Tree Automata Techniques and Applications

Hubert Comon Max Dauchet Rémi Gilleron
Florent Jacquemard Denis Lugiez Christof Löding
Sophie Tison Marc Tommasi

Contents

Errata **9**

Introduction **11**

Preliminaries **15**

1 Recognizable Tree Languages and Finite Tree Automata **19**
- 1.1 Finite Tree Automata 20
- 1.2 The Pumping Lemma for Recognizable Tree Languages 28
- 1.3 Closure Properties of Recognizable Tree Languages 29
- 1.4 Tree Homomorphisms 31
- 1.5 Minimizing Tree Automata 35
- 1.6 Top Down Tree Automata 38
- 1.7 Decision Problems and their Complexity 39
- 1.8 Exercises 43
- 1.9 Bibliographic Notes 47

2 Regular Grammars and Regular Expressions **51**
- 2.1 Tree Grammar 51
  - 2.1.1 Definitions 51
  - 2.1.2 Regularity and Recognizabilty 54
- 2.2 Regular Expressions. Kleene's Theorem for Tree Languages . . . 54
  - 2.2.1 Substitution and Iteration 55
  - 2.2.2 Regular Expressions and Regular Tree Languages 57
- 2.3 Regular Equations 61
- 2.4 Context-free Word Languages and Regular Tree Languages . . . 63
- 2.5 Beyond Regular Tree Languages: Context-free Tree Languages . 66
  - 2.5.1 Context-free Tree Languages 66
  - 2.5.2 IO and OI Tree Grammars 67
- 2.6 Exercises 68
- 2.7 Bibliographic notes 71

3 Logic, Automata and Relations **73**
- 3.1 Introduction 73
- 3.2 Automata on Tuples of Finite Trees 75
  - 3.2.1 Three Notions of Recognizability 75
  - 3.2.2 Examples of The Three Notions of Recognizability 77
  - 3.2.3 Comparisons Between the Three Classes 78

3.2.4 Closure Properties for $Rec_\times$ and Rec; Cylindrification and Projection . . . 80
3.2.5 Closure of GTT by Composition and Iteration . . . 82
3.3 The Logic WSkS . . . 87
3.3.1 Syntax . . . 87
3.3.2 Semantics . . . 88
3.3.3 Examples . . . 88
3.3.4 Restricting the Syntax . . . 89
3.3.5 Definable Sets are Recognizable Sets . . . 90
3.3.6 Recognizable Sets are Definable . . . 94
3.3.7 Complexity Issues . . . 95
3.3.8 Extensions . . . 95
3.4 Examples of Applications . . . 96
3.4.1 Terms and Sorts . . . 96
3.4.2 The Encompassment Theory for Linear Terms . . . 97
3.4.3 The First-order Theory of a Reduction Relation: the Case Where no Variables are Shared . . . 99
3.4.4 Reduction Strategies . . . 100
3.4.5 Application to Rigid E-unification . . . 102
3.4.6 Application to Higher-order Matching . . . 103
3.5 Exercises . . . 106
3.6 Bibliographic Notes . . . 109
3.6.1 GTT . . . 109
3.6.2 Automata and Logic . . . 109
3.6.3 Surveys . . . 109
3.6.4 Applications of tree automata to constraint solving . . . 109
3.6.5 Application of tree automata to semantic unification . . . 110
3.6.6 Application of tree automata to decision problems in term rewriting . . . 110
3.6.7 Other applications . . . 111

4 Automata with Constraints 113
4.1 Introduction . . . 113
4.2 Automata with Equality and Disequality Constraints . . . 114
4.2.1 The Most General Class . . . 114
4.2.2 Reducing Non-determinism and Closure Properties . . . 117
4.2.3 Decision Problems . . . 120
4.3 Automata with Constraints Between Brothers . . . 122
4.3.1 Definition . . . 122
4.3.2 Closure Properties . . . 122
4.3.3 Emptiness Decision . . . 122
4.3.4 Applications . . . 125
4.4 Reduction Automata . . . 125
4.4.1 Definition . . . 126
4.4.2 Closure Properties . . . 126
4.4.3 Emptiness Decision . . . 127
4.4.4 Finiteness Decision . . . 132
4.4.5 Term Rewriting Systems . . . 132
4.4.6 Application to the Reducibility Theory . . . 133
4.5 Other Decidable Subclasses . . . 133

