongs to the r.h.s. of a production starting with $A \in N$ then s is an A-stack. In particular, if $s \in$ WStacks then $pop_1(s) :: top_1(s)$. We denote by CStacks the set of such $s :: t$.

In Definition 3.12, we will associate, with any well-formed stack s, a ground term over N that we refer to as the *value* of s. To define this value, we first associate, with any element $s :: t$ in CStacks, a value denoted $[\![s :: t]\!]$. This value is a term over N of the same type as t. Intuitively, it is obtained by replacing the variables appearing in the term t by values encoded in the stack s, and one should therefore understand $[\![s :: t]\!]$ as the value of the term t in the context (or environment) of s.

Definition 3.11 (*The value of $s :: t$*)
For all $s :: t \in$ CStacks, we define the value of t in the context of s:

$$\begin{cases} [\![s :: t_1 t_2]\!] &= [\![s :: t_1]\!][\![s :: t_2]\!] & \text{if } t_1, t_2 \in \Gamma \\ [\![s :: A]\!] &= A & \text{if } A \in N \\ [\![s :: x]\!] &= [\![\mathrm{Arg}_{rk(x)}(s)]\!] & \text{if } x \in V \end{cases}$$

where for a stack s with top symbol $\varphi\, t_1 \cdots t_\ell$ with $\varphi \in V \cup N$ and $k \in [1, \rho(\varphi)]$, $\mathrm{Arg}_k(s)$ is defined by:

$$\begin{cases} \mathrm{Arg}_k(s) &= pop_1(s) :: t_k & \text{if } k \leqslant \ell, \\ \mathrm{Arg}_k(s) &= \mathrm{Arg}_{k-\ell}(\mathrm{collapse}(s)) & \text{otherwise.} \end{cases}$$

Let us provide some intuitions regarding the definition of $[\![s :: t]\!]$. Unsurprisingly $[\![s :: t]\!]$ is defined by structural induction on t, and the cases for the application and the non-terminal symbols are straightforward. It remains to consider the case where t is a variable x appearing in $rk(x)$-th position in the left-hand side $A\, x_1 \cdots x_{\rho(A)}$. As $s :: t \in$ CStacks, $top_1(s)$ is of the form $A t_1 \dots t_\ell$ for some $\ell \leqslant \rho(A)$. Note that ℓ is not necessarily equal to $\rho(A)$ meaning that some arguments of A might be missing. There are now two cases — that correspond to the two cases in the definition of $\mathrm{Arg}_k(s)$ — depending on whether x references to one of the t_i's (i.e., $rk(x) \leqslant \ell$) or one of the missing arguments (i.e., $rk(x) > \ell$):

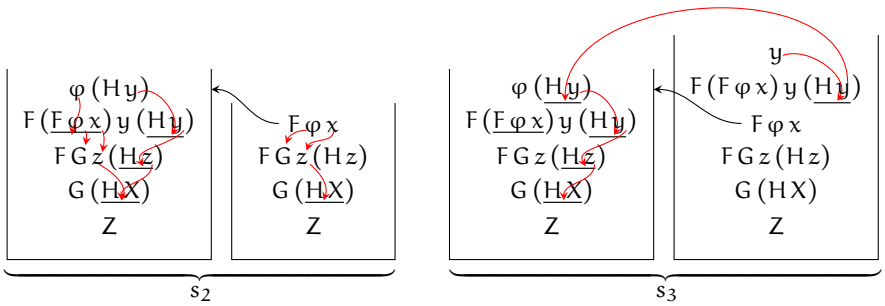
- If $\mathrm{rk}(x) \leqslant \ell$ then the term associated with x in s is equal to the term associated with $t_{\mathrm{rk}(x)}$ in $\mathrm{pop}_1(s)$, i.e., $[\![s :: x]\!] = [\![\mathrm{pop}_1(s) :: t_{\mathrm{rk}(x)}]\!]$.
- If $\mathrm{rk}(x) > \ell$ then the term $[\![s :: x]\!]$ is obtained by following the link attached to $top_1(s)$. Recall that, as s is a well-formed stack and $top_1(s)$ is not of ground type (as $\ell < \rho(A)$), there exists a link attached to $top_1(s)$. Moreover, $\mathrm{collapse}(s)$, the stack obtained by following the link, has a top-symbol of the form $\varphi\, t'_1 \dots t'_m$ for some $\varphi \in V$ and $m \geqslant 0$. Intuitively, t'_i corresponds to the $(\ell + i)$-th argument of A. If $\mathrm{rk}(x)$ belongs to $[\ell + 1, \ell + m]$ then the term $[\![s :: x]\!]$ is defined to be the term $[\![\mathrm{pop}_1(\mathrm{collapse}(s)) :: t'_{\mathrm{rk}(x)-\ell}]\!]$. If $\mathrm{rk}(x)$ is greater than $\ell + m$ then the link attached to the top symbol of $\mathrm{collapse}(s)$ is followed and the process is reiterated. As the size of the stack strictly decreases at each step this process terminates.

Now, if s is a well-formed φ-stack, its value is obtained by applying the value of φ in the context of $\mathrm{pop}_1(s)$ to the value of all its $\rho(\varphi)$ arguments. This leads to the following formal definition.

Definition 3.12 (*The value of a well-formed stack*)
The term associated with a well-formed φ-stack $s \in \mathrm{Stacks}$ with $\varphi \in N \cup V$ is

$$[\![s]\!] \stackrel{\text{def}}{=} [\![\mathrm{pop}_1(s) :: \varphi]\!][\![\mathrm{Arg}_1(s)]\!] \cdots [\![\mathrm{Arg}_{\rho(\varphi)}(s)]\!].$$

Let us consider the well-formed stacks s_2 and s_3 presented in Example 3.10. In the representation below the association between variables and their "values" are made explicit by the red arrows.



$$[\![s_1]\!] = [\![s_2]\!] = F\,G\,(H\,X)\,(H(H(H(H\,X))))$$
$$[\![s_3]\!] = H(H(H(H(H\,X))))$$

Simulating the LTS of $\mathcal{S}$ on Stacks

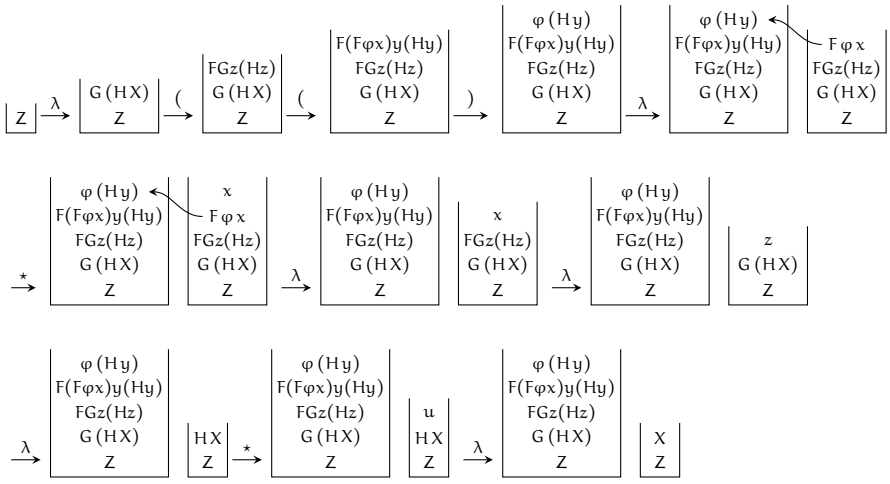
As an intermediate step, we define an LTS $\mathcal{M}$ over well-formed stacks and we prove that it generates the same tree as $\mathcal{S}$ (*i.e.* $\mathrm{Tree}^{\perp}(\mathcal{M}) = \mathrm{Tree}^{\perp}(\mathcal{S})$). From $\mathcal{M}$, a CPDA generating $\mathrm{Tree}^{\perp}(\mathcal{M})$ is then defined at the end of this section.

We let $\mathcal{M} = (\mathrm{WStacks}, [\dots[\perp Z]\dots]_n, \Sigma, (\xrightarrow[\mathcal{M}]{a})_{a \in \Sigma})$ and define the transitions as follows

- $s \xrightarrow[\mathcal{M}]{a} \mathrm{push}_1^t(s)$ if s is an A-stack with $A \in N$ and $A\,x_1 \cdots x_{\rho(A)} \xrightarrow{a} t \in \mathcal{R}$,
- $s \xrightarrow[\mathcal{M}]{\lambda} \mathrm{push}_1^t(r)$ if s is a φ-stack with $\varphi : o \in V$
  and $\mathrm{Arg}_{\mathrm{rk}(\varphi)}(\mathrm{pop}_1(s)) = r :: t$,

- $s \xrightarrow[\mathcal{M}]{\lambda} \mathrm{push}_1^{t,n-k+1}(r)$ if s is a φ-stack with $\varphi : \tau \in V$ of order $k > 0$ and $\mathrm{Arg}_{\mathrm{rk}(\varphi)}(\mathrm{pop}_1(\mathrm{push}_{n-k+1}(s))) = r :: t$.

In the figure below, we illustrate the definition of $\mathcal{M}$ on the scheme $\mathcal{S}_U$.



The first line of the definition of $\xrightarrow[\mathcal{M}]{}$ corresponds to the case of an N-stack. To simulate the application of a production rule $A\,x_1 \cdots x_n \xrightarrow{a} e$ on the term encoded by an A-stack s, we simply push the right-hand side e of the production on top of s. The correctness of this rule directly follows from the definition of $[\![\cdot]\!]$. Doing so, a term starting with a variable may be pushed on top of the stack, e.g., when applying the production rule $F\,\varphi\,x\,y \xrightarrow{)} \varphi\,(H\,y)$. Indeed, we need to retrieve the value of the head variable in order to simulate the next transition of $\mathcal{S}$: the second and third lines of the definition are normalisation rules that aim at replacing the variable at the head of the top of the stack by its definition (hence not changing the value of the associated term). By iterative application, we eventually end up with an N-stack encoding the same term and we can apply again the first rule.

Proposition 3.13
$\mathrm{Tree}^{\perp}(\mathcal{S}) = \mathrm{Tree}^{\perp}(\mathcal{M})$.

From the LTS $\mathcal{M}$, the construction of a collapsible pushdown automaton $\mathcal{C}$ is straightforward. The states of $\mathcal{C}$ are only used to keep track of the position of the argument when computing $\mathrm{Arg}_{\mathrm{rk}(\varphi)}(\cdot)$.

If the scheme we started from is safe, we proved in [52] that the links in the stack are not necessary: the collapse operation on a link of order ℓ applied to stack s always result in the stack $\mathrm{pop}_\ell(s)$. This is not surprising as safe recursion schemes are know to be equi-expressive with higher-order pushdown automaton.

Theorem 3.14 ([101])
For every higher-order pushdown automaton $\mathcal{A}$, there exists a safe recursion scheme of the same order $\mathcal{S}$ and of polynomial size such that $\mathrm{Tree}^{\perp}(\mathcal{A}) = \mathrm{Tree}^{\perp}(\mathcal{S})$.

For every safe recursion scheme $\mathcal{S}$, there exists a higher-order pushdown automaton $\mathcal{A}$ of the same order and of polynomial size such that $\mathrm{Tree}^{\perp}(\mathcal{S}) = \mathrm{Tree}^{\perp}(\mathcal{A})$.

3.3.3 MSO-marking problem for recursion schemes

In this section, we give an overview of the decidability of the MSO-marking problem for recursion scheme. In the safe case, this was first proved using graph transformations in [59]. For unsafe schemes, this results was first proof in [26] using the equivalence with collapsible pushdown automata. Since then, several alternative proofs have been given [129, 91, 88].

As we remarked in [26], the MSO-marking problem reduces to the marking problem for the μ-calculus. Indeed the marking t_φ of a term t by an MSO-formula $\varphi(x)$ can be obtained by:

As we already mentioned, in general, not all MSO-definable sets are definable in the μ-calculus.

- first marking the term by several μ-calculus formulas (theses formulas are constructed from a non-deterministic parity tree automaton recognizing $\varphi(x)$),
- then marking in the resulting term all nodes in the term that can be reached from the root by a word belonging to some regular languages R over the set of directions. We refer to this operation as a *regular-marking*.
- finally relabelling the ranked alphabet of the term.

Proposition 3.15 ([26])
If a class of trees is effectively closed under μ-calculus marking, regular-marking and relabelling then it is closed under MSO-marking.

This results can be applied to the classes of terms generated by safe and unsafe recursion schemes as they are effectively closed under regular marking and relabelling. Hence the core of problem to solve is the marking problem for the μ-calculus. We now focus on this problem.

For this, we fix a labeled recursion scheme $\mathcal{S} = (\Sigma, N, \mathcal{R}, Z, \perp)$ of order k and a μ-calculus formula φ. By Theorem 3.8, there exists a CPDA $\mathcal{A}$ of the same order such that $\mathrm{Tree}^\perp(\mathcal{S}) = \mathrm{Tree}^\perp(\mathcal{A})$. By construction, $\mathcal{A}$ has a distinguished state $q_\star$ which is the only state able to perform non-silent actions. Recall the $\mathrm{Tree}^\perp(\mathcal{A})$ is obtained by unfolding the LTS of $\mathcal{A}$ and contracting the λ-transitions.

Every node u of $\mathrm{Tree}^\perp(\mathcal{A})$ correspond to a path in the LTS of $\mathcal{A}$ ending in a configuration of the form $(q_\star, s)$, which we call *the configuration of the node* u. The fact that φ holds at a node u of $\mathrm{Tree}^\perp(\mathcal{A})$ only depends on the subtree of $\mathrm{Tree}^\perp(\mathcal{A})$ rooted at u. As a consequence, it only depends on the configuration of the node u. We let:

$$S_\varphi \stackrel{\text{def}}{=} \{s \mid \mathrm{Tree}^\perp \text{ for the LTS of } \mathcal{A} \text{ starting in } (q_\star, s) \text{ satisfies } \varphi\}.$$

Our goal is to *modify* $\mathcal{A}$ to obtain a new CPDA $\mathcal{B}$ which behaves like $\mathcal{A}$ except that when it is in a configuration $(q_\star, s)$ with $s \in S_\varphi$ it produces the marked label $\underline{a}$ instead of a. The term generated by the CPDA $\mathcal{B}$ will be $\mathrm{Tree}^\perp(\mathcal{A})_\varphi$, the term generated by $\mathcal{A}$ marked by the formula φ.

To construct $\mathcal{B}$, we will proceed in two steps:

- First, we show that the set S_φ is accepted by an appropriate model of finite automaton working on stacks of order k,
- We then show that for any set of stacks S accepted by such an automaton, it is possible to enrich any CPDA so that the top-most stack symbol reflects whether the current configuration belong to S.

We first present the construction of $\mathcal{B}$ in the safe case as it is simpler to understand and then move on to the general case.

The safe case

In the safe case, $\mathcal{A}$ is an higher-order pushdown automaton of order k. As the stacks do not contain links, a stack of order k can be represented by a well-bracketed word of nesting depth k. Formally, a stack $s = s_1, \cdots, s_\ell$ of order k is associated with the well-bracketed word of depth k, $\widetilde{s} \in (\Gamma \cup \{[,]\})^*$:

$$\widetilde{s} \stackrel{\text{def}}{=} \begin{cases} [\widetilde{s_1} \cdots \widetilde{s_\ell}] & \text{if } k \geqslant 1 \\ s & \text{if } k = 0 \text{ (i.e., } s \in \Sigma) \end{cases}$$

Definition 3.16
A set of stacks of order k over an alphabet Γ is *regular* if the corresponding language of words over $\Gamma \cup \{[,]\}$ is regular.

The notion of regularity is not strong enough to capture the set of stacks reachable from the initial configuration.

In [45], we showed that surprisingly this rather weak notion of regularity can describe the set S_φ.

Theorem 3.17 ([45])
For every μ-calculus formula φ and every higher-order pushdown automaton $\mathcal{A}$ of order k, the set S_φ is regular.

We will talk of HPDA games (resp. CPDA games) to designate two-players games played between Éloise and Abelard on the LTS of a HPDA (resp. CPDA). The owner of a configuration is given by a partition of the states of the automaton.

To prove this, we construct a two-players parity game played over the LTS of a higher-order pushdown automaton $\mathcal{B}$ such that a configuration $(q_\star, s)$ belongs to the winning region of the first player if and only if s belongs to S_φ. It remains to show that the winning region of higher-order pushdown game is regular. This is done by adapting the proof for pushdown games in [134].

The second ingredient is to show that given a regular set of stacks of order k and a higher-order pushdown automaton $\mathcal{A}$ of the same order, the HPDA $\mathcal{A}$ can be modified to obtain an HPDA $\mathcal{B}$ of the same order but which can keep track of whether the current configuration belongs to R. For this we consider a deterministic finite word automaton $\mathcal{C}$ accepting the set of stack R. The automaton $\mathcal{B}$ is obtained by enriching the stack alphabet with informations about the automaton $\mathcal{C}$. At order 1 (i.e. a simple pushdown automaton), it is enough to add the state reached by the automaton $\mathcal{C}$ when reading the stack. This operation is easily updated when performing a push_1 or a pop_1 operation. At order 2, the automaton $\mathcal{A}$ is able to copy an entire order 1 stack. To be able to maintain the information of which state is reached by $\mathcal{C}$ on the resulting stack, we need to add (and maintain) the information on how reading the full top most order-1 stack changes the state of $\mathcal{C}$ (i.e., a function from $Q_\mathcal{C}$ to $Q_\mathcal{C}$). This idea generalizes to all orders.

Proposition 3.18 ([45])
Given a regular set R of stacks of order k and a higher-order pushdown automaton $\mathcal{A}$ of order-k over a stack alphabet Γ, we can construct a higher-order pushdown $\mathcal{B}$ of the same order over a stack alphabet $\Gamma \times \Delta$ and a subset $F \subseteq \Delta$ such that:

- the LTS of $\mathcal{B}$ in which the stack alphabet $\Gamma \times \Delta$ is projected on its Γ component is equal to LTS of $\mathcal{A}$,

- for any configuration (q, s) of the LTS of $\mathcal{B}$, $top_1(s) \in \Gamma \times F$ if and only if the projection of s on its Γ component belongs to R.