4.6 Exercises 133
4.7 Bibliographic notes 135

5 Tree Set Automata **137**
5.1 Introduction 137
5.2 Definitions and Examples 142
5.2.1 Generalized Tree Sets 142
5.2.2 Tree Set Automata 142
5.2.3 Hierarchy of GTSA-recognizable Languages 145
5.2.4 Regular Generalized Tree Sets, Regular Runs 146
5.3 Closure and Decision Properties 149
5.3.1 Closure properties 149
5.3.2 Emptiness Property 152
5.3.3 Other Decision Results 154
5.4 Applications to Set Constraints 155
5.4.1 Definitions 155
5.4.2 Set Constraints and Automata 155
5.4.3 Decidability Results for Set Constraints 156
5.5 Bibliographical Notes 158

6 Tree Transducers **161**
6.1 Introduction 161
6.2 The Word Case 162
6.2.1 Introduction to Rational Transducers 162
6.2.2 The Homomorphic Approach 166
6.3 Introduction to Tree Transducers 167
6.4 Properties of Tree Transducers 171
6.4.1 Bottom-up Tree Transducers 171
6.4.2 Top-down Tree Transducers 174
6.4.3 Structural Properties 176
6.4.4 Complexity Properties 177
6.5 Homomorphisms and Tree Transducers 177
6.6 Exercises 178
6.7 Bibliographic notes 181

7 Alternating Tree Automata **183**
7.1 Introduction 183
7.2 Definitions and Examples 183
7.2.1 Alternating Word Automata 183
7.2.2 Alternating Tree Automata 185
7.2.3 Tree Automata versus Alternating Word Automata 187
7.3 Closure Properties 187
7.4 From Alternating to Deterministic Automata 188
7.5 Decision Problems and Complexity Issues 189
7.6 Horn Logic, Set Constraints and Alternating Automata 189
7.6.1 The Clausal Formalism 189
7.6.2 The Set Constraints Formalism 190
7.6.3 Two Way Alternating Tree Automata 191
7.6.4 Two Way Automata and Definite Set Constraints 193
7.6.5 Two Way Automata and Pushdown Automata 195

7.7 An (other) example of application 195
7.8 Exercises 195
7.9 Bibliographic Notes 196

8 Automata for Unranked Trees 199
8.1 Introduction 199
8.2 Definitions and Examples 200
  8.2.1 Unranked Trees and Hedges 200
  8.2.2 Hedge Automata 201
  8.2.3 Deterministic Automata 205
8.3 Encodings and Closure Properties 206
  8.3.1 First-Child-Next-Sibling Encoding 207
  8.3.2 Extension Operator 210
  8.3.3 Closure Properties 211
8.4 Weak Monadic Second Order Logic 212
8.5 Decision Problems and Complexity 214
  8.5.1 Representations of Horizontal Languages 214
  8.5.2 Determinism and Completeness 216
  8.5.3 Membership 216
  8.5.4 Emptiness 219
  8.5.5 Inclusion 220
8.6 Minimization 221
  8.6.1 Minimizing the Number of States 221
  8.6.2 Problems for Minimizing the Whole Representation . . . 223
  8.6.3 Stepwise automata 224
8.7 XML Schema Languages 229
  8.7.1 Document Type Definition (DTD) 232
  8.7.2 XML Schema 236
  8.7.3 Relax NG 239
8.8 Exercises 239
8.9 Bibliographic Notes 242

Bibliography 245

Index 259

Acknowledgments

Many people gave substantial suggestions to improve the contents of this book. These are, in alphabetic order, Witold Charatonik, Zoltan Fülöp, Werner Kuich, Markus Lohrey, Jun Matsuda, Aart Middeldorp, Hitoshi Ohsaki, P. K. Manivannan, Masahiko Sakai, Helmut Seidl, Stephan Tobies, Ralf Treinen, Thomas Uribe, Sandor Vágvölgyi, Kumar Neeraj Verma, Toshiyuki Yamada.

Errata

This is a list of some errors in the current release of TATA. For the next release these errors will be corrected. The list can also be found separately on `http://tata.gforge.inria.fr/`.