By combining Proposition 3.18, Theorem 3.17 and Proposition 3.18, we obtain the decidability of the MSO-marking problem for safe schemes.

Theorem 3.19 ([59, 45])
The MSO-marking problem for safe recursion schemes is decidable.

The unsafe case

In the general case, we have to deal with CPDA and not HPDA. Hence, we need to adapt the notion of regularity for stacks with links.

For this, we introduce a class of automata with a finite state-set that can be used to recognize sets of stacks with links. As in the safe case, for a stack with links s of order k, we consider the word $\widetilde{s}$ over $\Gamma \cup \{[,]\}$ describing the content of the stack s. In order to reflect the link structure, we define a partial function $\mathtt{target}(s) : \{1, \cdots, |\widetilde{s}|\} \to \{1, \cdots, |\widetilde{s}|\}$ that assigns to every position in $\{1, \cdots, |\widetilde{s}|\}$ the index of the end of the stack targeted by the corresponding link (if exists; indeed this is undefined for $\bot$, [and]). Thus with s is associated the pair $(\widetilde{s}, \mathtt{target}(s))$.

Consider the stack with links of order 3:

.

We have $\widetilde{s} = [[[\,\bot\,\alpha]]\,[[\,\bot\,]\,[\,\bot\,\alpha\,\beta\,\gamma]]]$ and $\mathtt{target}(15) = 11$ and $\mathtt{target}(16) = 7$.

The non-deterministic version is strictly more expressive.

We consider *deterministic* finite automata working on such representations of stacks. A *finite deterministic automaton with links* reads the word $\widetilde{s}$ from left to right. On reading a letter that does not have a link (i.e. target is undefined on its index) the automaton updates its state according to the current state and the letter; on reading a letter that has a link, the automaton updates its state according to the current state, the letter and the state it was in after processing the targeted position. A run is accepting if it ends in a final state.

Definition 3.20
A set of stacks with links of order k is *regular* if it is accepted by a finite deterministic automaton with links.

Theorem 3.17 and Property 3.18 can be generalized to the unsafe case with this new notion of regularity. The proofs are however much more involved and owe a lot to the work done in [92] to solve collapsible pushdown games.

Theorem 3.21 ([26])
The MSO-marking problem for recursion schemes is decidable.

An interesting consequence of this theorem is that the class of trees defined by recursion schemes is closed by MSO-interpretations followed by an unfolding operation. As a result applying the transformations used to defined the Caucal hierarchy does not yield new trees.

Corollary 3.22
Let t be tree generated by a recursion scheme $\mathcal{S}$ and let I be an MSO-interpretation. The unfolding of $I(t)$ from any vertex u can be generated a recursion scheme of order $n+1$.

3.3.4 Effective selection for MSO on schemes

In this section, we give an overview of the decidability of the effective selection of MSO for recursion schemes. In the previous section, we have seen that the MSO-marking problem could at its core be reduce to computing a finite representation for the winning region of a collapsible pushdown game. Similarly, the selection problem reduces to computing a suitable finite representation of a winning strategy in a collapsible pushdown game.

Fix a labeled recursion scheme $\mathcal{S}$ of order k generating a term t and an MSO-calculus formula $\varphi(X)$. Assuming that $t \vDash \exists X, \varphi(X)$, our aim is to construct a scheme $\mathcal{S}'$ generating a term t marked with some set of nodes U such that $t \vDash \varphi[U]$. By Theorem 3.8, there exists a CPDA $\mathcal{A}$ of the same order such that $\mathrm{Tree}^{\perp}(\mathcal{S}) = \mathrm{Tree}^{\perp}(\mathcal{A}) = t$.

Very informally, using Rabin's theorem, we can construct from $\varphi(X)$ a tree automaton $\mathcal{B}_\varphi$ accepting the marking which satisfies φ. By a slight variation of the standard acceptance game for tree automata, we can construct a parity collapsible pushdown game $\mathbb{G}$ such that the winning strategies for the first player Éloise from the initial configuration encode accepting runs of $\mathcal{B}_\varphi$ on t and therefore a marking U of t such that $t \vDash \varphi[U]$. The game $\mathcal{G}$ is essentially a finite product between the LTS of $\mathcal{A}$ and a finite graph encoding the behaviour of $\mathcal{B}_\varphi$.

By a non-trivial adaptation of the proof of the decidability of parity collapsible pushdown games presented in [92], it is possible to show that a winning strategy can be represented by a CPDA. This CPDA reads the transitions taken so far and outputs the transition Éloise should take when it is her turn to play. A crucial property is that this CPDA is synchronized with the CPDA defining the game in the sense that they always perform the same type of operations on the stack.

Theorem 3.23 ([51])
Let $\mathcal{A}$ be a CPDA of order n and let $\mathbb{G}$ be a CPDA parity game defined on $\mathcal{A}$. If the initial configuration is winning for the first player then one can effectively construct an CPDA $\mathcal{T}$ of order n *synchronised* with $\mathcal{A}$ realising a *winning* strategy for this player from the initial configuration.

Using Theorem 3.23, it is possible to performed the synchronized product between of the CPDA $\mathcal{A}$ generating t and a CPDA $\mathcal{T}$ defining a winning strategy for the first player. The resulting CPDA, denoted $\mathcal{C}$, encodes both the tree t as well as a winning strategy for the first player in $\mathcal{G}$ (which in turns encodes a set U such that $t \vDash \varphi[U]$). With some minor modifications to $\mathcal{C}$, we finally obtain a CPDA generating a term t_U for some U such that $t \vDash \varphi[U]$. Using the translation of CPDA into labelled recursion schemes from [92], we at last solved the selection problem.

In the safe case, we prove in [58] that it is possible to synthesise positional winning strategies in higher-oder pushdown parity games. The set of configurations on which a given move is to be played are given by a *strongly regular* set of stacks. The notion of strong regularity was introduced in [33] (and independently in [81]) and his

more restrictive than the notion of regularity presented in Definition 3.16. To our knowledge, it is not known if an analogous result is possible with the notion of regularity introduced in Definition 3.16.

3.4 A saturation algorithm for collapsible pushdown automata

In the introduction, we mentioned the success obtained in practice by model-checking tools for recursion schemes. In particular, Kobayashi's TRecS tool [103], which checks properties expressible by a deterministic trivial Büchi automaton (all states accepting). One of our contribution was to show that the approach based on a translation to collapsible pushdown automata can also yield efficient algorithms. The model-checking of trivial properties on recursion schemes reduces to a (one-player) reachability game played on a collapsible pushdown automaton (CPDG for short): the goal of the reachability is to reach a distinguished error state p_{err} of the CPDA.

In [23], we presented a saturation algorithm for solving reachability CPDG. We generalized the well-known saturation algorithm for solving reachability pushdown games of [18] and [79] (see [43] for a recent survey on this method).

A precise description of the algorithm can be found in [23] but is too technical to be presented here. Instead we will try to give the main ingredients.

A standard way of computing the winning region for the first player Éloise, in a reachability game is the attractor construction. Assuming that the target set is F, we incrementally construct the winning region for Éloise by defining a increasing sequence of sets

$$\mathcal{W}_0 \subseteq \mathcal{W}_1 \subseteq \cdots$$

where $\mathcal{W}_0 = F$ is the target set and $\mathcal{W}_{i+1}$ is obtained by adding to $\mathcal{W}_i$ the vertices belonging to Éloise that have at least one successor in $\mathcal{W}_i$ and the vertices belonging to the second player Abelard, that have all their successors in $\mathcal{W}_i$. As the game we consider are of finite outdegree, the winning region for Éloise is the union of all the $\mathcal{W}_i$.

To transpose this naive approach to solve a reachability CPDG, we work symbolicly with finite automata representing sets of configurations. Starting with an automaton $\mathcal{A}_0$ accepting F, we construct a finite sequence $(\mathcal{A}_i)_{i\geqslant 0}$ where $\mathcal{A}_{i+1}$ is obtained by adding transitions and states to $\mathcal{A}_i$ following rules which ensure that the language accepted by $\mathcal{A}_{i+1}$ is included in the winning region for Éloise. As we can bound the number of possible new states, this process eventually converges to an automaton which can be shown to accept the winning region of Éloise.

It could be tempting to use the deterministic automaton with links presented in Definition 3.20 to represent sets of configurations. However this model is not suitable for this application and instead we considered an equivalent model of automaton which is alternating and explores the stack from its top rather than from its bottom.

Furthermore to simplify the correctness proof, we introduced a variant of the collapsible pushdown automata called *annotated pushdown automata*. Instead of using links to point to the context in which a symbol was first introduced, the annotated automaton directly stores the context. This small shift allows for a very clean correctness proof. However when implementing the algorithm, we reverted back to collapsible pushdown automata.

Based on the saturation algorithm of [23], we developed the C-SHORe[3] model checker for recursion schemes [24]. A naive implementation of the saturation algorithm fails already on schemes of moderate sizes and orders. It was necessary to develop non-trivial optimisations which was done mainly by Christopher Broadbent and Matthew Hague. In particular, they developed a pruning technic to reduce the size of the input CPDA. They also computed an over approximation in forward manner of the winning region to speed up the computation of the saturation algorithm which, as we described previously, performs a backward search. As result of these theoretical optimisations and the use of appropriate data structures which are presented in [25], C-SHORe was at the time of its release competitive with the other model-checkers for recursion schemes, namelly, Kobayashi et al GTRecS(2) [102] and Neatherway at al TravMC [112] tools, that are both based on intersection type inference. A recent overview of HORS model-checking was given by Ong [119].

Since C-SHORe was released, new tools were developped that perform significantly better than C-SHORe. In particular, Broadbent and Kobayashi introduced HorSat and HorSat2, which are an application of the saturation technique and initial forward analysis directly to intersection type analysis of recursion schemes [27].

3.5 Structures defined by recursion schemes

So far, we have only used recursion schemes to define deterministic trees. But recursion schemes can be used to define other types of structures. Our motivation to study these structures is two-fold. First, by considering simpler structures such as linear orders, we can better understand of the expressivity the infinite terms defined by recursion schemes. In Section 3.5.1, by considering the ordinals defined by safe recursion schemes, we obtain a new proof of the strictness of the corresponding hierarchy of terms. Our second motivation is that these structures can provide connections with neighboring fields such as process algebra, word combinatorics. In Section 3.5.2, we study synchronization trees introduced in process algebra. Also, in [21], we showed that ω-words defined by recursion schemes of order 1 are precisely the morphic words, a well-studied class in word combinatorics.

One direction of this equivalence follows from the work of Caucal in [62].

Before going further, we need to clarify how structures such as linear orders, ω-word, ... can be defined using recursion schemes. There are two main approaches to do so. The first one consists in applying an MSO-interpretation to the infinite term generated by a recursion scheme. The second one views the recursion scheme as a system of equations and computes its least solution in some continuous algebra.

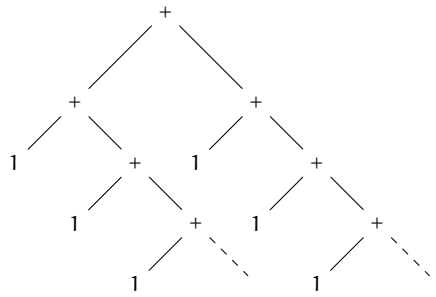
Let us exemplify these two approaches in the case of linear orders.

In the first approach, we consider the class of all linear orders that can be obtained by applying MSO-interpretation in the class of terms generated by recursion schemes. For example, consider the following recursion scheme $\mathcal{S}_{\omega\cdot 2}$ of order 0 defined by:

$$\begin{aligned} Z &\to + A\, A \\ A &\to + 1\, A \end{aligned}$$

with non-terminals Z and A of ground type and the terminal $+$ of arity 2 and the constant 1. The infinite term $t_{\omega\cdot 2}$ generated by $\mathcal{S}_{\omega\cdot 2}$ is depicted below.

[3] C-SHORe is available at http://cshore.cs.rhul.ac.uk.



The leaves of the term $t_{\omega\cdot 2}$ are naturally ordered from left to right (i.e., a leaf u is on the left of a leaf v if the word of direction leading to u from the root is lexicographically smaller than the one leading to v). The resulting linear order is called the *frontier* of the term and is definable by an MSO-interpretation Fr. The frontier of $t_{\omega\cdot 2}$ is isomorphic to $\omega + \omega = \omega \cdot 2$.

In the second approach, we consider a continuous Σ-algebra where Σ is the ranked alphabet of terminals of the recursion scheme.

We interpret the terminal $+$ as the sum for linear orders and the constant 1 as the one-element order $\mathbf{1}$ and associate with the scheme the *smallest solution* of the corresponding system of equations:

$$\begin{aligned} Z &= A + A \\ A &= 1 + A \end{aligned}$$

To formalize, the notion of smallest solution, we consider the cpo **Lin** of linear orders ordered by inclusion. The sum, as defined in Section 2.1.2, is a continuous function in **Lin**. Hence the function from $\mathbf{Lin} \times \mathbf{Lin}$ to $\mathbf{Lin} \times \mathbf{Lin}$ associated to the system of equation:

$$\begin{pmatrix} Z \\ A \end{pmatrix} \mapsto \begin{pmatrix} A + A \\ 1 + A \end{pmatrix}$$

is also continuous. Therefore it admits a least fixpoint. In our example, this least fixpoint is isomorphic to $(\omega\cdot 2, \omega)$. The solution the recursion scheme is the component corresponding to the axiom. In our case, $\omega \cdot 2$.

An equivalent characterisation of the solution of the recursion scheme is as the supremum of the interpretations of the approximants of the term $t_{\omega\cdot 2}$. Our example is only of order 0 but this approach generalizes to any order by considering the cpo of continuous functions of the appropriate types. For a complete description, we refer the reader to [130, Section 12].

Remark, that this approach does not work up-to isomorphism as it crucially relies on the names of the elements of the linear order to define the notion of supremum. If we forget this fact, we might wrongly conclude that the limit is ω and not $\omega + \omega$.

In [34], we showed that these two approaches are equivalent (meaning that they define the same class of structures) when the interpretations of the terminals are given in terms of Courcelle's VR operators [70] (extended from graphs to arbitrary relational structures). However, in general this second approach is more powerful than the first one.

3.5.1 Linear orderings

As we hinted in the introduction of this section, for linear orders, definability in MSO and interpretation in **Lin** yield the same class.

In [21], this results is only obtained for safe schemes but the same construction works for unsafe schemes.

Theorem 3.24 ([21])
For all order $k \geqslant 0$, the following classes are equal:

- the linear orders MSO-interpretable in the terms generated by unsafe (resp. safe) recursion schemes of order k,
- the frontiers of terms defined by unsafe (resp. safe) recursion schemes of order k,
- the solutions of unsafe (resp. safe) recursion schemes of order k in **Lin**.

In particular, linear orders defined by recursions schemes have a decidable MSO-theory.

In [21], we study two classes of linear orders defined in safe recursion schemes: ordinals (i.e., well-found linear orders) and scattered linear orders. We precisely characterized which ordinals are solution of safe recursion schemes and give an upper-bound on the Hausdorff rank of the scattered linear orders that can be defined this way.

At order 0, the linear orders defined by recursion scheme are also called *regular* linear orders. Indeed, these linear orders are precisely those obtained by ordering the words of a regular language using the lexicographic order. The scattered regular linear orders have an Hausdorff rank (see Section 2.1.2 for the definition) less than ω [95] and the ordinals are known to be those strictly smaller than ω^ω.

The frontiers of order-1 recursion schemes are also called *algebraic linear orders*. Indeed, these linear orders are precisely those obtained by ordering the words of a deterministic context-free language using the lexicographic order. Is was shown in [13, 12] that algebraic ordinals are precisely the ordinals strictly smaller than ω^{ω^ω}. In [14], it is shown that any scattered algebraic linear order has a Hausdorff rank strictly smaller than ω^ω. Bloom and Ésik conjectured that similar bounds can be obtained for safe recursion schemes of arbitrary orders.

In [35], we show that there exists a (non-deterministic) context-free language which when ordered with the lexicographic order has an undecidable MSO-theory.

In [20], Braud showed that all the ordinals below $\omega \Uparrow (n+1)$ (where $\omega \Uparrow 1 = \omega$ and $\omega \Uparrow (n+1) = \omega^{\omega \Uparrow n}$) are frontiers of terms generated by safe recursion schemes of order n. In [21], we showed that no other ordinals can be obtained as the frontier of a safe recursion scheme of order n.

Theorem 3.25 ([21])
The ordinals solutions of safe recursion schemes of order k over **Lin** are precisely the ordinal strictly smaller than $\omega \Uparrow (k+2)$.

The proof uses the characterization of safe schemes by graph transformations presented in Section 3.2.3 which allows to proceed by induction on the order.

As a corollary, we obtain a new proof of the strictness of the hierarchy of terms defined by safe recursion schemes that was first obtained in [78].

Corollary 3.26 ([78])
The hierarchy of terms defined by safe recursion schemes is strict.

For scattered linear orders, a sufficient condition for the frontier of the term to be scattered is for the term to be *tame* (i.e., to contain only countably many infinite branches). In fact, we proved that scattered linear orders solutions of safe recursion schemes of order k are frontiers of tame term generated by safe recursion schemes

of order k.

We also obtained a bound on the Hausdorff rank of scattered linear orders defined by safe recursion schemes.

Theorem 3.27 ([21])
The Hausdorff rank of scattered linear orders definable solutions of safe recursion schemes of order k over **Lin** is less than $\omega \Uparrow (k+1)$.