Chapter 8

1. Page 218: In the proof of Theorem 8.5.9(1) it is stated that DFHA(DFA) can easily be completed. This is not true. To complete a DFHA(DFA) one has to construct the complement of the union of several horizontal languages given by DFAs: Assume that there are several transitions $a(R_1) \to q_1, \ldots, a(R_n) \to q_n$ for letter a. To complete the automaton one has to add a transition $a(R) \to q_\perp$ with R the complement of $R_1 \cup \cdots \cup R_n$ and $q_\perp$ a rejecting sink state. Building an automaton for R can be expensive.

   The proof of (2) is not affected because the construction of a DFHA(DFA) from an NFHA(NFA) yields a complete DFHA(DFA), and for complete DFHA(DFA) Theorem 8.5.9(1) holds.

   The statement in Theorem 8.5.9(1) remains true if the DFHA(DFA) is normalized, i.e., for each a and q there is at most one transition $a(R) \to q$. In this case the inclusion problem can be reduced to the inclusion problem for unambiguous ranked tree automata; see Theorem 5 of

   W. Martens and J. Niehren. *On the Minimization of XML Schemas and Tree Automata for Unranked Trees.* Full version of DBPL 2005 paper. Journal of Computer and System Sciences (JCSS), 73(4), pp. 550-583, 2007.

2. Page 223, Figure 8.8: A transition with label q_{bb} is missing from state C_0 to state C_2.

3. Page 225, Figure 8.10, left-hand side: A transition with label BB is missing from state C_0 to state BB.

Introduction

During the past few years, several of us have been asked many times about references on finite tree automata. On one hand, this is the witness of the liveness of this field. On the other hand, it was difficult to answer. Besides several excellent survey chapters on more specific topics, there is only one monograph devoted to tree automata by Gécseg and Steinby. Unfortunately, it is now impossible to find a copy of it and a lot of work has been done on tree automata since the publication of this book. Actually using tree automata has proved to be a powerful approach to simplify and extend previously known results, and also to find new results. For instance recent works use tree automata for application in abstract interpretation using set constraints, rewriting, automated theorem proving and program verification, databases and XML schema languages.

Tree automata have been designed a long time ago in the context of circuit verification. Many famous researchers contributed to this school which was headed by A. Church in the late 50's and the early 60's: B. Trakhtenbrot, J.R. Büchi, M.O. Rabin, Doner, Thatcher, etc. Many new ideas came out of this program. For instance the connections between automata and logic. Tree automata also appeared first in this framework, following the work of Doner, Thatcher and Wright. In the 70's many new results were established concerning tree automata, which lose a bit their connections with the applications and were studied for their own. In particular, a problem was the very high complexity of decision procedures for the monadic second order logic. Applications of tree automata to program verification revived in the 80's, after the relative failure of automated deduction in this field. It is possible to verify temporal logic formulas (which are particular Monadic Second Order Formulas) on simpler (small) programs. Automata, and in particular tree automata, also appeared as an approximation of programs on which fully automated tools can be used. New results were obtained connecting properties of programs or type systems or rewrite systems with automata.

Our goal is to fill in the existing gap and to provide a textbook which presents the basics of tree automata and several variants of tree automata which have been devised for applications in the aforementioned domains. We shall discuss only *finite tree* automata, and the reader interested in infinite trees should consult any recent survey on automata on infinite objects and their applications (See the bibliography). The second main restriction that we have is to focus on the operational aspects of tree automata. This book should appeal the reader who wants to have a simple presentation of the basics of tree automata, and to see how some variations on the idea of tree automata have provided a nice tool for solving difficult problems. Therefore, specialists of the domain probably know almost all the material embedded. However, we think that this book can be helpful for many researchers who need some knowledge on tree automata. This is typically the case of a PhD student who may find new ideas and guess connections with his (her) own work.

Again, we recall that there is no presentation nor discussion of tree automata for infinite trees. This domain is also in full development mainly due to applications in program verification and several surveys on this topic do exist. We have tried to present a tool and the algorithms devised for this tool. Therefore, most of the proofs that we give are constructive and we have tried to give as many complexity results as possible. We don't claim to present an exhaustive description of all possible finite tree automata already presented in the literature and we did some choices in the existing menagerie of tree automata. Although some works are not described thoroughly (but they are usually described in exercises), we think that the content of this book gives a good flavor of what can be done with the simple ideas supporting tree automata.

This book is an open work and we want it to be as interactive as possible. Readers and specialists are invited to provide suggestions and improvements. Submissions of contributions to new chapters and improvements of existing ones are welcome.

Among some of our choices, let us mention that we have not defined any precise language for describing algorithms which are given in some pseudo algorithmic language. Also, there is no citation in the text, but each chapter ends with a section devoted to bibliographical notes where credits are made to the relevant authors. Exercises are also presented at the end of each chapter.

Tree Automata Techniques and Applications is composed of eight main chapters (numbered 1–8). The first one presents tree automata and defines recognizable tree languages. The reader will find the classical algorithms and the classical closure properties of the class of recognizable tree languages. Complexity results are given when they are available. The second chapter gives an alternative presentation of recognizable tree languages which may be more relevant in some situations. This includes regular tree grammars, regular tree expressions and regular equations. The description of properties relating regular tree languages and context-free word languages form the last part of this chapter. In Chapter 3, we show the deep connections between logic and automata. In particular, we prove in full details the correspondence between finite tree automata and the weak monadic second order logic with k successors. We also sketch several applications in various domains.

Chapter 4 presents a basic variation of automata, more precisely automata with equality constraints. An equality constraint restricts the application of rules to trees where some subtrees are equal (with respect to some equality relation). Therefore we can discriminate more easily between trees that we want to accept and trees that we must reject. Several kinds of constraints are described, both originating from the problem of non-linearity in trees (the same variable may occur at different positions).

In Chapter 5 we consider automata which recognize sets of sets of terms. Such automata appeared in the context of set constraints which themselves are used in program analysis. The idea is to consider, for each variable or each predicate symbol occurring in a program, the set of its possible values. The program gives constraints that these sets must satisfy. Solving the constraints gives an upper approximation of the values that a given variable can take. Such an approximation can be used to detect errors at compile time: it acts exactly as a typing system which would be inferred from the program. Tree set automata (as we call them) recognize the sets of solutions of such constraints (hence sets of sets of trees). In this chapter we study the properties of tree set automata and their relationship with program analysis.

Originally, automata were invented as an intermediate between function description and their implementation by a circuit. The main related problem in the sixties was the *synthesis problem*: which arithmetic recursive functions can be achieved by a circuit? So far, we only considered tree automata which accepts sets of trees or sets of tuples of trees (Chapter 3) or sets of sets of trees (Chapter 5). However, tree automata can also be used as a computational device. This is the subject of Chapter 6 where we study *tree transducers*.

In Chapter 7 we present basic properties of alternating automata. As an application we consider set constraints that have already been introduced in Chapter 5.

In Chapter 8 we drop the restriction for ranked trees that the label of a node determines the number of successors of this node, so we consider the model of unranked ordered trees. This model is used for representing the structure of XML documents. We discuss the basic model of hedge automaton, study its algorithmic and closure properties, and then present some formalisms used in practice to define classes of XML documents.

Preliminaries

Terms

We denote by N the set of positive integers. We denote the set of finite strings over N by N^*. The empty string is denoted by ε.

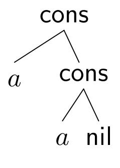
A ***ranked alphabet*** is a couple $(\mathcal{F}, Arity)$ where $\mathcal{F}$ is a finite set and $Arity$ is a mapping from $\mathcal{F}$ into N. The ***arity*** of a symbol $f \in \mathcal{F}$ is $Arity(f)$. The set of symbols of arity p is denoted by $\mathcal{F}_p$. Elements of arity $0, 1, \ldots p$ are respectively called constants, unary, ..., p-ary symbols. We assume that $\mathcal{F}$ contains at least one constant. In the examples, we use parenthesis and commas for a short declaration of symbols with arity. For instance, $f(,)$ is a short declaration for a binary symbol f.

Let $\mathcal{X}$ be a set of constants called ***variables***. We assume that the sets $\mathcal{X}$ and $\mathcal{F}_0$ are disjoint. The set $T(\mathcal{F}, \mathcal{X})$ of ***terms*** over the ranked alphabet $\mathcal{F}$ and the set of variables $\mathcal{X}$ is the smallest set defined by:

- $\mathcal{F}_0 \subseteq T(\mathcal{F}, \mathcal{X})$ and
- $\mathcal{X} \subseteq T(\mathcal{F}, \mathcal{X})$ and
- if $p \geq 1$, $f \in \mathcal{F}_p$ and $t_1, \ldots, t_p \in T(\mathcal{F}, \mathcal{X})$, then $f(t_1, \ldots, t_p) \in T(\mathcal{F}, \mathcal{X})$.