This bound is obtained by a non-trivial reduction to the case of ordinals and uses the decidability of the effective selection problem for MSO over safe schemes. Given a tame term t defined by a safe recursion scheme $\mathcal{S}$ (over the signature of **Lin**) of order k, we use the effective selection to construct a safe scheme $\mathcal{S}'$ of the same order generating a tame term t' such that:

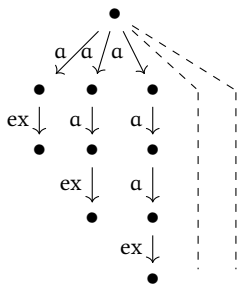
- the terms t and t' correspond to the same unordered term,
- the frontier of t' is well-ordered.

It can be shown that $\mathrm{Fr}(t)$ and $\mathrm{Fr}(t')$ have the same Hausdorff rank. By Theorem 3.25, we know that $\mathrm{Fr}(t') < \omega \Uparrow (k+2)$ and hence $r_H(\mathrm{Fr}(t')) = r_H(\mathrm{Fr}(t)) < \omega \Uparrow (k+1)$.

3.5.2 Synchronization trees

In [2], we studied synchronization trees defined by recursion scheme of orders 0 and 1. A *synchronization tree* is essentially an LTS which is a tree with a distinguished exit label ex . As depicted below, this exit label can only occur on edges whose target is a leaf. Note that these trees are non-deterministic and possibly of infinite out-degree.

The distinction between safe and unsafe schemes only appears at order 2.



In the setting of process algebras such as CCS [110] and ACP [7], synchronization trees are a classic model of process behaviour. They arise as unfoldings of labelled transition systems that describe the operational semantics of process terms and have been used to give denotational semantics to process description languages—see, for instance, [1]. In this context, the labels (except the exit label) are called actions. We denote by $\mathrm{ST}(A)$, the class of synchronization trees over the set of actions A.

We considered the solutions of recursion schemes in two algebras over $\mathrm{ST}(A)$ based respectively on Basic CCS [110] and Basic Process Algebra (BPA) [7].

The *signature* Γ of the algebra based on Basic CCS contains a binary choice operator $+$ which in the algebra corresponds to gluing two trees at the root. For each action a, Γ contains a unary action prefixing operation $a.\_$ which corresponds to prefixing the synchronization tree by an a action. Finally, it contains two constants 0 and 1 which represent respectively the empty synchronization tree and a synchronization restricted to one ex-labeled edge.

To be able to associate synchronization trees to recursion scheme over Γ, we should equip $\mathrm{ST}(A)$ with a partial order to obtain a cpo. Furthermore, we also need to fix concrete definitions for the operations in Γ that ensure that they are continuous in this cpo. Such a definition is possible by for instance considering the cpo of $\mathrm{ST}(A)$ ordered by inclusion. However fixing a naming scheme for the vertices of the various operations to ensure continuity is quite painful. In [2], we avoided this task by considering continuous categorical Γ-algebras which are a categorical generalization of the classical notion of continuous (Γ-)algebra.

Definition 3.28 (*Γ-regular and Γ-algebraic synchronization trees*)
A synchronization tree is *Γ-regular* if it is the solution of a recursion scheme of order 0 over Γ. A synchronization tree is *Γ-algebraic* if it is the solution of a recursion scheme of order 1 over Γ.

Our example of synchronization tree is the solution of the following order 1 recursion scheme over Γ:

$$Z \to F(a\,1) \qquad F\,x \to +x\,F(a\,x)$$

with Z of ground type and $F : o \to o$. Hence it is Γ-algebraic.

The *signature* Δ of the algebra based on BPA also contains a binary choice operator and the two constants 0 and 1 with the same interpretation as the Γ-algebra. There is one constant a for each action that may label the edge of a synchronization tree. This constant is interpreted as the synchronization tree corresponding to a successful computation with the action a. In addition it offers a full-blown sequential composition, or sequential product, operator. Intuitively, the sequential product $t \cdot t'$ of two synchronization trees is obtained by appending a copy of t' to the leaves of t that describe successful termination of a computation.

Definition 3.29 (*Δ-regular and Δ-algebraic synchronization trees*)
A synchronization tree is *Δ-regular* if it is the solution of a recursion scheme of order 0 over Δ.
A synchronization tree is *Δ-algebraic* if it is the solution of a recursion scheme of order 1 over Δ.

This equation could also be written as $Z = (Z \cdot a) + a$.

Our example of synchronization tree is the solution of the following order 0 recursion scheme over Δ:

$$Z \to +a\,(\cdot\,Z\,a).$$

It is therefore Δ-regular and Γ-algebraic.

In general, as the Δ-algebra can express the Γ-algebra, any Γ-regular (resp. Γ-algebraic) is Δ-regular (resp. Δ-algebraic). Solutions of recursion schemes of order n over Δ are solutions of recursion schemes of order $n + 1$.

In [2], we established a precise comparison of the classes of Γ-regular, Δ-regular, Γ-algebraic and Δ-algebraic synchronization trees. We compared them up to isomorphism, bisimulation [110, 120] and language equivalence (i.e., two synchronization are equivalent if the languages of their successful computations are equal).

Theorem 3.30
For isomorphism and bisimulation equivalence, the following hierarchy holds:

$$\Gamma\text{-regular} \subsetneq \Delta\text{-regular} \subsetneq \Gamma\text{-algebraic} \subsetneq \Delta\text{-algebraic}$$

For language equivalence, the following hierarchy holds:

$$\underbrace{\Gamma\text{-regular}}_{\text{regular languages}} \subsetneq \underbrace{\Delta\text{-regular} = \Gamma\text{-algebraic}}_{\text{context-free languages}} \subsetneq \underbrace{\Delta\text{-algebraic}}_{\text{indexed languages [5]}}$$

Finally we compared this approach to the approach by definability in MSO logic. This corresponds to comparing the previously defined classes to the first levels of the Caucal hierarchy Tree_1, Graph_1, Tree_2, Graph_2, Tree_3 and Graph_3 restricted to synchronization trees. Our findings are summarized in Figure 3.4. In particular, we have shown that unsurprisingly Γ-algebraic and Δ-algebraic are MSO-definable in the terms generated by safe recursion schemes of order 2 and hence have a decidable MSO-theory.

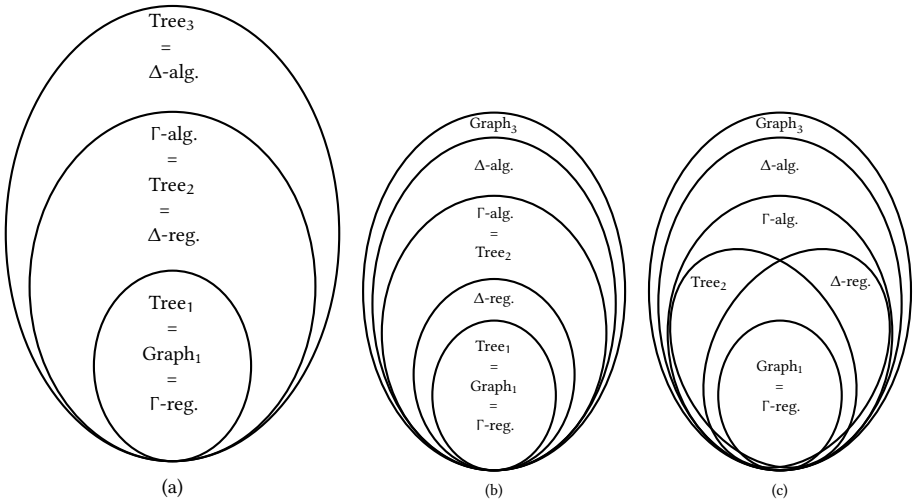


Figure 3.4. The expressiveness hierarchies up to language equivalence (a), up to bisimilarity (b) and up to isomorphism (c)

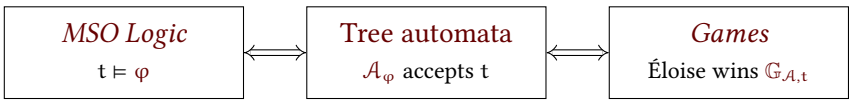
This chapter is based on [46] co-authored with Christof Löding, [42, 43] co-authored with Axel Haddad and Olivier Serre, [41, 53] co-authored with Olivier Serre and [49] co-authored with Christof Löding and Olivier Serre.

4

Extensions of tree automata

This chapter describes my contributions related to the model of tree automata on infinite trees. Roughly speaking a finite automaton on infinite trees is a finite memory machine that takes as input an infinite node-labelled binary tree and processes it in a top-down fashion as follows. It starts at the root of the tree in its initial state, and picks (possibly nondeterministically) two successor states, one per child, according to the current control state, the letter at the current node and the transition relation. Then the computation proceeds in parallel from both children, and so on. Hence, a run of the automaton on an input tree is a labelling of this tree by control states of the automaton, that should satisfy the local constraints imposed by the transition relation. A branch in a run is accepting if the ω-word obtained by reading the states along the branch satisfies some acceptance condition (typically an ω-regular condition such as a Büchi or a parity condition). Finally, a tree is accepted by the automaton if there exists a run over this tree in which *every* branch is accepting. An *ω-regular tree language* is a tree language accepted by some tree automaton equipped with a parity condition.

For monadic second order, tree automata are at the center of the equivalence between logic, automata and games as summarised in the figure below.

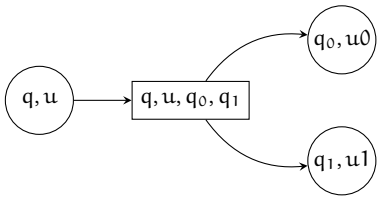


A fundamental result of Rabin is that ω-regular tree languages are the languages definable in MSO logic and form a Boolean algebra [124]. The main technical difficulty in establishing this result is to show the closure under complementation. Since the publication of this result in 1969, it has been a challenging problem to simplify this proof. A much simpler one was obtained by Gurevich and Harrington in [90] making use of two-player games for checking membership of a tree in the language accepted by the automaton: Éloise (a.k.a. *Automaton*) builds a run on the input tree while Abelard (a.k.a. *Pathfinder*) tries to exhibit a rejecting branch in the run.

Note that the idea of using games to prove this result was already proposed by Büchi in [29].

More precisely, fix a parity tree automaton $\mathcal{A} = (A, Q, q_0, \Delta, \mathrm{Acc})$ and a tree t. The acceptance game $\mathbb{G}_{\mathcal{A},t}$ for the automaton $\mathcal{A}$ on the tree t is constructed such that

Éloise wins the game from the (q_0, ε) if and only if $\mathcal{A}$ accepts t. The structure of the game is depicted below where the colour of a vertex is the colour of the state in the automaton.



for any $(q, t(u), q_0, q_1) \in \Delta$

Intuitively, a play in $\mathbb{G}_{\mathcal{A},t}$ consists in moving a pebble along a branch of t in a top-down manner: to the pebble is attached a state, and in a node u with state q, Éloise picks a transition $(q, t(u), q_0, q_1) \in \Delta$, and then Abelard chooses to move down the pebble either to $u \cdot 0$ (and update the state to q_0) or to $u \cdot 1$ (and update the state to q_1).

Another fruitful connection between automata and games is for emptiness checking. The emptiness problem for an automaton on infinite trees can be modelled as a finite pariy game where Éloise builds an input tree together with a run while Abelard tries to exhibit a rejecting branch in the run. Beyond these results, the tight connection between automata and games is one of the main tools in automata theory [138, 86, 108].

The results presented in this section build on this connection to either try enrich the expressivity of MSO logic or to better understand its limitation.

Contributions and outline

A first contribution is a new proof, obtained in [46], of the non-existence of an MSO-definable choice function on the full binary tree based on tree automata. Compared to the original proof by Gurevich and Shelah, this proof is constructive in the sense that it builds a concrete family of sets on which any MSO-definable function fails to chose a unique element.

Another contribution is the study of several notions of qualitative tree automata over infinite trees. Our approach is to relax the notion of accepting runs for tree automata. In [46], we introduced three relaxations of different nature:

- A probabilistic one which considers a run to be accepting if the set of rejecting branches in the run has measure 0.
- One based on cardinality constraints which considers a run to be accepting if the set of rejecting branches is at most countable. This idea was already present in the work of Beauquier, Nivat and Niwinski [10, 11].
- The last one is of a topological nature and considers a run to be accepting if its set of rejecting branches in *meager*. Intuitively, meager set are an *a topological approximation* of the set of measure 0. This intuition is substantiated in [141].

Thanks to this approach, we are able to define a notion of probabilistic tree automata which is algorithmically tractable [43].

In [55], we consider the game counterpart of these alternative semantics. We introduce games with a third player call Nature and we relax the notion of winning strategy in a similar manner to what we have done for the runs.

A third contribution is the decidability of the emptiness problem for parity tree automata over infinite trees enriched with the ability to test equality/disequality between the direct subtree of a node. This result generalizes to infinite trees, a result of Bogaert and Tison for finite trees [16].As remarked in [106], our result implies the decidability of MSO enriched with a predicate expressing the equality between the two subtrees of a given node.

In Section 4.1, we present our proof of the non-existence of an MSO-definable choice function. Section 4.2 regroups our results on qualitative notions of tree automata over infinite trees and probabilistic tree automata. Finally, Section 4.3 focuses on parity tree automata with equality/disequality constraints.

4.1 Choice functions on the full binary tree

In this section, we present a new proof, obtained in [46], for the non-existence of an MSO-definable choice function on the full binary tree Δ_2. A choice function f on the full binary tree is mapping associating to every non-empty set of nodes one of its elements (i.e., for $U \neq \varnothing$, $f(U) \in U$). The function is said to be MSO-definable if there exists an MSO formula $\varphi(X, x)$ such that for all non-empty set of nodes U, $f(u)$ is the only node satisfying φ when X is interpreted as U. In other terms, for all non-empty set U, $\Delta_2 \models \varphi[U, f(u)]$ and for all node v, if $\Delta_2 \models \varphi[U, v]$ then $v = f(u)$.

The non-existence of an MSO-definable choice function was first proved by Gurevich and Shelah in [89] using advance tools from set theory. It is fair to say that this original proof is not fully understood (cf. [66, p. 10]). In an unpublished note, Niwinski and Walukiewicz used this result to prove that the regular language of infinite trees labelled by $\{a, b\}$ which contain at least one node labelled by a is inherently ambiguous (i.e., there exists no unambiguous parity tree automaton accepting this language). In [46], we gave an elementary proof of the non-existence of an MSO-definable choice function using tree automata. As an added benefit, this proof constructs a concrete family of sets $(U_n)_{n \geqslant 0}$ such that for any MSO-formula $\varphi(X, x)$ there exists a set U_n such that φ fails to uniquely choose an element in U_n. These two results were combined in a joint journal publication [48].

Theorem 4.1 ([89, 46])
There exists no MSO-definable choice function of the full binary tree.

4.1.1 Proof overview

Our proof uses a special form of tree automata instead of MSO-formulas $\varphi(X, x)$. These automata work on $\{0, 1\}$-labelled trees which are meant to encode sets of nodes of Δ_2. More precisely, to every set of nodes $U \subseteq \{0, 1\}^*$ of the full binary tree, we associate the $\{0, 1\}$-labelled tree t_U defined by $t_U(v) = 1$ for $v \in U$ and 0 otherwise.

A *choice automaton* $\mathcal{A}$ is a parity tree automaton over $\{0, 1\}$-labelled trees with a distinguished set of states F. The states in F are used by the automaton to select a

node. For a tree t, the set of nodes selected by $\mathcal{A}$ in t, denoted by $\mathcal{A}(t)$, is defined by:

$$A(t) = \{v \in \{0,1\}^* \mid \rho(v) \in F \text{ for some accepting run } \rho \text{ of } \mathcal{A} \text{ on } t\}.$$

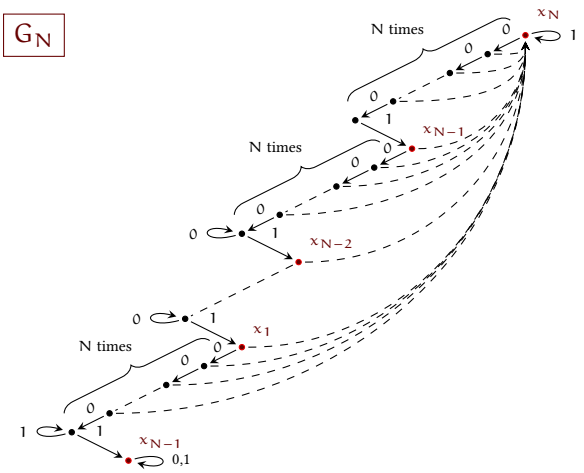
It follows from Rabin's theorem [124] that there exists an MSO-formula $\varphi(X, x)$ realizing a choice function on Δ_2 if and only if there exists a choice automaton such that for every non-empty set of nodes U, $\mathcal{A}(t_U) = \{u\}$ for some $u \in U$.

We define a family $(U_N)_{N \geqslant 1}$ of sets such that for each choice automaton $\mathcal{A}$ we can find an N such that $\mathcal{A}$ cannot select a unique element of U_N.

For $N \geqslant 1$ the set $U_N \subseteq \{0,1\}^*$ is defined by the following regular expression

$$U_N = \{0,1\}^*(0^N 0^* 1)^N \{0,1\}^*.$$

To reach a node labelled by 1 in the tree t_{U_N} it is necessary to go down to the left at least N times then proceed to the right and to repeat this process $N-1$ times. The tree t_{U_N} can be obtained by unfolding the finite graph G_N depicted below from the node x_N. In this picture, the dashed arrows represent 1-labelled edges leading back to the node x_N. The chains of 0-edges between x_{k+1} and x_k have length N. All nodes in this graph are labelled 0 except x_0, which is labelled by 1.



If N is large enough compared to the number of states of the choice automaton, we show that the automaton cannot select a unique node in U_N.

Proposition 4.2
For any choice tree automaton $\mathcal{A}$ with n states,

$$|\mathcal{A}(t_{U_N}) \cap U| \neq 1 \text{ for } N \geqslant 2^n + 1.$$

Let us give some intuition of the proof. Fix a choice automaton $\mathcal{A}$ with n states and let $N = 2^n + 1$. Assume that there exists an accepting run ρ of $\mathcal{A}$ on t_{U_N} selecting a node in U_N. We are going to construct another accepting run selecting a different node.

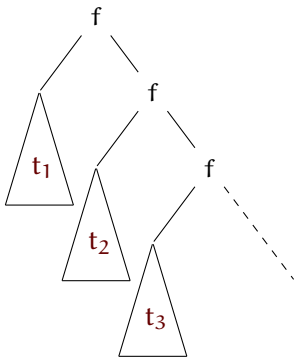
For $i \in [1, N]$, let t_i denote the tree obtained by unfolding the graph G_N from the node x_i. By the choice of N, it is possible to show that there exists $\ell < N$ such that t_ℓ and t_N are indistinguishable by $\mathcal{A}$, meaning that they are accepted from the same states of $\mathcal{A}$.

Consider the path between $x_{\ell+1}$ and x_ℓ. If we take a 1-edge before having reached the end of the 0-chain, i.e., if we take a dashed edge in the picture, then we reach a subtree of type t_N. But if we walk to the end of the 0-chain and then move to the right using a 1-edge, then we arrive at a subtree of type t_k. As t_N and t_ℓ are equivalent for $\mathcal{A}$, then $\mathcal{A}$ has no mean to identify when it enters the part where taking a 1-edge leads to a subtree of type t_ℓ. We then exploit this fact by pumping the run on this part of the tree such that we obtain another run selecting a different node.

4.1.2 Consequences on recursion schemes

Due to the very simple nature of the sets U_n, it is possible to define a term generated by a recursion scheme that contains all the trees t_{U_n} for $n \geqslant 1$

For all $n \geqslant 0$, the tree t_{U_n} can be represented by an infinite term t_n over $\{f, \underline{f}\}$ with both symbols of arity 2. The nodes in U_n are precisely those labelled by $\underline{f}$. As U_n is regular, each t_n is generated by a scheme of order 0. Now consider the term g defined as:



The term g is generated by the safe recursion scheme of order 3:

$$
\begin{array}{lcllcl}
Z & \to & F\,O\,I & V\,\varphi & \to & \varphi\,(V\,\varphi)\,T \\
F\,\psi\,\varphi & \to & f\,(V\,(\psi\,\varphi))\,(F\,(B\,\psi)\,(A\,\varphi)) & T & \to & \underline{f}\,T\,T \\
B\,\psi\,\varphi\,x\,y & \to & \varphi\,x\,(\psi\,\varphi\,x\,y) & O\,x\,y & \to & f\,x\,(O\,y\,y) \\
A\,\varphi\,x\,y & \to & f\,x\,(\varphi\,x\,y) & I\,\varphi\,x\,y & \to & \varphi\,x\,y
\end{array}
$$

where if we take $\tau \stackrel{\text{def}}{=} o \to o \to o$, we have $Z : o$, $F : (\tau \to \tau) \to \tau \to o$, $B : (\tau \to \tau) \to (\tau \to \tau)$, $A : \tau \to \tau$, $V : \tau \to o$, $T : o$, $O : \tau$ and $I : \tau \to \tau$ and $\psi : \tau \to \tau$, $\varphi : \tau$, $x : o$ and $y : o$.

Clearly the term g satisfies the MSO formula $\exists X, \varphi(X)$ expressing that there exists a set X of nodes labeled by $\underline{f}$ which has exactly one element in each subtree tree rooted on the right-most branch. However there is no set U MSO-definable in g such that $g \vDash \varphi[U]$. Indeed, from an MSO-formula $\psi(x)$ defining a suitable set U on g, we could construct a choice automaton $\mathcal{A}$ that would select a unique element in U_n for arbitrary large values of n. This would contradict Proposition 4.2.

Proposition 4.3
There exists a safe recursion scheme of order 3 generating a term t and an MSO-formula $\exists X, \varphi(X)$ such that $t \vDash \exists X, \varphi(X)$ and there exists no MSO-definable set U such that $t \vDash \varphi[U]$.

By a slight adaptation of g, it is possible to define a two-players reachability game where the first player wins from every node but for which no positional winning strategy is MSO-definable. For instance, we can give the nodes on the left-most branch to the second player and all other nodes to the first player. The target of the reachability games is then to reach a node labeled by $\underline{f}$.

Proposition 4.4
There exists a reachability game played on the term generated by a safe recursion scheme of order 3 for which there is not MSO-definable positional strategy.

4.1.3 MSO-definable well-founded orders

An immediate consequence of Theorem 4.1 is that it is not possible to define in MSO logic a total well-order on the nodes of the full binary tree. Indeed if such a formula $\varphi_{\leqslant}(x,y)$ existed, we could define a choice function in MSO by selecting the smallest element of the set:

$$\varphi(X,x) \stackrel{\text{def}}{=} x \in X \ \wedge \ \forall y \in X, \varphi_{\leqslant}(x,y).$$

Corollary 4.5
There exists no MSO-definable well-order on the nodes of of full binary tree.

In [46], we showed a stronger result: it is not possible to equip the full binary tree with any well-founded order on its nodes while preserving the decidability of the MSO-theory.

Theorem 4.6
The MSO-theory of the full-binary tree together with any well-founded order is undecidable.

In the particular case of t_{llex}, the infinite binary tree with the length-lexicographic order (recall that nodes of the infinite binary tree are the words in $\{0,1\}^*$), this result is well-known [15]. In fact, we showed that t_{llex} can be MSO-interpreted in the infinite binary tree with any well-founded order.

Theorem 4.7
There exists an MSO-interpretation I such that for every well-ordered infinite binary tree t, $I(t)$ is isomorphic to t_{llex}

As MSO-interpretations preserve the decidability of MSO, Theorem 4.6 follows from the undecidability of the MSO-theory of t_{llex} and Theorem 4.7.

4.2 Notions of qualitative tree automata

In this section, we present a summary of a series of articles [42, 43, 41, 53] whose aim was to enrich tree automata with qualitative features. One of our main goal was to define a notion of probabilistic tree automata with *good* algorithmic properties.

For ω-words, the notion of probabilistic Büchi automaton was introduced by Baier, Bertrand and Größer in [8, 9]. Quite naturally, they proceed by adding probability distributions on the transitions of a Büchi ω-automaton. For a fixed infinite word, the probabilities on the transitions induce a probability measure on the runs of the automaton. An ω-word is accepted by the automaton if the probability for a run

to be accepting is equal to 1. They show that the emptiness problem for this class of automata is decidable by a reduction to deciding a winner in partial observation Markow decision process (POMDP).

The extension to tree automata may seem, at first sight, straightforward. We associate probabilities to transitions. A tree is accepted if the probability of a run to be accepting is 1. However there are two main obstacles when working with this natural definition:

- the set of accepting runs is only measurable for the Büchi acceptance condition. In [116], it is prove that certain languages of trees accepted by deterministic co-Büchi automata are not in the Borel hierarchy and hence are not measurable. Unlike for ω-words, for infinite trees, the Büchi acceptance condition is know to be strictly weaker than the parity condition.

In [43], we wrongly announced that the set of accepting runs is measurable for the parity condition (but in fact we only provided the proof for the Büchi condition). This mistake was pointed out by Thomas Weidner and corrected in [57]

- it seems very difficult to preserve the connection with games. In particular, already for the reachability condition, we did not manage to define a correct notion of acceptance game for this model of automata. Intuitively, it is difficult to device a game that can simultaneously capture the probabilistic quantification over runs (i.e. almost all runs should be accepting) and the universal quantification over branchs of the runs (i.e. all branches of the run should be accepting).

To overcome these issues in [43], we proposed to relax the notion of accepting run for a tree automaton. Usually a run of tree automaton is said to be accepting if all branches satisfy the acceptance condition. We introduced a qualitative version which says that a run is qualitatively accepting if almost all branches are accepting (i.e., the probability that a branch of the run is accepting is 1). The idea to relax the notion of accepting run is not new and was already present in the work Beauquier, Nivat and Niwiński [10, 11] who relaxed the notion of accepting run by considering only the cardinality of the set of accepting branches in the run.

In Section 4.2.1, we present qualitative tree automata which are non-deterministic automata using the notion of qualitative acceptance for runs instead of the standard one [43]. This model although not truly probabilistic is interesting for its on sake. First they can model interesting properties cannot be captured by standard parity tree automata. For instance, if we assume that the input tree represents the possible executions of a process, a qualitative automaton can express that *bad* executions occur with probability 0. This property cannot be expressed by a standard tree automaton. Second at the time of their introduction, they appeared as a natural starting point to capture extension of MSO with probabilistic quantifiers. This intuition was partially confirmed by [17] which use an extension of qualitative tree automata called nonzero automata to prove the decidability of Thin MSO + zero.

As a side result, we also present two other notions of *qualitative* tree automata which are based on different notions of accepting runs. The first notion is based on counting the number of rejecting branches in a run and revisits the work of Beauquier, Nivat and Niwiński. The second notion is based on the topological notion of negligeable set called meager sets. In this context, a run is accepting if the set of rejecting branches is meager. As these condition can be expressed in MSO logic on trees, the corresponding models of tree automata are not more expressive than standard parity tree automata.

In 4.2.2, we introduce probabilistic trees automata both with the standard notion of acceptance for runs and with the qualitative one [43]. We will see that probabilis-

tic Büchi tree automata with the qualitative acceptance condition for runs have a decidable emptiness problem via a reduction to POMDP.

In 4.2.3, we present games with Nature which are the game counterpart of the tree automata with qualitative acceptance conditions.

4.2.1 Qualitative trees automata

In this section, we present qualitative tree automata which are based on the notion of qualitative acceptance for runs. At the end of this section, we present other notions of acceptance for runs.

To simplify the presentation, only consider complete binary infinite trees in this section

To formally define these notions, we need to fix a measure on the set $\{0,1\}^\omega$ of infinite branches of a binary tree. We consider the unique measure μ such that for all $u \in \{0,1\}^*$, $\mu(u \cdot \{0,1\}^\omega) = 2^{-|u|}$. Intuitively this measure correspond to the case where we have the same probability $\frac{1}{2}$ to go left or right down the tree.

A run is *qualitatively accepting* if almost every (in the sense of the measure μ) branch in it is accepting. More formally, consider a tree automaton $\mathcal{A}$ with an ω-regular acceptance condition Acc. A run ρ of $\mathcal{A}$ is *qualitatively accepting* if the set $\mathrm{AccBr}(\rho) = \{\pi \in \{0,1\}^\omega \mid \rho(\pi) \in \mathrm{Acc}\}$ has measure 1, i.e., $\mu(\mathrm{AccBr}(\rho)) = 1$. In [43], we proved that the set $\mathrm{AccBr}(\rho)$ is indeed measurable.

A tree t is *qualitatively accepted* by $\mathcal{A}$ if there exists a qualitatively accepting run of $\mathcal{A}$ over t and the set of all trees qualitatively accepted by $\mathcal{A}$ is denoted $\mathrm{L}_{\mathrm{Qual}}(\mathcal{A})$. Finally, a *qualitative tree language* is a set L of trees such that there exists a parity automaton $\mathcal{A}$ with $\mathrm{L}_{\mathrm{Qual}}(\mathcal{A}) = L$.

Let us give two examples of qualitative tree languages. First consider, $\mathcal{L}_a$ to be the language of $\{a,b\}$-labelled trees whose set of branches containing at least one a has measure 1. This language is recognised by the following reachability deterministic automaton $\mathcal{A} = (\{q_0, q_f\}, \{a,b\}, q_0, \Delta, \{q_f\})$ where: $\Delta = \{q_0 \xrightarrow{b} (q_0, q_0), q_0 \xrightarrow{a} (q_f, q_f), q_f \xrightarrow{a} (q_f, q_f), q_f \xrightarrow{b} (q_0, q_0)\}$.

If one considers $\mathcal{A}$ as a *Büchi* automaton, the accepted language consists of those trees whose set of branches containing infinitely many a has measure 1.

For a further example, consider the language $\mathcal{L}_1$ to be the language of trees t such that in almost every branch, there is a node u labelled by a such that the subtree t_u has only a on its leftmost branch. This language is recognised by the non-deterministic reachability automaton $\mathcal{A} = \langle A, Q, q_w, \Delta, \{q_{acc}\}\rangle$ with $A = \{a,b\}$, $Q = \{q_w, q_l, q_{acc}, q_{rej}\}$, and Δ contains the following transitions: $q_w \xrightarrow{*} (q_w, q_w)$, $q_w \xrightarrow{a} (q_l, q_{acc})$, $q_l \xrightarrow{a} (q_l, q_{acc})$, $q_l \xrightarrow{b} (q_{rej}, q_{rej})$, $q_{acc} \xrightarrow{*} (q_{acc}, q_{acc})$, $q_{rej} \xrightarrow{*} (q_{rej}, q_{rej})$ (here $*$ is a shorthand for an arbitrary letter). Intuitively, the automaton can wait in state q_w as long as it wants. Using the second transition, the automaton can guess that the node u (labeled by a) has a leftmost branch containing only a. This assumption is checked by sending on the leftmost branch the state q_l and the accepting state q_{acc} on all other branches. As long as the nodes are labelled by a the state q_l is propagated to the left son. If all nodes on the leftmost branch starting at u are labelled by a, this branch will be rejecting, but this does not affect the measure as there are only countably many such branches). If a node v labelled by b is encountered in state q_l the non-accepting state q_{rej} is propagated on all branches. This last scenario cannot occur in an accepting run as these cones of rejecting branches have a strictly positive measure. Hence the automaton is penalised for a wrong guess.

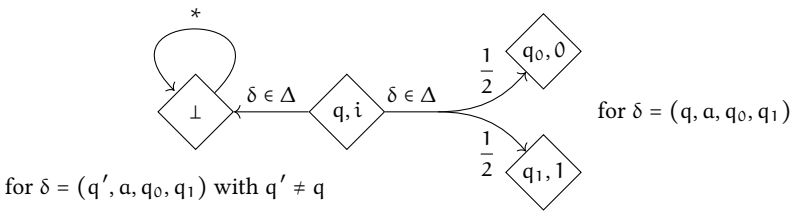


Figure 4.1. The arena $\mathcal{G}_{\mathcal{A}}$ of the emptiness game.

Using a pumping argument, we can show that the complement of $\mathcal{L}_a$ is not a qualitative tree language.

Proposition 4.8
Qualitative tree languages are closed under union, intersection but not under complementation.

Moreover qualitative tree languages are incomparable with regular tree languages.

The emptiness problem is decidable for qualitative tree automata. We construct an emptiness game very similar to the standard emptiness game for parity tree automata but where the choice of Abelard are replaced by random choices. Hence the 2-players games becomes a Markov decision process (MDP) (see Section 2.6.1).

More formally, consider the (finite) arena $\mathcal{G}_{\mathcal{A}} = \langle S, s_{ini}, \Sigma, \zeta \rangle$, depicted in Figure 4.1, where $S = Q \times \{0,1\} \cup \{q_0, \bot\}$, $s_{ini} = q_0$, $\Sigma = \Delta$ and ζ is defined as follows. First we let $d_\bot$ be the distribution defined by $d_\bot(s) = 1$ if $s = \bot$ and $d_\bot(s) = 0$ otherwise, and, for all $q_0, q_1 \in Q$, we let d_{q_0,q_1} be the distribution such that $d_{q_0,q_1}((q_0,0)) = d_{q_0,q_1}((q_1,1)) = 1/2$ and $d_{q_0,q_1}(s) = 0$ for all other $s \in S$. Then we let $\zeta((q,i),(q,a,q_0,q_1)) = d_{q_0,q_1}$, $\zeta((q,i),(q',a,q_0,q_1)) = d_\bot$ if $q \neq q'$, $\zeta(q_0,(q_0,a,q_0,q_1)) = d_{q_0,q_1}$, $\zeta(q_0,(q,a,q_0,q_1)) = d_\bot$ if $q \neq q_0$, and $\zeta(\bot,\sigma) = d_\bot$ for all $\sigma \in \Delta$. Finally, we define a colouring function ρ by letting $\rho((q,i)) = \mathrm{Col}(q)$ and $\rho(\bot) = 1$, and we call $\mathbb{G}_{\mathcal{A}} = (\mathcal{G}_{\mathcal{A}}, \mathcal{O}_\rho)$ the MDP equipped with the parity objective $\mathcal{O}_\rho$ defined by ρ.

Theorem 4.9
The language $L_{\mathrm{Qual}}(\mathcal{A})$ is non empty if and only if Éloise almost-surely wins in $\mathbb{G}_{\mathcal{A}}$ from q_0.

In particular, one can decide whether $L_{\mathrm{Qual}}(\mathcal{A}) = \varnothing$ in polynomial time. Moreover, if $L_{\mathrm{Qual}}(\mathcal{A}) \neq \varnothing$, it contains a regular tree, and such a tree can be constructed in polynomial time.

Other notions of qualitative acceptance

In [53], we defined two other notions of qualitative acceptance for runs of tree automata. Intuitively both conditions try to capture the fact the set of rejecting branches in a run is *negligible* without using probability theory.

The first one, follows the idea of Beauquier, Nivat and Niwiński [11], and considers a run to be *cardinality accepting* if it contains at most countably many rejecting branches. Clearly a countable set of rejecting branch has measure 0 (for the measure μ defined previously) but of course there are sets of rejecting branches of measure 0 which are not countable.