If $\mathcal{X} = \emptyset$ then $T(\mathcal{F}, \mathcal{X})$ is also written $T(\mathcal{F})$. Terms in $T(\mathcal{F})$ are called ***ground terms***. A term t in $T(\mathcal{F}, \mathcal{X})$ is ***linear*** if each variable occurs at most once in t.

Example 0.0.1. Let $\mathcal{F} = \{\mathsf{cons}(,), \mathsf{nil}, a\}$ and $\mathcal{X} = \{x, y\}$. Here cons is a binary symbol, nil and a are constants. The term $\mathsf{cons}(x, y)$ is linear; the term $\mathsf{cons}(x, \mathsf{cons}(x, \mathsf{nil}))$ is non linear; the term $\mathsf{cons}(a, \mathsf{cons}(a, \mathsf{nil}))$ is a ground term. Terms can be represented in a graphical way. For instance, the term $\mathsf{cons}(a, \mathsf{cons}(a, \mathsf{nil}))$ is represented by:



Terms and Trees

A finite ordered ***tree*** t over a set of labels E is a mapping from a prefix-closed set $\mathcal{P}os(t) \subseteq N^*$ into E. Thus, a term $t \in T(\mathcal{F}, \mathcal{X})$ may be viewed as a finite ordered ranked tree, the leaves of which are labeled with variables or constant symbols and the internal nodes are labeled with symbols of positive arity, with out-degree equal to the arity of the label, i.e. a term $t \in T(\mathcal{F}, \mathcal{X})$ can also be defined as a partial function $t : N^* \rightarrow \mathcal{F} \cup \mathcal{X}$ with domain $\mathcal{P}os(t)$ satisfying the following properties:

(i) $\mathcal{P}os(t)$ is nonempty and prefix-closed.

(ii) $\forall p \in \mathcal{P}os(t)$, if $t(p) \in \mathcal{F}_n, n \geq 1$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \{1, \ldots, n\}$.

(iii) $\forall p \in \mathcal{P}os(t)$, if $t(p) \in \mathcal{X} \cup \mathcal{F}_0$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \emptyset$.

We confuse terms and trees, that is by “tree” we always mean a finite ordered ranked tree satisfying (i), (ii) and (iii). In the later chapters we also consider unranked trees but we then we give the corresponding definitions in the respective chapter.

The reader should note that finite ordered trees with bounded rank k – i.e. there is a bound k on the out-degrees of internal nodes – can be encoded in finite ordered ranked trees: a label $e \in E$ is associated with k symbols $(e, 1)$ of arity 1, . . . , (e, k) of arity k.

Each element in $\mathcal{P}os(t)$ is called a ***position***. A ***frontier position*** is a position p such that $\forall j \in N$, $pj \notin \mathcal{P}os(t)$. The set of frontier positions is denoted by $\mathcal{FP}os(t)$. Each position p in t such that $t(p) \in \mathcal{X}$ is called a ***variable position***. The set of variable positions of p is denoted by $\mathcal{VP}os(t)$. We denote by $\mathcal{H}ead(t)$ the ***root symbol*** of t which is defined by $\mathcal{H}ead(t) = t(\varepsilon)$.

SubTerms

A ***subterm*** $t|_p$ of a term $t \in T(\mathcal{F}, \mathcal{X})$ at position p is defined by the following:

- $\mathcal{P}os(t|_p) = \{j \mid pj \in \mathcal{P}os(t)\}$,
- $\forall q \in \mathcal{P}os(t|_p)$, $t|_p(q) = t(pq)$.

We denote by $t[u]_p$ the term obtained by replacing in t the subterm $t|_p$ by u.

We denote by $\trianglerighteq$ the ***subterm ordering***, i.e. we write $t \trianglerighteq t'$ if t' is a subterm of t. We denote $t \triangleright t'$ if $t \trianglerighteq t'$ and $t \neq t'$.

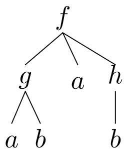
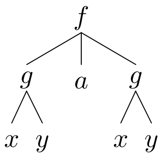
A set of terms F is said to be ***closed*** if it is closed under the subterm ordering, i.e. $\forall t \in F$ $(t \trianglerighteq t' \Rightarrow t' \in F)$.