The second one is based on topological "bigness" and "smallness" is given by *large* and *meagre* sets respectively (see [85, 141] for a survey of the notion). The idea is to see the set of branches in a tree as a topological space by taking as basic open sets the *cones*. For a node $u \in \{0,1\}^*$, the cone $\mathrm{Cone}(u)$ is defined as $\{\pi \in \{0,1\}^\omega \mid u \sqsubseteq \pi\}$. A set of branches $B \subseteq \{0,1\}^\omega$ is *nowhere dense* if for all nodes u, there exists $v \in \{0,1\}^*$ such that no branch of B has uv as a prefix. It is *meagre* if it is the countable union of nowhere dense sets. Finally it is *large* if it is the complement of a meagre set. Meager and large sets can be characterized by Banach-Mazur games which are two-player games played on a tree.

From the modelisation point of view, the intuition is that meager sets (the complements of large sets) are somehow negligible. In [141], the authors give weight to this idea by showing that, for regular trees (i.e., those trees obtained by unfolding finite graphs), the set of branches satisfying an ω-regular condition is large if and only if it has probability 1. However they also show that in general, even for the Büchi condition and when the tree is the unfolding of a pushdown graph, this is no longer true (see [141, p. 27]).

A run is *topologically accepting* if the set of rejecting branches is meagre.

As both notions of accepting runs can be expressed in MSO logic, the languages of tree accepted by tree automata with the cardinality acceptance or the topological acceptance for runs are regular.

In [53], we provide a direct proof of this fact by constructing acceptance games for these models of automata from which we construct equivalent tree automata with the standard notion of accepting run.

4.2.2 Probabilistic trees automata

A *probabilistic tree automaton* $\mathcal{A}$ is a tuple $(A, Q, q_{ini}, \delta, \mathrm{Acc})$ where A is the *input alphabet*, Q is a finite *set of states*, $q_0 \in Q$ is the *initial state*, $\mathrm{Acc} \subseteq Q^\omega$ is the *acceptance condition* and δ is a mapping from $Q \times A \times Q \times Q$ to $[0,1]$ such that for all $q \in Q$ and $a \in A$, $\sum_{q_0,q_1 \in Q} \delta(q,a,q_0,q_1) = 1$. Intuitively, the value $\delta(q,a,q_0,q_1)$ is the probability for a transition $q \xrightarrow{a} (q_0,q_1)$ to be used by the automaton when it is in state q and reads the symbol a.

This probability distribution on the transitions induces a probability measure on the set of runs of $\mathcal{A}$. In this setting, a *run* of $\mathcal{A}$ is simply a Q-labeled tree whose root is labeled by the initial state q_0. We denote by $\mathrm{Runs}(\mathcal{A})$ (or simply Runs if $\mathcal{A}$ is clear from the context) the set of all runs of $\mathcal{A}$. We refer to [43] for a precise definition of the probability measure μ_t on the measurable space $(\mathrm{Runs}, \mathcal{F}_R)$. It is defined as expected on finite partial runs as the product of all the probabilities of the transitions appearing in this run and it is extended to the σ-algebra $\mathcal{F}_R$ generated by the cylinders associated to the partial runs. Note that both μ_t and $(\mathrm{Runs}, \mathcal{F}_R)$ depend on t.

We denote by $\mathrm{AccRuns}(\mathcal{A})$ the set of accepting runs of $\mathcal{A}$ and by $\mathrm{QualAccRuns}(\mathcal{A})$ the set of qualitatively accepting runs of $\mathcal{A}$.

Proposition 4.10 ([43, 57])
For all probabilistic tree automata $\mathcal{A}$ with an ω-regular acceptance condition, the $\mathrm{QualAccRuns}(\mathcal{A})$ are measurable.

However the set $\mathrm{AccRuns}(\mathcal{A})$ is only measurable in general for the Büchi condition.

As we did not obtain any positive results for probabilistic tree automata with the classical notion of acceptance for runs, we focus on probabilistic tree automata equipped with the qualitative notion of acceptance for runs.

A tree t is *(almost-surely) accepted* by $\mathcal{A}$ with the qualitative semantics if almost all runs of $\mathcal{A}$ on t are qualitatively accepting, i.e., $\mu_t(\mathrm{QualAccRuns}(\mathcal{A})) = 1$. We denote by $L^{=1}_{\mathrm{Qual}}(\mathcal{A})$ the set of trees accepted by $\mathcal{A}$ with the qualitative semantics for runs. More formally, $L^{=1}_{\mathrm{Qual}}(\mathcal{A}) = \{t \mid \mu_t(\mathrm{QualAccRuns}(\mathcal{A})) = 1\}$.

Not too surprisingly, the languages accepted by probabilistic tree automata can be shown to be incomparable with regular tree languages and qualitative tree languages.

Before proceeding, let us give some examples of languages accepted by probabilistic tree automata. For an ω-word language $L \subseteq \{a,b\}^\omega$, we denote by $\mathrm{Path}^{=1}(L)$ the set of trees labeled by $\{a,b\}$ with almost all their branch labels in L (i.e., $\mu(\{\pi \in \mathrm{Br} \mid t(\pi) \in L\}) = 1$). It is easy to see that, for any ω-regular language L, the tree language $\mathrm{Path}^{=1}(L)$ is a qualitative tree language.

More interestingly, if L is almost-surely accepted by a probabilistic ω-word automaton with an ω-regular acceptance condition, we can show that $\mathrm{Path}^{=1}(L)$ is accepted by a probabilistic tree automaton (with the qualitative semantics).

Proposition 4.11
Given a probabilistic ω-word automaton $\mathcal{B}$ with an ω-regular acceptance condition, there exists a probabilistic tree automaton $\mathcal{A}$ with the same acceptance condition such that $L^{=1}_{\mathrm{Qual}}(\mathcal{A})$ is equal to $\mathrm{Path}^{=1}(L^{=1}(\mathcal{B}))$.

This proposition in particular shows that the emptiness problem for probabilistic ω-word automata can be reduced to the same problem for probabilistic tree automata for the same acceptance condition. In particular, we inherit the negative results from [8].

Theorem 4.12
The emptiness problem for probabilistic co-Büchi tree automaton $\mathcal{A}$ is undecidable.

On the positive side, we showed that the emptiness problem is decidable for the Büchi acceptance condition. We reduce this problem to deciding almost-surely winning in a POMDP, and the reduction works for any ω-regular acceptance condition. However, the corresponding decision problem on POMDPs is only decidable for the Büchi condition. Hence we only obtain decidability in the Büchi case.

Let $\mathcal{A} = (Q, A, q_0, \delta, \mathrm{Acc})$ be a probabilistic automaton with an ω-regular acceptance condition and let $\Delta = Q \times Q \times Q$.

Define a POMDP $\mathbb{G}_{\mathcal{A}} = (\mathcal{G}, \sim, \mathcal{O})$ as follows. The arena $\mathcal{G}$ is equal to $\langle S, s_{\mathrm{ini}}, A, \zeta\rangle$ where $S = Q \times \{0,1,\bot\} \times (\Delta \cup \{\bot\})$, $s_{\mathrm{ini}} = (q_0, \bot, \bot)$ and ζ is defined as follows. For all $a \in A$ and $(p,x,t) \in S$, $\zeta((p,x,t),a)$ is the distribution that assigns $\frac{1}{2}\delta(p,a,q_0,q_1)$ to $(q_y, y, (p,q_0,q_1))$ where $y = 0,1$ and 0 to all other state. The objective $\mathcal{O}$ is the set of plays for which the sequence of states obtained when projecting on the first component belongs to Acc. The equivalence relation $\sim$ is defined by $(q,x,t) \sim (q',x',t')$ iff $x = x'$. Intuitively, in $\mathbb{G}_{\mathcal{A}}$, Éloise describes a branch along a tree and Random builds a piece of run along this branch. As Éloise does not observe the state in the run constructed by Random, it does not influence her choice for the branch.

Theorem 4.13
Let $\mathcal{A}$ be a probabilistic tree automaton with an ω-regular acceptance condition.

The language $\mathrm{L}^{=1}_{\mathrm{Qual}}(\mathcal{A})$ is non-empty if and only if Éloise almost-surely wins in $\mathbb{G}_{\mathcal{A}}$.

For the Büchi acceptance condition, this leads to a decidability result for the emptiness problem. From the existence of positional winning strategies in Büchi POMDP, we derive the existence of a regular tree in the accepted language.

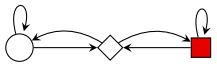
Corollary 4.14
Let $\mathcal{A}$ be a probabilistic Büchi tree automaton. Deciding emptiness of $\mathrm{L}^{=1}_{\mathrm{Qual}}(\mathcal{A})$ is an ExpTime-complete problem. Moreover, if $\mathrm{L}^{=1}_{\mathrm{Qual}}(\mathcal{A}) \neq \varnothing$, it contains a regular tree.

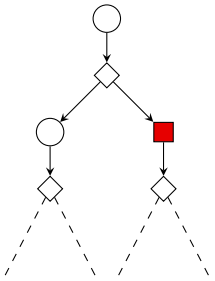
4.2.3 Games with Nature

In this section, we present the game counter-part of the qualitative notions of acceptance for tree automata presented in Section 4.2.1. This idea was originally suggested to us by Damian Niwiński.

To achieve this goal, in [41], we added to Éloise and Abelard, a third player called Nature. Intuitively, Nature is seen as an uncontrolled player and its choices are left *uninterpreted*. The semantics of Nature will be given when we define the notion of winning strategy. The notions of strategy, play, ... are defined as usual (see Section 2.6.1).

In a game $\mathbb{G}$ starting from some initial vertex v_0, with Éloise, Abelard and Nature with an objective $\Omega \subseteq V^\omega$, if we fix a strategy $\varphi_{\mathbf{E}}$ for Éloise and a strategy $\varphi_{\mathbf{A}}$ for Abelard, the outcome is no longer a single play but a set of plays, denoted by $\mathrm{Outcomes}^{\varphi_{\mathbf{E}},\varphi_{\mathbf{A}}}$, which correspond to the different choices of Nature. By definition, $\mathrm{Outcomes}^{\varphi_{\mathbf{E}},\varphi_{\mathbf{A}}}$ has a tree structure where the branching nodes correspond to nodes belong to Nature. The figure below present (on the left) finite Büchi game played between Éloise (whose vertex is a represented by a circle), Abelard (whose vertex is represented by a square) and Nature (whose vertex is represented by a diamond). The aim is to visit infinitely often the vertex colored in red. On the right, it present the tree $\mathrm{Outcomes}^{\varphi_{\mathbf{E}},\varphi_{\mathbf{A}}}$ for the strategy $\varphi_{\mathbf{E}}$ of Éloise consisting in always moving to the vertex own by Nature and the strategy $\varphi_{\mathbf{A}}$ of Abelard also consisting of always moving to vertex own by Nature.





The standard notion of winning strategy for Éloise would be that for all strategy $\varphi_{\mathbf{A}}$ of Abelard all the element of $\mathrm{Outcomes}^{\varphi_{\mathbf{E}},\varphi_{\mathbf{A}}}$ belong to Ω.

We consider relaxed notions of winning strategies similar to the qualitative notion of accepting runs we introduced at the end of Section 4.2.1.

A strategy $\varphi_{\mathbf{E}}$ for Éloise is *cardinality good* if for all strategy $\varphi_{\mathbf{A}}$ of Abelard the tree $\mathrm{Outcomes}^{\varphi_{\mathbf{E}},\varphi_{\mathbf{A}}}$ contains at most countably many rejecting branches (i.e., not in

Ω).

A strategy $\varphi_{\mathbf{E}}$ for Éloise is *topologically good* if for all strategy $\varphi_{\mathbf{A}}$ of Abelard the the set of rejecting branches is meager in the tree $\text{Outcomes}^{\varphi_{\mathbf{E}},\varphi_{\mathbf{A}}}$.

In the Büchi game presented above, Éloise does not have a winning strategy. However the strategy φ_{E} consisting in always moving to the node controlled by Nature is both cardinality good and topological good.

We can easily recover the probabilistic semantic by defining a strategy $\varphi_{\mathbf{E}}$ to be almost-surely winning if for all strategy $\varphi_{\mathbf{A}}$ of Abelard the set of accepting branches in the tree $\text{Outcomes}^{\varphi_{\mathbf{E}},\varphi_{\mathbf{A}}}$ has measure 1 (or equivalently the set of rejecting branches has measure 0).

Our contribution in [41], is to transform games with Nature equipped with the cardinality or topological semantic into equivalent games without Nature. These transformations do not assume that the arena is finite and only assume that the objective of the game is Borel. We also obtain similar transformations for games with imperfect informations but with more restrictions. The scope of our results is summarised in the table below.

	Cardinality good ?	Topologically good?
Perfect-information		
	No extra hypothesis on $\mathbb{G}$	Eve + Nature only
	No extra hypothesis on Ω	No extra hypothesis on Ω
Imperfect-information		
	Adam perfect	Eve + Nature only
	Ω: parity	Ω: parity

Let us conclude by briefly mentioning some consequences of this work. In the case of the topological semantic, our approach subsumes the approach of Asarin et al. in [6]. In fact their approach corresponds to games played between Éloise and Nature in our context but we are not limited to finite arena or parity conditions.

One of the main consequences of this work is can use the cardinality or topological semantic in situations where the probabilistic semantic is undecidable. For example, we can decide if Éloise wins a one-player collapsible parity game with the topological semantic but this problem is undedicable for the probabilistic semantic. Using the decidability of MSO on the unfolding of these games, we can even decide the winner in a two-player collapsible parity game with the topological semantics, thanks to a remark of Pawel Parys. However at the moment, we do not have a game reduction in this setting.

4.3 Tree automata with equality between siblings

In this section, we consider the extension of parity tree automata with the ability to test equality (or disequality) between the direct sub-trees of a node. Roughly speaking we consider parity tree automata where some transitions are guarded and can be used only when the two direct sub-trees of the current node are equal/disequal.

This work generalizes to infinite trees the work by Bogaert and Tison [16] who considered a model of automata on *finite ranked* trees where one can check equality and disequality constraints between direct subtrees: they proved that this class of

automata is closed under Boolean operations and that both the emptiness and the finiteness problem of the accepted language are decidable.

For infinite trees, we showed in [49] that the resulting class of languages encompasses the one of regular languages of infinite trees while sharing most of its closure properties, in particular it is a Boolean algebra. Our main technical contribution was then to prove that it also enjoys a decidable emptiness problem. In fact, we proved a stronger decidability result and showed that we can compute the cardinality of the accepted language.

The theory of tree automata with equality and disequality constraints on finite trees has been developed during the last two decades. Over this period, the decidability results have been pushed to stronger and stronger models. As one remarkable result, the theory of these automata has provided tools for solving a long standing open question, namely the decidability of the "HOM problem" [84, 83], which asks for a given regular language T of finite trees and a tree homomorphism h, whether the image $h(T)$ of T under h is a regular tree language.

In the case of infinite trees, our main motivation was to enrich the properties that can be modeled by tree automata while remaining algorithmically tractable. As remarked in [106], our result implies the decidability of MSO enriched with a predicate expressing the equality between the two subtrees of a given node. In [106], Landwehr and Löding showed the closure under projection for the Büchi acceptance condition. In [107], they studied parity tree automata with global constraints as opposed to the local ones considered in this section.

4.3.1 Definition and basic properties

With any A-labelled tree t we associate a unique $A \times \{=, \neq\}$-labelled tree denoted $t^{\stackrel{?}{=}}$ obtained by annotating every node u in t by an extra information regarding on whether the left and the right subtrees of u are equal or not. More formally, for every $u \in \{0,1\}^*$, we have:

$$t^{\stackrel{?}{=}}(u) = \begin{cases} (t(u), =) & \text{if } t_{u0} = t_{u1} \\ (t(u), \neq) & \text{if } t_{u0} \neq t_{u1} \end{cases}$$

A *parity tree automaton with (equality and disequality) constraints* over an alphabet A is a parity tree automaton $\mathcal{A}$ over the alphabet $A \times \{=, \neq\}$. Hence, viewed as a standard tree automaton, it recognises a language of $A \times \{=, \neq\}$-labelled trees. However, it will mainly be used to define languages of A-labelled trees: for that we define

$$L^{con}(\mathcal{A}) = \{t \mid t^{\stackrel{?}{=}} \in L(\mathcal{A})\}.$$

An alternative way of thinking of an automaton with constraints processing an A-labelled tree is by considering it as using guards: a transition $(q, (a, \iota), q_0, q_1)$ can only be fired in a node labelled by a where both subtrees are equal (resp. different) in case ι is $=$ (resp. $\neq$).

In the following, we will refer to $L^{con}(\mathcal{A})$ as the *language recognised* by $\mathcal{A}$. Sometimes we explicitly refer to $L^{con}(\mathcal{A})$ as the *constrained language* of $\mathcal{A}$ to stress that it satisfies the constraints from the transitions. We denote by $\mathbf{REG}^{\stackrel{?}{=}}$ the class of languages recognised by automata with equality and disequality constraints. As the regular tree languages form an effective Boolean algebra, we immediatly have that:

Proposition 4.15
The class $\mathbf{REG}^{\stackrel{?}{=}}$ is an effective Boolean algebra.

Obviously, every standard tree automaton can easily be turned into an equivalent tree automaton with constraints (for instance by introducing two transitions, one with each guard, for each transition). Furthermore, it is clear that the language of $\{a, b\}$-labelled trees $L = \{t \mid t_0 = t_1\}$ can be recognised by a tree automaton with constraints but not by a standard tree automaton.

Proposition 4.16
The class of regular languages of infinite trees is strictly included in $\mathbf{REG}^{\stackrel{?}{=}}$.

4.3.2 Decidability of the cardinality problem

In this section, we are interested in computing the cardinality of languages of the form $L^{con}(\mathcal{A})$. Independently of the Continuum Hypothesis, we proved that the cardinality of a constrained language $L^{con}(\mathcal{A})$ is either finite, $\aleph_0$ (the cardinal of the natural numbers), or $2^{\aleph_0}$ (the cardinal of the reals). A similar statement is proved for regular languages of infinite trees in [115].

Problem 4.17 (*Cardinality Problem*)
The cardinality problem asks, for a given automaton with equality and disequality constraints $\mathcal{A}$ to compute the cardinality of $L^{con}(\mathcal{A})$.

Obviously, the decidability of the cardinality problem implies the decidability of the emptiness problem. Furthermore, it generalises the finiteness problem, which is to decide for a given automaton whether its language is finite.

In [115], Niwiński proved that the cardinality problem is decidable for regular languages of infinite trees. In addition, he gave effective characterisations for countable and uncountable regular languages of infinite trees which will play a central role in our proof.

First we showed that we can consider special cases of automata, *automata with disequality everywhere*, that correspond to automata with a transition relation $\Delta \subseteq Q \times (A \times \{\neq\}) \times Q \times Q$. Obviously, these automata are strictly less expressive than the full class of automata with equality and disequality constraints. However, Theorem 4.18 below shows that one can remove equality constraints and even require disequality constraints in all transitions without changing the cardinality of the accepted language.

Theorem 4.18
Let $\mathcal{A}$ be a parity automaton with equality and disequality constraints. Then one can build an automaton $\mathcal{B}$ with disequality everywhere (over an alphabet with two new symbols) that is such that $L^{con}(\mathcal{A})$ and $L^{con}(\mathcal{B})$ have the same cardinality. If $\mathcal{A}$ is a safety (resp. Büchi) automaton, then so is $\mathcal{B}$.

For parity tree automata with disequality everywhere, we are able to prove the decidability of the cardinality problem. Combined with Theorem 4.18, we obtained the desired result.

Theorem 4.19
The cardinality problem for parity tree automata with with equality and disequality

constraints is decidable.

The proof of this theorem is extremely involved. We will try to present a flavor of it in the case of the Büchi acceptance condition.

The first idea of the proof is to consider the unconstrained version $\widehat{\mathcal{A}}$ of a tree automaton with constraints $\mathcal{A}$. Intuitively $\widehat{\mathcal{A}}$ mimics $\mathcal{A}$ but ignores the guards. More formally, starting from $\mathcal{A} = (Q, A \times \{=, \neq\}, q_0, \Delta, \mathrm{Col})$ we let $\widehat{\mathcal{A}} = (Q, A, q_0, \Delta', \mathrm{Col})$ where $\Delta' = \{(q, a, q_0, q_1) \mid (q, (a, =), q_0, q_1) \in \Delta \text{ or } (q, (a, \neq), q_0, q_1) \in \Delta\}$. We refer to $L(\widehat{\mathcal{A}})$ as the *unconstrained language* of $\mathcal{A}$.

For any automaton with equality and disequality constraints $\mathcal{A}$ one has $L^{con}(\mathcal{A}) \subseteq L(\widehat{\mathcal{A}})$.

A first result of the proof is that if the unconstrained language of $\mathcal{A}$ is countable, we can reduce the cardinality problem to the finiteness problem for finite tree automata with equality and disequality which is shown to be decidable in [16]. Essentially this is due to the very special form of countable regular languages of infinite trees established in [115].

Theorem 4.20
Let $\mathcal{A}$ be a parity tree automaton with equality and disequality constraints on infinite trees. If $L(\widehat{\mathcal{A}})$ is countable, then one can compute the cardinality of $L^{con}(\mathcal{A})$, and in case it is finite $L^{con}(\mathcal{A})$ consists only of regular trees and can be effectively computed.

We now turn to the solution of the cardinality problem for Büchi acceptance condition. Our aim is to compute the *cardinality profile of* $\mathcal{A}$, denoted by $\kappa_{\mathcal{A}}$, which is a mapping $\kappa_{\mathcal{A}} : Q \to \mathbb{N} \cup \{\aleph_0, 2^{\aleph_0}\}$ associating to every state q of $\mathcal{A}$ the cardinality of the constrained language accepted by $\mathcal{A}$ from q, denoted by $L^{con}(\mathcal{A}_q)$.

Our algorithm works with a slightly extended model of automaton which can directly check, in some node u, that $t[u]$ equals some regular tree. This will not add any expressive power to our model (both the constrained and the unconstrained one) but it greatly simplifies the presentation. An *extended tree automata* can propagate a regular tree instead of a state in its transitions – the intended meaning being to only accept this particular regular tree.

We now give two constructions used in our algorithm. Each of them takes an (extended) automaton with disequality everywhere $\mathcal{A}$ and produces another automaton $\mathcal{B}$ that is such that (when used in the right context) $L^{con}(\mathcal{A}) = L^{con}(\mathcal{B})$; and $L(\widehat{\mathcal{B}}) \subseteq L(\widehat{\mathcal{A}})$.

Let $\mathcal{A} = (Q, A, q_0, \Delta, \mathrm{Col})$ be an extended automaton with disequality everywhere. Let $q \in Q$ be a state such that $L^{con}(\mathcal{A}_q) = \varnothing$. Then we define a new automaton $\mathcal{A}_{q \mapsto \varnothing} = (Q \setminus \{q\}, A, q_0, \Delta', \mathrm{Col})$ where Δ' is obtained from Δ by only keeping transitions that do not involve q, i.e., $\Delta' = \{(p, (a, \neq), p_0, p_1) \in \Delta \mid p, p_0, p_1 \neq q\}$.

Let $\mathcal{A} = (Q, A, q_0, \Delta, \mathrm{Col})$ be an extended automaton with disequality everywhere. Let $q \in Q$ be a state such that $L^{con}(\mathcal{A}_q) = \{t_1, \ldots, t_n\}$ is a finite set of regular trees. Then we define a new automaton $\mathcal{A}_{q \mapsto t_1, \ldots, t_n} = (Q \setminus \{q\}, A, q_0, \Delta', \mathrm{Col})$ where Δ' is obtained from Δ by replacing every transition of the form $(p, (a, \neq), q_0, q_1)$ with q_0 and/or q_1 being equal to q by all the transitions obtained by substituting occurrences of q with elements of $\{t_1, \ldots, t_n\}$, where in case $q_0 = q_1 = q$ the trees substituted for the two occurrences of q have to be different.

Our algorithm (Algorithm 1) takes as input a Büchi automaton with disequality constraints everywhere $\mathcal{A} = (Q, A, q_0, \Delta, \mathrm{Acc})$. The algorithm identifies states whose

Algorithm 1 Solve the cardinality problem for safety/Büchi automata

Input:

Tree automaton with disequality constraints everywhere $\mathcal{A} = (Q, A, q_0, \Delta, \mathrm{Acc})$

Data Structure:

Set $S \leftarrow Q$
Automaton $\mathcal{B} \leftarrow \mathcal{A}$
Function $\kappa : Q \to \mathbb{N} \cup \{\aleph_0, 2^{\aleph_0}\}$; $\kappa(q) \leftarrow 2^{\aleph_0}$ for all q

Code:

1: **while** $\exists q \in S$ s.t. $|L(\widehat{\mathcal{B}_q})| \leqslant \aleph_0$ **do**
2: $\quad \kappa(q) \leftarrow |L^{con}(\mathcal{B}_q)|$
3: $\quad$ **if** $\kappa(q) = 0$ **then**
4: $\quad\quad \mathcal{B} \leftarrow \mathcal{B}_{q \mapsto \varnothing}$
5: $\quad$ **else if** $\kappa(q) < \aleph_0$ **then**
6: $\quad\quad$ Let $L^{con}(\mathcal{B}_q) = \{t_1, \ldots, t_n\}$
7: $\quad\quad \mathcal{B} \leftarrow \mathcal{B}_{q \mapsto t_1, \ldots, t_n}$
8: $\quad$ **end if**
9: $\quad S \leftarrow S \setminus \{q\}$
10: **end while**
11: **return** κ

unconstrained language is countable (which is decidable according to [115]), and then determines the cardinality of the constrained language (Theorem 4.20). States for which the constrained language is finite are substituted by the regular trees in this language (which can be computed according to Theorem 4.20) using the operation $\mathcal{A}_{q \mapsto t_1, \ldots, t_n}$, and states with empty constrained language are eliminated using the operation $\mathcal{A}_{q \mapsto \varnothing}$. Note that these states remain in the state set because we want to keep the set fixed. However, they become unreachable by these operations.

The modifications do not change the constrained language accepted by the automaton. However, it might change the unconstrained languages. In particular, an unconstrained language that was uncountable may become countable by such a modification. The algorithm iterates this process until no new states with countable unconstrained language are found, and returns a cardinality profile κ. We prove that if the acceptance condition in $\mathcal{A}$ is a Büchi condition, then κ is the cardinality profile of $\mathcal{A}$. This proof basically amounts to showing that the states for which the unconstrained language is uncountable upon termination of the algorithm, also have an uncountable constrained language. The proof is a technical construction showing how to build uncountably many trees in the constrained language of each state q such that $\kappa(q) = 2^{\aleph_0}$. It is based on a result of Niwinski [115] that characterises uncountable ω-regular languages. This construction only works for the Büchi condition. Indeed this algorithm is incorrect when applied to a co-Büchi tree automaton. In fact, our proof in the co-Büchi case is quite different. The proof for the parity case combines the ideas of both the Büchi and co-Büchi case.

5
Perspectives

We now present some questions and problems that were opened by the work presented in this manuscript. Of course, our research plan is not limited to these questions.

Efficient synthesis algorithms for recursion schemes

As we have seen in Section 3.3, they are efficient algorithms for model-checking reachability or safety properties on recursion schemes. A first natural task is to generalize the saturation based algorithm we introduced in [23] to solve *parity* collapsible pushdown games. At order 1, Hague and Ong have generalized the standard saturation method from reachability pushdown games to parity pushdown games [94]. We believe the same general methodology can be extended to higher order parity games.

A more challenging problem is to find efficient algorithms for the effective selection problem for the modal μ-calculus. We have seen in Section 3.3.4, that the effective selection problem for the μ-calculus at its core reduces to constructing winning strategies in parity collapsible pushdown games. If we want to device algorithms that work well in practice, we need some form of incremental construction and not algorithms that start by construction objects of exponential (or towers of exponential) sizes. Following this line of reasoning the question is whether it is possible to find saturation-like algorithms for the synthesis of winning strategies in collapsible pushdown games.

Already at order 1 (i.e., for simple pushdown games), the question is already non-trivial. The saturation method for computing the winning region in a reachability pushdown game naturally defines a strategy implemented by a pushdown automata. However we would ideally like to synthesis positional strategies where the move only depends on the current configuration. It is known that for pushdown games, positional strategies can be defined using deterministic finite automata reading the configuration (from the bottom of the stack) and outputting the move to play. These strategies are called regular strategies. It follows from [139] that a winning regular

strategy can be computed in EXPTIME. However this method starts by building an object of exponential size and is hence unlikely to be useful in practice.

We have been studying it with Didier Caucal and Olivier Serre for quite a while now. Our initial goal was to give a saturation algorithm for computing positional winning strategies in parity pushdown games. Sadly we did not fully succeed yet but we have obtained some results along the road.

A first natural question is which positional strategy to synthesise. For reachability games, the most natural candidate is the optimal strategy: the strategy that reaches the target set in the smallest amount of steps regardless of the opponent's moves. With Olivier Serre, we showed in [56] that in general enriching a pushdown game with the optimal strategy may result in a graph with an undecidable MSO-theory. In particular the resulting graph is no longer a pushdown graph. This give a strong indication that the optimal strategy is not the one we should try to implement.

In [31], Cachat gave an algorithm to compute the distance of a configuration to the target set based on the saturation method. Cachat algorithm enriches the standard saturation algorithm to assign to each transition of the saturated automaton a weight in $\mathbb{N}$ and defines the distance of a configuration as the minimal weight of an accepting run. With Matthew Hague, we showed in [44] that this algorithm only computes an upper-bound on the distance. We corrected the algorithm by assigning more precise weights in $\mathbb{N}^d$.

Still with Matthew Hague, we showed that a regular positional strategy can be computed from the saturation algorithm for reachability pushdown games. The finite deterministic finite automata characterizing the positional strategies are constructed using the informations collected during the saturation algorithm. The complexity matches that of [139]. However this approach suffers from the same shortcomings as we end up building an exponential object in all cases.

In an unpublished work with Didier Caucal and Olivier Serre, we give a saturation algorithm for computing positional winning strategies in reachability pushdown games. We work with graph grammars instead of pushdown systems. Graph grammars [70] are deterministic graph rewriting systems that (under some restrictions) are equivalent to pushdown systems (see [61] for a survey of this notion). The algorithm takes as input a graph grammar generating the pushdown game and saturates it with edges marking a winning strategy for the first player. The complexity of the resulting algorithm applied to a graph grammar G of arity ρ is in $\mathcal{O}(2^{p(\rho)}q(|G|))$ for some polynomials p and q. In particular, our algorithm is fixed parameter tractable. The algorithm is quite simple to implement most of the difficulty is in its correctness proof.

A non-trivial task is to extend this approach to parity pushdown games and to higher-order pushdown games.

Structural characterization of recursion schemes

One of the goal of our work was to demonstrate the power of the approach by graph transformation *à la Caucal* to study safe recursion schemes. It is possible to extend it to terms generated by unsafe recursion schemes [64]. Of course, we cannot hope to have a characterization as nice as for the safe recursion schemes. For instance, it is known that collapsible pushdown graph have an undecidable MSO-theory already at order 2 [92]. This approach remains unpublished and therefore relatively unexplored.

It might be possible to use it to derive decidability result for the MSO logic using the recent advances on models for the λY-calculus as some of the challenges are the similar. But we also think that this approach could be instrumental in understanding the structures defined by unsafe recursion schemes. As a test case, it would be interesting to obtain a characterization of the ordinal definable by unsafe recursion schemes.

Structures defined by recursion schemes

In Section 3.5, we gave some results on families of structures defined by safe recursion schemes. This work is only preliminary and the expressivity of recursion schemes when describing infinite structures is not well-understood.

It would be for instance interesting to characterize the functions f such that Comb_f is representable by a scheme of order k. A similar family of functions (defined by safe schemes) was studied by Fratani and Sénizergues [80] for which they established numerous closure properties. The class in the unsafe case should enjoy similar closure properties.

In Section 3.5.1, we characterized the ordinals defined by safe recursion schemes. Unsafe recursion schemes are more expressive than safe ones for generating terms [123]. We conjecture however that they have the same expressive power when it comes to generating ordinals. A possible way to prove this result could be via a typing approach similar to the one developed by Parys for deciding the finiteness problem in [122]. Proving this conjecture would also provide another proof of the strictness of the hierarchy of terms generated by unsafe recursion schemes.

If this conjecture is true, only the ordinal below ε_0 are definable by recursion schemes. It is natural to wonder if there exists a tree with a decidable MSO-theory whose frontier is ε_0. This problem is more difficult than it seems. However it would be a good starting point to extend the hierarchy of recursion schemes.

Revisiting notions of regularity

Three notions of regularity have been defined for set of higher-order stacks and stacks with links : two of them where introduced in [45, 26] and are presented in this document in Section 3.3.3 and the other one was introduced in [33] and independently in [82]. Surprisingly these three notions share many properties: they are all Boolean algebras, they capture the winning regions of parity games on the relevant model of higher-order pushdown automata, ...

We believe that all these results can be uniformly using the existence of finitary models of the λY-calculus for MSO. In some sense, the notion of regularity can be derived from the recursion schemes defining the corresponding model of higher-order pushdown automaton.

A more ambitious task would be to unify the different finitary models introduced for MSO on the λY-calculus.

Games with Nature

A natural question regarding games with Nature presented in Section 4.2.3 is to generalize our approach to games with Éloise, Abelard and Nature in the topological

setting.

Another line of research is to consider games with Nature in their full form. Recall that, in a game with Nature, fixing a strategy for both Éloise and Abelard does no longer result in a unique infinite play but rather in a tree of plays. It is natural to consider the winning objective to be a regular set of infinite trees rather than an ω-regular set of infinite words. As remarked to us by Pawel Parys, if the game with Nature has a decidable MSO-theory, we can decide if Éloise has winning strategy. However these general games are largely unexplored. In particular, we do not know for which objectives these games are determined or if these general objectives could be used to model interesting properties.

Bibliography

[1] Samson Abramsky. "A Domain Equation for Bisimulation". In: *Inf. Comput.* 92.2 (1991), pp. 161–218. DOI: 10.1006/inco.1991.9999.

[2] Luca Aceto, Arnaud Carayol, Zoltán Ésik, and Anna Ingólfsdóttir. "Algebraic Synchronization Trees and Processes". In: *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*. Ed. by Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer. Vol. 7392. Lecture Notes in Computer Science. Springer, 2012, pp. 30–41. DOI: 10.1007/978-3-642-31585-5_7.

[3] Klaus Aehlig. "A Finite Semantics of Simply-Typed Lambda Terms for Infinite Runs of Automata". In: *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*. Ed. by Zoltán Ésik. Vol. 4207. Lecture Notes in Computer Science. Springer, 2006, pp. 104–118. DOI: 10.1007/11874683_7.

[4] Klaus Aehlig, Jolie G. de Miranda, and C.-H. Luke Ong. "The Monadic Second Order Theory of Trees Given by Arbitrary Level-Two Recursion Schemes Is Decidable". In: *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*. Ed. by Pawel Urzyczyn. Vol. 3461. Lecture Notes in Computer Science. Springer, 2005, pp. 39–54. DOI: 10.1007/11417170_5.

[5] Alfred V. Aho. "Indexed Grammars - An Extension of Context-Free Grammars". In: *J. ACM* 15.4 (1968), pp. 647–671. DOI: 10.1145/321479.321488.

[6] Eugene Asarin, Raphaël Chane-Yack-Fa, and Daniele Varacca. "Fair Adversaries and Randomization in Two-Player Games". In: *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by C.-H. Luke Ong. Vol. 6014. Lecture Notes in Computer Science. Springer, 2010, pp. 64–78. DOI: 10.1007/978-3-642-12032-9_6.

[7] Jos C. M. Baeten, Twan Basten, and Michel A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Vol. 50. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2009. DOI: 10.1017/CBO9781139195003.