Functions on Terms

The ***size*** of a term t, denoted by $\|t\|$ and the ***height*** of t, denoted by $\mathcal{H}eight(t)$ are inductively defined by:

- $\mathcal{H}eight(t) = 0$, $\|t\| = 0$ if $t \in \mathcal{X}$,
- $\mathcal{H}eight(t) = 1$, $\|t\| = 1$ if $t \in \mathcal{F}_0$,
- $\mathcal{H}eight(t) = 1 + \max(\{\mathcal{H}eight(t_i) \mid i \in \{1, \ldots, n\}\})$, $\|t\| = 1 + \sum_{i \in \{1, \ldots, n\}} \|t_i\|$ if $\mathcal{H}ead(t) \in \mathcal{F}_n$.

Example 0.0.2. Let $\mathcal{F} = \{f(,,), g(,), h(), a, b\}$ and $\mathcal{X} = \{x, y\}$. Consider the terms

$t =$  $;\ t' =$ 

The root symbol of t is f; the set of frontier positions of t is $\{11, 12, 2, 31\}$; the set of variable positions of t' is $\{11, 12, 31, 32\}$; $t|_3 = h(b)$; $t[a]_3 = f(g(a, b), a, a)$; $\mathcal{H}eight(t) = 3$; $\mathcal{H}eight(t') = 2$; $\|t\| = 7$; $\|t'\| = 4$.

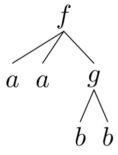
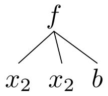
Substitutions

A ***substitution*** (respectively a ***ground substitution***) σ is a mapping from $\mathcal{X}$ into $T(\mathcal{F}, \mathcal{X})$ (respectively into $T(\mathcal{F})$) where there are only finitely many variables not mapped to themselves. The ***domain*** of a substitution σ is the subset of variables $x \in \mathcal{X}$ such that $\sigma(x) \neq x$. The substitution $\{x_1{\leftarrow}t_1, \ldots, x_n{\leftarrow}t_n\}$ is the identity on $\mathcal{X} \setminus \{x_1, \ldots, x_n\}$ and maps $x_i \in \mathcal{X}$ on $t_i \in T(\mathcal{F}, \mathcal{X})$, for every index $1 \leq i \leq n$. Substitutions can be extended to $T(\mathcal{F}, \mathcal{X})$ in such a way that:

$$\forall f \in \mathcal{F}_n, \forall t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X}) \quad \sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)).$$

We confuse a substitution and its extension to $T(\mathcal{F}, \mathcal{X})$. Substitutions will often be used in postfix notation: $t\sigma$ is the result of applying σ to the term t.

Example 0.0.3. Let $\mathcal{F} = \{f(,,), g(,), a, b\}$ and $\mathcal{X} = \{x_1, x_2\}$. Let us consider the term $t = f(x_1, x_1, x_2)$. Let us consider the ground substitution $\sigma = \{x_1{\leftarrow}a, x_2{\leftarrow}g(b, b)\}$ and the substitution $\sigma' = \{x_1{\leftarrow}x_2, x_2{\leftarrow}b\}$. Then

$t\sigma = t\{x_1{\leftarrow}a, x_2{\leftarrow}g(b, b)\} =$  $;\ t\sigma' = t\{x_1{\leftarrow}x_2, x_2{\leftarrow}b\} =$ 

Contexts

Let $\mathcal{X}_n$ be a set of n variables. A linear term $C \in T(\mathcal{F}, \mathcal{X}_n)$ is called a ***context*** and the expression $C[t_1, \ldots, t_n]$ for $t_1, \ldots, t_n \in T(\mathcal{F})$ denotes the term in $T(\mathcal{F})$ obtained from C by replacing variable x_i by t_i for each $1 \leq i \leq n$, that is $C[t_1, \ldots, t_n] = C\{x_1{\leftarrow}\, t_1, \ldots, x_n{\leftarrow}\, t_n\}$. We denote by $\mathcal{C}^n(\mathcal{F})$ the set of contexts over $(x_1, \ldots, x_n)$.