[8] Christel Baier, Nathalie Bertrand, and Marcus Größer. "On Decision Problems for Probabilistic Büchi Automata". In: *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*. Ed. by Roberto M. Amadio. Vol. 4962. Lecture Notes in Computer Science. Springer, 2008, pp. 287–301. DOI: 10.1007/978-3-540-78499-9_21.

[9] Christel Baier, Marcus Größer, and Nathalie Bertrand. "Probabilistic ω-automata". In: *J. ACM* 59.1 (2012), 1:1–1:52. DOI: 10.1145/2108242.2108243.

[10] Danièle Beauquier, Maurice Nivat, and Damian Niwinski. "About the Effect of the Number of Successful Paths in an Infinite Tree on the Recognizability by a Finite Automaton with Büchi Conditions". In: *Fundamentals of Computation Theory, 8th International Symposium, FCT '91, Gosen, Germany, September 9-13, 1991, Proceedings*. Ed. by Lothar Budach. Vol. 529. Lecture Notes in Computer Science. Springer, 1991, pp. 136–145. DOI: 10.1007/3-540-54458-5_58.

[11] Danièle Beauquier and Damian Niwinski. "Automata on Infinite Trees with Counting Constraints". In: *TAPSOFT'93: Theory and Practice of Software Development, International Joint Conference CAAP/FASE, Orsay, France, April 13-17, 1993, Proceedings*. Ed. by Marie-Claude Gaudel and Jean-Pierre Jouannaud. Vol. 668. Lecture Notes in Computer Science. Springer, 1993, pp. 266–281. DOI: 10.1007/3-540-56610-4_70.

[12] Stephen L. Bloom and Zoltán Ésik. "Algebraic Ordinals". In: *Fundam. Inform.* 99.4 (2010), pp. 383–407. DOI: 10.3233/FI-2010-255.

[13] Stephen L. Bloom and Zoltán Ésik. "Regular and Algebraic Words and Ordinals". In: *Algebra and Coalgebra in Computer Science, Second International Conference, CALCO 2007, Bergen, Norway, August 20-24, 2007, Proceedings*. Ed. by Till Mossakowski, Ugo Montanari, and Magne Haveraaen. Vol. 4624. Lecture Notes in Computer Science. Springer, 2007, pp. 1–15. DOI: 10.1007/978-3-540-73859-6_1.

[14] Stephen L. Bloom and Zoltán Ésik. "Scattered Algebraic Linear Orderings". In: *6th Workshop on Fixed Points in Computer Science, FICS 2009, Coimbra, Portugal, September 12-13, 2009*. Ed. by Ralph Matthes and Tarmo Uustalu. Institute of Cybernetics, 2009, pp. 25–29.

[15] Achim Blumensath and Erich Grädel. "Automatic Structures". In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 2000, pp. 51–62. DOI: 10.1109/LICS.2000.855755.

[16] Bruno Bogaert and Sophie Tison. "Equality and Disequality Constraints on Direct Subterms in Tree Automata". In: *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*. Ed. by Alain Finkel and Matthias Jantzen. Vol. 577. Lecture Notes in Computer Science. Springer, 1992, pp. 161–171. DOI: 10.1007/3-540-55210-3_181.

[17] Mikolaj Boja'nczyk, Hugo Gimbert, and Edon Kelmendi. "Emptiness of Zero Automata Is Decidable". In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*. Ed. by Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 106:1–106:13. DOI: 10.4230/LIPIcs.ICALP.2017.106.

[18] Ahmed Bouajjani, Javier Esparza, and Oded Maler. "Reachability Analysis of Pushdown Automata: Application to Model-Checking". In: *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*. Ed. by Antoni W. Mazurkiewicz and Józef Winkowski. Vol. 1243. Lecture Notes in Computer Science. Springer, 1997, pp. 135–150. DOI: 10.1007/3-540-63141-0_10.

[19] Julian C. Bradfield and Igor Walukiewicz. "The mu-calculus and Model Checking". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 871–919. DOI: 10.1007/978-3-319-10575-8_26.

[20] Laurent Braud. "Covering of ordinals". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009, December 15-17, 2009, IIT Kanpur, India*. Ed. by Ravi Kannan and K. Narayan Kumar. Vol. 4. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009, pp. 97–108. DOI: 10.4230/LIPIcs.FSTTCS.2009.2310.

[21] Laurent Braud and Arnaud Carayol. "Linear Orders in the Pushdown Hierarchy". In: *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*. Ed. by Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis. Vol. 6199. Lecture Notes in Computer Science. Springer, 2010, pp. 88–99. DOI: 10.1007/978-3-642-14162-1_8.

[22] Christopher H. Broadbent. "On collapsible pushdown automata, their graphs and the power of links". PhD thesis. University of Oxford, UK, 2011.

[23] Christopher H. Broadbent, Arnaud Carayol, Matthew Hague, and Olivier Serre. "A Saturation Method for Collapsible Pushdown Systems". In: *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*. Ed. by Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer. Vol. 7392. Lecture Notes in Computer Science. Springer, 2012, pp. 165–176. DOI: 10.1007/978-3-642-31585-5_18.

[24] Christopher H. Broadbent, Arnaud Carayol, Matthew Hague, and Olivier Serre. "C-SHORe: A collapsible approach to higher-order verification". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 13–24. DOI: 10.1145/2500365.2500589.

[25] Christopher H. Broadbent, Arnaud Carayol, Matthew Hague, and Olivier Serre. "C-SHORe: Higher-Order Verification via Collapsible Pushdown System Saturation". In: *CoRR* abs/1703.04429 (2017).

[26] Christopher H. Broadbent, Arnaud Carayol, C.-H. Luke Ong, and Olivier Serre. "Recursion Schemes and Logical Reflection". In: *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*. IEEE Computer Society, 2010, pp. 120–129. DOI: 10.1109/LICS.2010.40.

[27] Christopher H. Broadbent and Naoki Kobayashi. "Saturation-Based Model Checking of Higher-Order Recursion Schemes". In: *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*. Ed. by Simona Ronchi Della Rocca. Vol. 23. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013, pp. 129–148. DOI: 10.4230/LIPIcs.CSL.2013.129.

[28] J. Richard Büchi. "On a decision method in restricted second order arithmetic". In: *Proceedings of the 1960 International Congress of Logic, Methodology and Philosophy of Science*. Ed. by E. Nagel, P. Suppes, and A. Tarski. Stanford University Press, 1962, pp. 1–11.

[29] J. Richard Büchi. "Using Determinancy of Games to Eliminate Quantifiers". In: *Fundamentals of Computation Theory, Proceedings of the 1977 International FCT-Conference, Poznan-Kórnik, Poland, September 19-23, 1977*. Ed. by Marek Karpinski. Vol. 56. Lecture Notes in Computer Science. Springer, 1977, pp. 367–378. DOI: 10.1007/3-540-08442-8_104.

[30] J. Richard Büchi and Dirk Siefkes. "The Monadic Second Order Theorie of All Countable Ordinals". In: *Decidable Theories II*. Ed. by G. H. Muller and Dirk Siefkes. Vol. 328. Springer-Verlag, 1973, pp. 1–217.

[31] Thierry Cachat. "Higher Order Pushdown Automata, the Caucal Hierarchy of Graphs and Parity Games". In: *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*. Ed. by Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger. Vol. 2719. Lecture Notes in Computer Science. Springer, 2003, pp. 556–569. DOI: 10.1007/3-540-45061-0_45.

[32] Arnaud Carayol. "Automates infinis, logiques et langages". PhD thesis. University of Rennes 1, France, 2006.

[33] Arnaud Carayol. "Regular Sets of Higher-Order Pushdown Stacks". In: *Mathematical Foundations of Computer Science 2005, 30th International Symposium, MFCS 2005, Gdansk, Poland, August 29 - September 2, 2005, Proceedings*. Ed. by Joanna Jedrzejowicz and Andrzej Szepietowski. Vol. 3618. Lecture Notes in Computer Science. Springer, 2005, pp. 168–179. DOI: 10.1007/11549345_16.

[34] Arnaud Carayol and Thomas Colcombet. "On Equivalent Representations of Infinite Structures". In: *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*. Ed. by Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger. Vol. 2719. Lecture Notes in Computer Science. Springer, 2003, pp. 599–610. DOI: 10.1007/3-540-45061-0_48.

[35] Arnaud Carayol and Zoltán Ésik. "A Context-Free Linear Ordering with an Undecidable First-Order Theory". In: *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings*. Ed. by Jos C. M. Baeten, Thomas Ball, and

Frank S. de Boer. Vol. 7604. Lecture Notes in Computer Science. Springer, 2012, pp. 104–118. DOI: 10.1007/978-3-642-33475-7_8.

[36] Arnaud Carayol and Zoltán Ésik. "An analysis of the equational properties of the well-founded fixed point". In: *J. Log. Algebr. Meth. Program.* 86.1 (2017), pp. 308–318. DOI: 10.1016/j.jlamp.2016.09.004.

[37] Arnaud Carayol and Zoltán Ésik. "An Analysis of the Equational Properties of the Well-Founded Fixed Point (short paper)". In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*. Ed. by Chitta Baral, James P. Delgrande, and Frank Wolter. AAAI Press, 2016, pp. 533–536.

[38] Arnaud Carayol and Zoltán Ésik. "The FC-rank of a context-free language". In: *Inf. Process. Lett.* 113.8 (2013), pp. 285–287. DOI: 10.1016/j.ipl.2013.01.005.

[39] Arnaud Carayol and Stefan Göller. "On Long Words Avoiding Zimin Patterns". In: *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*. Ed. by Heribert Vollmer and Brigitte Vallée. Vol. 66. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 19:1–19:13. DOI: 10.4230/LIPIcs.STACS.2017.19.

[40] Arnaud Carayol and Stefan Göller. "On Long Words Avoiding Zimin Patterns". In: *Theory Comput. Syst.* 63.5 (2019), pp. 926–955. DOI: 10.1007/S00224-019-09914-2.

[41] Arnaud Carayol, Axel Haddad, and Olivier Serre. "Erratum for "Randomization in Automata on Infinite Trees"". In: *ACM Trans. Comput. Log.* 16.4 (2015), 36:1–36:2. DOI: 10.1145/2824254.

[42] Arnaud Carayol, Axel Haddad, and Olivier Serre. "Qualitative Tree Languages". In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 2011, pp. 13–22. DOI: 10.1109/LICS.2011.28.

[43] Arnaud Carayol, Axel Haddad, and Olivier Serre. "Randomization in Automata on Infinite Trees". In: *ACM Trans. Comput. Log.* 15.3 (2014), 24:1–24:33. DOI: 10.1145/2629336.

[44] Arnaud Carayol and Matthew Hague. "Optimal Strategies in Pushdown Reachability Games". In: *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*. Ed. by Igor Potapov, Paul G. Spirakis, and James Worrell. Vol. 117. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 42:1–42:14. DOI: 10.4230/LIPICS.MFCS.2018.42.

[45] Arnaud Carayol, Matthew Hague, Antoine Meyer, C.-H. Luke Ong, and Olivier Serre. "Winning Regions of Higher-Order Pushdown Games". In: *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 2008, pp. 193–204. DOI: 10.1109/LICS.2008.41.

[46] Arnaud Carayol and Christof Löding. "MSO on the Infinite Binary Tree: Choice and Order". In: *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*. Ed. by Jacques Duparc and Thomas A. Henzinger. Vol. 4646. Lecture Notes in Computer Science. Springer, 2007, pp. 161–176. DOI: 10.1007/978-3-540-74915-8_15.

[47] Arnaud Carayol and Christof Löding. "Uniformization in Automata Theory". In: *Logic, Methodology and Philosophy of Science - Logic and Science Facing the New Technologies - Proceedings of the 14th International Congress (Nancy)* (2015). Ed. by Peter Schroeder-Heister, Gerhard Heinzmann, Wilfrid Hodges, and Pierre Edouard Bour, pp. 1–26.

[48] Arnaud Carayol, Christof Löding, Damian Niwinski, and Igor Walukiewicz. "Choice Functions and Well-orderings over the Infinite Binary Tree". In: *Open Mathematics* 8.4 (Jan. 2010), pp. 662–682. DOI: 10.2478/s11533-010-0046-z.

[49] Arnaud Carayol, Christof Löding, and Olivier Serre. "Automata on Infinite Trees with Equality and Disequality Constraints Between Siblings". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2016, New York, NY, USA, July 5-8, 2016*. Ed. by Martin Grohe, Eric Koskinen, and Natarajan Shankar. ACM, 2016, pp. 227–236. DOI: 10.1145/2933575.2934504.

[50] Arnaud Carayol, Christof Löding, and Olivier Serre. "Pure Strategies in Imperfect Information Stochastic Games". In: *Fundam. Inform.* 160.4 (2018), pp. 361–384. DOI: 10.3233/FI-2018-1687.

[51] Arnaud Carayol and Cyril Nicaud. "Distribution of the number of accessible states in a random deterministic automaton". In: *29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th - March 3rd, 2012, Paris, France*. Ed. by Christoph Dürr and Thomas Wilke. Vol. 14. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012, pp. 194–205. DOI: 10.4230/LIPIcs.STACS.2012.194.

[52] Arnaud Carayol and Olivier Serre. "Collapsible Pushdown Automata and Labeled Recursion Schemes: Equivalence, Safety and Effective Selection". In: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 2012, pp. 165–174. DOI: 10.1109/LICS.2012.73.

[53] Arnaud Carayol and Olivier Serre. "Counting branches in trees using games". In: *Inf. Comput.* 252 (2017), pp. 221–242. DOI: 10.1016/j.ic.2016.11.005.

[54] Arnaud Carayol and Olivier Serre. "Higher-order recursion schemes and their automata models". In: Handbook of Automata. J.-E. Pin and W. Thomas (Eds.), 2018. Chap. 35, pp. 440–477.

[55] Arnaud Carayol and Olivier Serre. "How Good Is a Strategy in a Game with Nature?" In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. 2015, pp. 609–620. DOI: 10.1109/LICS.2015.62.

[56] Arnaud Carayol and Olivier Serre. "Marking Shortest Paths On Pushdown Graphs Does Not Preserve MSO Decidability". In: *CoRR* abs/1510.04000 (2015).

[57] Arnaud Carayol and Olivier Serre. "Marking shortest paths on pushdown graphs does not preserve MSO decidability". In: *Inf. Process. Lett.* 116.10 (2016), pp. 638–643. DOI: 10.1016/j.ipl.2016.04.015.

[58] Arnaud Carayol and Michaela Slaats. "Positional Strategies for Higher-Order Pushdown Parity Games". In: *Mathematical Foundations of Computer Science 2008, 33rd International Symposium, MFCS 2008, Torun, Poland, August 25-29, 2008, Proceedings*. Ed. by Edward Ochmanski and Jerzy Tyszkiewicz. Vol. 5162. Lecture Notes in Computer Science. Springer, 2008, pp. 217–228. DOI: 10.1007/978-3-540-85238-4_17.

[59] Arnaud Carayol and Stefan Wöhrle. "The Caucal Hierarchy of Infinite Graphs in Terms of Logic and Higher-Order Pushdown Automata". In: *FSTTCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference, Mumbai, India, December 15-17, 2003, Proceedings*. Ed. by Paritosh K. Pandya and Jaikumar Radhakrishnan. Vol. 2914. Lecture Notes in Computer Science. Springer, 2003, pp. 112–123. DOI: 10.1007/978-3-540-24597-1_10.

[60] D. Caucal. "Sur des graphes infinis réguliers". Habilitation thesis. Université de Rennes 1, 1998.

[61] Didier Caucal. "Deterministic graph grammars". In: *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*. Ed. by Jörg Flum, Erich Grädel, and Thomas Wilke. Vol. 2. Texts in Logic and Games. Amsterdam University Press, 2008, pp. 169–250.

[62] Didier Caucal. "On Infinite Terms Having a Decidable Monadic Theory". In: *Mathematical Foundations of Computer Science 2002, 27th International Symposium, MFCS 2002, Warsaw, Poland, August 26-30, 2002, Proceedings*. Ed. by Krzysztof Diks and Wojciech Rytter. Vol. 2420. Lecture Notes in Computer Science. Springer, 2002, pp. 165–176. DOI: 10.1007/3-540-45687-2_13.

[63] Didier Caucal. "On Infinite Transition Graphs Having a Decidable Monadic Theory". In: *Automata, Languages and Programming, 23rd International Colloquium, ICALP96, Paderborn, Germany, 8-12 July 1996, Proceedings*. Ed. by Friedhelm Meyer auf der Heide and Burkhard Monien. Vol. 1099. Lecture Notes in Computer Science. Springer, 1996, pp. 194–205. DOI: 10.1007/3-540-61440-0_128.

[64] Didier Caucal. *Private communication*. 2012.

[65] Krishnendu Chatterjee, Marcin Jurdzinski, and Thomas A. Henzinger. "Quantitative stochastic parity games". In: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*. Ed. by J. Ian Munro. SIAM, 2004, pp. 121–130.

[66] Thomas Colcombet. "Unambiguity in Automata Theory". In: *Descriptional Complexity of Formal Systems - 17th International Workshop, DCFS 2015, Waterloo, ON, Canada, June 25-27, 2015. Proceedings*. Ed. by Jeffrey Shallit and Alexander Okhotin. Vol. 9118. Lecture Notes in Computer Science. Springer, 2015, pp. 3–18. DOI: 10.1007/978-3-319-19225-3_1.

[67] David Conlon, Jacob Fox, and Benny Sudakov. "Tower-type bounds for unavoidable patterns in words". In: *Trans. Amer. Math. Soc.* (2019). DOI: 10.1090/tran/7751.

[68] Bruno Courcelle. “A Representation of Trees by Languages I”. In: *Theor. Comput. Sci.* 6 (1978), pp. 255–279. DOI: 10.1016/0304-3975(78)90008-7.

[69] Bruno Courcelle. “A Representation of Trees by Languages II”. In: *Theor. Comput. Sci.* 7 (1978), pp. 25–55. DOI: 10.1016/0304-3975(78)90039-7.

[70] Bruno Courcelle. “Graph Rewriting: An Algebraic and Logic Approach”. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Ed. by Jan van Leeuwen. Elsevier and MIT Press, 1990, pp. 193–242. ISBN: 0-444-88074-7. DOI: 10.1016/b978-0-444-88074-1.50010-x.

[71] Bruno Courcelle. “The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic”. In: *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. 1997, pp. 313–400.

[72] Bruno Courcelle. “The Monadic Second-Order Logic of Graphs, II: Infinite Graphs of Bounded Width”. In: *Mathematical Systems Theory* 21.4 (1989), pp. 187–221. DOI: 10.1007/BF02088013.

[73] Bruno Courcelle and Igor Walukiewicz. “Monadic Second-Order Logic, Graph Coverings and Unfoldings of Transition Systems”. In: *Ann. Pure Appl. Logic* 92.1 (1998), pp. 35–62. DOI: 10.1016/S0168-0072(97)00048-1.

[74] Costas Courcoubetis and Mihalis Yannakakis. “Markov Decision Processes and Regular Events (Extended Abstract)”. In: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings*. Ed. by Mike Paterson. Vol. 443. Lecture Notes in Computer Science. Springer, 1990, pp. 336–349. DOI: 10.1007/BFb0032043.

[75] Werner Damm. “The IO- and OI-Hierarchies”. In: *Theor. Comput. Sci.* 20 (1982), pp. 95–207. DOI: 10.1016/0304-3975(82)90009-3.

[76] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge: Cambridge University Press, 1990. ISBN: 0521365848 9780521365840 0521367662 9780521367660.

[77] E. Allen Emerson and Charanjit S. Jutla. “Tree Automata, Mu-Calculus and Determinacy (Extended Abstract)”. In: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE Computer Society, 1991, pp. 368–377. DOI: 10.1109/SFCS.1991.185392.

[78] Joost Engelfriet. “Iterated Stack Automata and Complexity Classes”. In: *Inf. Comput.* 95.1 (1991), pp. 21–75. DOI: 10.1016/0890-5401(91)90015-T.

[79] Alain Finkel, Bernard Willems, and Pierre Wolper. “A direct symbolic approach to model checking pushdown systems”. In: *Electr. Notes Theor. Comput. Sci.* 9 (1997), pp. 27–37. DOI: 10.1016/S1571-0661(05)80426-8.

[80] S. Fratani and Géraud Sénizergues. “Iterated pushdown automata and sequences of rational numbers”. In: *Ann. Pure Appl. Logic* 141.3 (2006), pp. 363–411. DOI: 10.1016/j.apal.2005.12.004.

[81] Séverine Fratani. “Automates à piles de piles ... de piles”. PhD thesis. Université de Bordeaux 1, 2005.

[82] Séverine Fratani. “Regular sets over extended tree structures”. In: *Theor. Comput. Sci.* 418 (2012), pp. 48–70. DOI: 10.1016/j.tcs.2011.10.020.

[83] Guillem Godoy and Omer Giménez. "The HOM problem is decidable". In: *J. ACM* 60.4 (2013), 23:1–23:44. DOI: 10.1145/2501600.

[84] Guillem Godoy, Omer Giménez, Lander Ramos, and Carme Àlvarez. "The HOM problem is decidable". In: *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*. Ed. by Leonard J. Schulman. ACM, 2010, pp. 485–494. DOI: 10.1145/1806689.1806757.

[85] Erich Grädel. "Banach-Mazur Games on Graphs". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*. Ed. by Ramesh Hariharan, Madhavan Mukund, and V. Vinay. Vol. 2. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008, pp. 364–382. DOI: 10.4230/LIPIcs.FSTTCS.2008.1768.

[86] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Vol. 2500. Lecture Notes in Computer Science. Springer-Verlag, 2002.

[87] Charles Grellois. "Semantics of linear logic and higher-order model-checking. (Sémantique de la logique linéaire et "model-checking" d'ordre supérieur)". PhD thesis. Paris Diderot University, France, 2016.

[88] Charles Grellois and Paul-André Melliès. "Finitary Semantics of Linear Logic and Higher-Order Model-Checking". In: *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part I*. Ed. by Giuseppe F. Italiano, Giovanni Pighizzini, and Donald Sannella. Vol. 9234. Lecture Notes in Computer Science. Springer, 2015, pp. 256–268. DOI: 10.1007/978-3-662-48057-1_20.

[89] Y. Gurevich and S. Shelah. "Rabin's Uniformization Problem." In: *J. Symb. Log.* 48.4 (1983), pp. 1105–1119.

[90] Yuri Gurevich and Leo Harrington. "Trees, Automata, and Games". In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*. Ed. by Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber. ACM, 1982, pp. 60–65. DOI: 10.1145/800070.802177.

[91] Axel Haddad. "Model Checking and Functional Program Transformations". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*. Ed. by Anil Seth and Nisheeth K. Vishnoi. Vol. 24. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013, pp. 115–126. DOI: 10.4230/LIPIcs.FSTTCS.2013.115.

[92] Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. "Collapsible Pushdown Automata and Recursion Schemes". In: *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 2008, pp. 452–461. DOI: 10.1109/LICS.2008.34.

[93] Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. "Collapsible Pushdown Automata and Recursion Schemes". In: *ACM Trans. Comput. Log.* 18.3 (2017), 25:1–25:42. DOI: 10.1145/3091122.

[94] Matthew Hague and C.-H. Luke Ong. "A saturation method for the modal μ-calculus over pushdown systems". In: *Inf. Comput.* 209.5 (2011), pp. 799–821. DOI: 10.1016/j.ic.2010.12.004.

[95] Stephan Heilbrunner. "An Algorithm for the Solution of Fixed-Point Equations for Infinite Words". In: *ITA* 14.2 (1980), pp. 131–141. DOI: 10.1051/ita/1980140201311.

[96] J. M. E. Hyland and C.-H. Luke Ong. "On Full Abstraction for PCF: I, II, and III". In: *Inf. Comput.* 163.2 (2000), pp. 285–408. DOI: 10.1006/inco.2000.2917.

[97] David Janin and Igor Walukiewicz. "On the Expressive Completeness of the Propositional mu-Calculus with Respect to Monadic Second Order Logic". In: *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*. Ed. by Ugo Montanari and Vladimiro Sassone. Vol. 1119. Lecture Notes in Computer Science. Springer, 1996, pp. 263–277. DOI: 10.1007/3-540-61604-7_60.

[98] Daniel Kirsten. "Alternating Tree Automata and Parity Games". In: *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Ed. by Erich Grädel, Wolfgang Thomas, and Thomas Wilke. Vol. 2500. Lecture Notes in Computer Science. Springer, 2001, pp. 153–167. DOI: 10.1007/3-540-36387-4_9.

[99] Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. "Deciding Monadic Theories of Hyperalgebraic Trees". In: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*. Ed. by Samson Abramsky. Vol. 2044. Lecture Notes in Computer Science. Springer, 2001, pp. 253–267. DOI: 10.1007/3-540-45413-6_21.

[100] Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. "Higher-Order Pushdown Trees Are Easy". In: *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by Mogens Nielsen and Uffe Engberg. Vol. 2303. Lecture Notes in Computer Science. Springer, 2002, pp. 205–222. DOI: 10.1007/3-540-45931-6_15.

[101] Teodor Knapik, Damian Niwinski, Pawel Urzyczyn, and Igor Walukiewicz. "Unsafe Grammars and Panic Automata". In: *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*. Ed. by Luıs Caires, Giuseppe F. Italiano, Luıs Monteiro, Catuscia Palamidessi, and Moti Yung. Vol. 3580. Lecture Notes in Computer Science. Springer, 2005, pp. 1450–1461. DOI: 10.1007/11523468_117.

[102] Naoki Kobayashi. "A Practical Linear Time Algorithm for Trivial Automata Model Checking of Higher-Order Recursion Schemes". In: *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011.*

Proceedings. Ed. by Martin Hofmann. Vol. 6604. Lecture Notes in Computer Science. Springer, 2011, pp. 260–274. DOI: 10.1007/978-3-642-19805-2_18.

[103] Naoki Kobayashi. "Types and higher-order recursion schemes for verification of higher-order programs". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 416–428. DOI: 10.1145/1480881.1480933.

[104] Naoki Kobayashi, Étienne Lozes, and Florian Bruse. "On the relationship between higher-order recursion schemes and higher-order fixpoint logic". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 246–259. DOI: 10.1145/3009837.3009854.

[105] Naoki Kobayashi and C.-H. Luke Ong. "A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes". In: *2009 24th Annual IEEE Symposium on Logic In Computer Science* (Aug. 2009). DOI: 10.1109/lics.2009.29.

[106] Patrick Landwehr and Christof Löding. "Projection for Büchi Tree Automata with Constraints Between Siblings". In: *Developments in Language Theory - 22nd International Conference, DLT 2018, Tokyo, Japan, September 10-14, 2018, Proceedings*. Ed. by Mizuho Hoshi and Shinnosuke Seki. Vol. 11088. Lecture Notes in Computer Science. Springer, 2018, pp. 478–490. DOI: 10.1007/978-3-319-98654-8_39.

[107] Patrick Landwehr and Christof Löding. "Tree Automata with Global Constraints for Infinite Trees". In: *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany*. Ed. by Rolf Niedermeier and Christophe Paul. Vol. 126. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, 47:1–47:15. DOI: 10.4230/LIPIcs.STACS.2019.47.

[108] Christof Löding. "Infinite Games and Automata Theory". In: *Lectures in Game Theory for Computer Scientists*. Ed. by Krzysztof R. Apt and Erich Grädel. Cambridge University Press, 2011, pp. 38–73.

[109] A. N. Maslov. "Multilevel Stack Automata". In: *Problems of Information Transmission* 12 (1976), pp. 38–43.

[110] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[111] 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. DOI: 10.1016/0304-3975(85)90087-8.

[112] Robin P. Neatherway, Steven J. Ramsay, and C.-H. Luke Ong. "A traversal-based algorithm for higher-order model checking". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. Ed. by Peter Thiemann and Robby Bruce Findler. ACM, 2012, pp. 353–364. DOI: 10.1145/2364527.2364578.

[113] Maurice Nivat. “Langages algébriques sur le magma libre et sémantique des schémas de programme”. In: *Proceedings of the 1st International Colloquium on Automata, Languages and Programming, Colloquium (ICALP)*. Ed. by Maurice Nivat. North-Holland, 1972, pp. 293–308.

[114] Damian Niwinski. “Fixed Points vs. Infinite Generation”. In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS ’88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 1988, pp. 402–409. DOI: 10.1109/LICS.1988.5137.

[115] Damian Niwinski. “On the Cardinality of Sets of Infinite Trees Recognizable by Finite Automata”. In: *Mathematical Foundations of Computer Science 1991, 16th International Symposium, MFCS’91, Kazimierz Dolny, Poland, September 9-13, 1991, Proceedings*. Ed. by Andrzej Tarlecki. Vol. 520. Lecture Notes in Computer Science. Springer, 1991, pp. 367–376. DOI: 10.1007/3-540-54345-7_80.

[116] Damian Niwinski and Igor Walukiewicz. “A gap property of deterministic tree languages”. In: *Theor. Comput. Sci.* 303.1 (2003), pp. 215–231. DOI: 10.1016/S0304-3975(02)00452-8.

[117] C.-H. Luke Ong. “Normalisation by Traversals”. In: *CoRR* abs/1511.02629 (2015). arXiv: 1511.02629.

[118] C.-H. Luke 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. DOI: 10.1109/LICS.2006.38.

[119] Luke Ong. “Higher-Order Model Checking: An Overview”. In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. IEEE Computer Society, 2015, pp. 1–15. DOI: 10.1109/LICS.2015.9.

[120] David Michael Ritchie Park. “Concurrency and Automata on Infinite Sequences”. In: *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*. Ed. by Peter Deussen. Vol. 104. Lecture Notes in Computer Science. Springer, 1981, pp. 167–183. DOI: 10.1007/BFb0017309.

[121] Pawel Parys. “Homogeneity Without Loss of Generality”. In: *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*. Ed. by Hélène Kirchner. Vol. 108. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 27:1–27:15. DOI: 10.4230/LIPIcs.FSCD.2018.27.

[122] Pawel Parys. “Intersection Types for Unboundedness Problems”. In: *Proceedings Twelfth Workshop on Developments in Computational Models and Ninth Workshop on Intersection Types and Related Systems, DCM/ITRS 2018, Oxford, UK, 8th July 2018*. Ed. by Michele Pagani and Sandra Alves. Vol. 293. EPTCS. 2018, pp. 7–27. DOI: 10.4204/EPTCS.293.2.

[123] Pawel Parys. “On the Significance of the Collapse Operation”. In: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. IEEE Computer Society, 2012, pp. 521–530. DOI: 10.1109/LICS.2012.62.

[124] Michael Rabin. "Decidability of second-order theories and automata on infinite trees". In: *Trans. Amer. Math. Soc.* 141 (1969), pp. 1–35.

[125] Fabian Reiter. "Distributed Automata and Logic. (Automates Distribués et Logique)". PhD thesis. Sorbonne Paris Cité, France, 2017.

[126] Joseph G. Rosenstein. *Linear Orderings*. Pure and applied mathematics. Academic Press, 1981.

[127] Sylvain Salvati and Igor Walukiewicz. "A Model for Behavioural Properties of Higher-order Programs". In: *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*. Ed. by Stephan Kreutzer. Vol. 41. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 229–243. DOI: 10.4230/LIPIcs.CSL.2015.229.

[128] Sylvain Salvati and Igor Walukiewicz. "Krivine Machines and Higher-Order Schemes". In: *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*. Ed. by Luca Aceto, Monika Henzinger, and Jirı Sgall. Vol. 6756. Lecture Notes in Computer Science. Springer, 2011, pp. 162–173. DOI: 10.1007/978-3-642-22012-8_12.

[129] Sylvain Salvati and Igor Walukiewicz. "Recursive Schemes, Krivine Machines, and Collapsible Pushdown Automata". In: *Reachability Problems - 6th International Workshop, RP 2012, Bordeaux, France, September 17-19, 2012. Proceedings*. Ed. by Alain Finkel, Jérôme Leroux, and Igor Potapov. Vol. 7550. Lecture Notes in Computer Science. Springer, 2012, pp. 6–20. DOI: 10.1007/978-3-642-33512-9_2.

[130] Peter Selinger. "Lecture notes on the lambda calculus". In: *CoRR* abs/0804.3434 (2008). arXiv: 0804.3434.

[131] Alexei L. Semenov. "Decidability of Monadic Theories". In: *Mathematical Foundations of Computer Science 1984, Praha, Czechoslovakia, September 3-7, 1984, Proceedings*. Ed. by Michal Chytil and Václav Koubek. Vol. 176. Lecture Notes in Computer Science. Springer, 1984, pp. 162–175. DOI: 10.1007/BFb0030296.

[132] Géraud Sénizergues. "L(A)=L(B)? decidability results from complete formal systems". In: *Theor. Comput. Sci.* 251.1-2 (2001), pp. 1–166. DOI: 10.1016/S0304-3975(00)00285-1.

[133] Géraud Sénizergues. "The Equivalence Problem for Deterministic Pushdown Automata is Decidable". In: *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*. Ed. by Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela. Vol. 1256. Lecture Notes in Computer Science. Springer, 1997, pp. 671–681. DOI: 10.1007/3-540-63165-8_221.

[134] Olivier Serre. "Note on winning positions on pushdown games with [omega]-regular conditions". In: *Inf. Process. Lett.* 85.6 (2003), pp. 285–291. DOI: 10.1016/S0020-0190(02)00445-3.

[135] Colin Stirling. *Decidability of Bisimulation Equivalence for Pushdown Processes*. Tech. rep. EDI-INF-RR-0005. School of Informatics, University of Edinburgh, 2000.

[136] L. J. Stockmeyer. "The complexity of decision problems in automata and logic". PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, 1974.

[137] Larry J. Stockmeyer and Albert R. Meyer. "Word Problems Requiring Exponential Time: Preliminary Report". In: *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*. Ed. by Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong. ACM, 1973, pp. 1–9. DOI: 10.1145/800125.804029.

[138] Wolfgang Thomas. "Languages, Automata, and Logic". In: *Handbook of Formal Language Theory*. Ed. by G. Rozenberg and A. Salomaa. Vol. III. Springer-Verlag, 1997, pp. 389–455.

[139] Moshe Y. Vardi. "Reasoning about The Past with Two-Way Automata". In: *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*. Ed. by Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel. Vol. 1443. Lecture Notes in Computer Science. Springer, 1998, pp. 628–641. DOI: 10.1007/BFb0055090.

[140] Mahesh Viswanathan and Ramesh Viswanathan. "A Higher Order Modal Fixed Point Logic". In: *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*. Ed. by Philippa Gardner and Nobuko Yoshida. Vol. 3170. Lecture Notes in Computer Science. Springer, 2004, pp. 512–528. DOI: 10.1007/978-3-540-28644-8_33.

[141] Hagen Völzer and Daniele Varacca. "Defining Fairness in Reactive and Concurrent Systems". In: *J. ACM* 59.3 (2012), 13:1–13:37. DOI: 10.1145/2220357.2220360.

[142] Igor Walukiewicz. "Lambda Y-Calculus With Priorities". In: *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 2019, pp. 1–13. DOI: 10.1109/LICS.2019.8785674.

[143] Igor Walukiewicz. "Monadic second-order logic on tree-like structures". In: *Theoretical Computer Science* 275.1-2 (Mar. 2002), pp. 311–346. DOI: 10.1016/s0304-3975(01)00185-2.

[144] Igor Walukiewicz. "Pushdown Processes: Games and Model-Checking". In: *Inf. Comput.* 164.2 (2001), pp. 234–263. DOI: 10.1006/inco.2000.2894.