We denote by $\mathcal{C}(\mathcal{F})$ the set of contexts containing a single variable. A context is trivial if it is reduced to a variable. Given a context $C \in \mathcal{C}(\mathcal{F})$, we denote by C^0 the trivial context, C^1 is equal to C and, for $n > 1$, $C^n = C^{n-1}[C]$ is a context in $\mathcal{C}(\mathcal{F})$.

Chapter 1

Recognizable Tree Languages and Finite Tree Automata

In this chapter, we present basic results on finite tree automata in the style of the undergraduate textbook on finite automata by Hopcroft and Ullman [HU79]. Finite tree automata deal with finite ordered ranked trees or finite ordered trees with bounded rank. We discuss finite automata for unordered and/or unranked finite trees in subsequent chapters. We assume that the reader is familiar with finite automata. Words over a finite alphabet can be viewed as unary terms. For instance a word *abb* over $A = \{a, b\}$ can be viewed as a unary term $t = a(b(b(\sharp)))$ over the ranked alphabet $\mathcal{F} = \{a(), b(), \sharp\}$ where $\sharp$ is a new constant symbol. The theory of tree automata arises as a straightforward extension of the theory of word automata when words are viewed as unary terms.

In Section 1.1, we define bottom-up finite tree automata where "bottom-up" has the following sense: assuming a graphical representation of trees or ground terms with the root symbol at the top, an automaton starts its computation at the leaves and moves upward. Recognizable tree languages are the languages recognized by some finite tree automata. We consider the deterministic case and the nondeterministic case and prove the equivalence. In Section 1.2, we prove a pumping lemma for recognizable tree languages. This lemma is useful for proving that some tree languages are not recognizable. In Section 1.3, we prove the basic closure properties for set operations. In Section 1.4, we define tree homomorphisms and study the closure properties under these tree transformations. In this Section the first difference between the word case and the tree case appears. Indeed, recognizable word languages are closed under homomorphisms but recognizable tree languages are closed only under a subclass of tree homomorphisms: linear homomorphisms, where duplication of trees is forbidden. We will see all along this textbook that non linearity is one of the main difficulties for the tree case. In Section 1.5, we prove a Myhill-Nerode Theorem for tree languages and the existence of a unique minimal automaton. In Section 1.6, we define top-down tree automata. A second difference appears with the word case because it is proved that deterministic top-down tree automata are strictly less powerful than nondeterministic ones. The last section of the present chapter gives a list of complexity results.

1.1 Finite Tree Automata

Nondeterministic Finite Tree Automata

A ***finite Tree Automaton*** (NFTA) over $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where Q is a set of (unary) states, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the following type:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)),$$

where $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in \mathcal{X}$.

Tree automata over $\mathcal{F}$ run on ground terms over $\mathcal{F}$. An automaton starts at the leaves and moves upward, associating along a run a state with each subterm inductively. Let us note that there is no initial state in a NFTA, but, when $n = 0$, i.e. when the symbol is a constant symbol a, a transition rule is of the form $a \to q(a)$. Therefore, the transition rules for the constant symbols can be considered as the "initial rules". If the direct subterms $u_1, \ldots, u_n$ of $t = f(u_1, \ldots, u_n)$ are labeled with states $q_1, \ldots, q_n$, then the term t will be labeled by some state q with $f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta$. We now formally define the move relation defined by a NFTA.

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a NFTA over $\mathcal{F}$. The ***move relation*** $\to_{\mathcal{A}}$ is defined by: let t, $t' \in T(\mathcal{F} \cup Q)$,

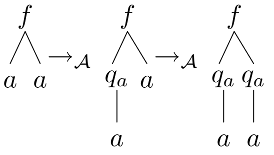
$$t \underset{\mathcal{A}}{\to} t' \Leftrightarrow \begin{cases} \exists C \in \mathcal{C}(\mathcal{F} \cup Q), \exists u_1, \ldots, u_n \in T(\mathcal{F}), \\ \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta, \\ t = C[f(q_1(u_1), \ldots, q_n(u_n))], \\ t' = C[q(f(u_1, \ldots, u_n))]. \end{cases}$$

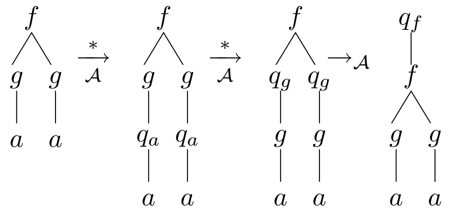
$\xrightarrow[\mathcal{A}]{*}$ is the reflexive and transitive closure of $\to_{\mathcal{A}}$.

Example 1.1.1. Let $\mathcal{F} = \{f(,), g(), a\}$. Consider the automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = \{q_a, q_g, q_f\}$, $Q_f = \{q_f\}$, and Δ is the following set of transition rules:

$$\begin{array}{lrclrcll} \{ & a & \to & q_a(a) & g(q_a(x)) & \to & q_g(g(x)) & \\ & g(q_g(x)) & \to & q_g(g(x)) & f(q_g(x), q_g(y)) & \to & q_f(f(x, y)) & \} \end{array}$$

We give two examples of reductions with the move relation $\to_{\mathcal{A}}$




f
g g
a a
$\xrightarrow[\mathcal{A}]{*}$
f
g g
q_a q_a
a a
$\xrightarrow[\mathcal{A}]{*}$
f
q_g q_g
g g
a a
$\rightarrow_{\mathcal{A}}$
q_f
f
g g
a a

A ground term t in $T(\mathcal{F})$ is ***accepted*** by a finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ if

$$t \xrightarrow[\mathcal{A}]{*} q(t)$$

for some state q in Q_f. The reader should note that our definition corresponds to the notion of nondeterministic finite tree automaton because our finite tree automaton model allows zero, one or more transition rules with the same left-hand side. Therefore there are possibly more than one reduction starting with the same ground term. And, a ground term t is accepted if there is one reduction (among all possible reductions) starting from this ground term and leading to a configuration of the form $q(t)$ where q is a final state. The tree language $L(\mathcal{A})$ ***recognized*** by $\mathcal{A}$ is the set of all ground terms accepted by $\mathcal{A}$. A set L of ground terms is ***recognizable*** if $L = L(\mathcal{A})$ for some NFTA $\mathcal{A}$. The reader should also note that when we talk about the set recognized by a finite tree automaton $\mathcal{A}$ we are referring to the specific set $L(\mathcal{A})$, not just any set of ground terms all of which happen to be accepted by $\mathcal{A}$. Two NFTA are said to be ***equivalent*** if they recognize the same tree languages.

Example 1.1.2. Let $\mathcal{F} = \{f(,), g(), a\}$. Consider the automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = \{q, q_g, q_f\}$, $Q_f = \{q_f\}$, and $\Delta =$

$$\begin{array}{lrcl rcl}
\{ & a & \rightarrow & q(a) & g(q(x)) & \rightarrow & q(g(x)) \\
 & g(q(x)) & \rightarrow & q_g(g(x)) & g(q_g(x)) & \rightarrow & q_f(g(x)) \\
 & f(q(x), q(y)) & \rightarrow & q(f(x, y)) \quad \}. & & &
\end{array}$$

We now consider a ground term t and exhibit three different reductions of term t w.r.t. move relation $\rightarrow_{\mathcal{A}}$.

$$\begin{array}{llcll}
t = g(g(f(g(a), a))) & \xrightarrow[\mathcal{A}]{*} & g(g(f(q_g(g(a)), q(a)))) & & \\
t = g(g(f(g(a), a))) & \xrightarrow[\mathcal{A}]{*} & g(g(q(f(g(a), a)))) & \xrightarrow[\mathcal{A}]{*} & q(t) \\
t = g(g(f(g(a), a))) & \xrightarrow[\mathcal{A}]{*} & g(g(q(f(g(a), a)))) & \xrightarrow[\mathcal{A}]{*} & q_f(t)
\end{array}$$

The term t is accepted by $\mathcal{A}$ because of the third reduction. It is easy to prove that $L(\mathcal{A}) = \{g(g(t)) \mid t \in T(\mathcal{F})\}$ is the set of ground instances of $g(g(x))$.

The set of transition rules of a NFTA $\mathcal{A}$ can also be defined as a ground rewrite system, i.e. a set of ground transition rules of the form: $f(q_1, \ldots, q_n) \rightarrow q$. A move relation $\rightarrow_{\mathcal{A}}$ can be defined as before. The only difference is that, now, we "forget" the ground subterms. And, a term t is accepted by a NFTA $\mathcal{A}$ if

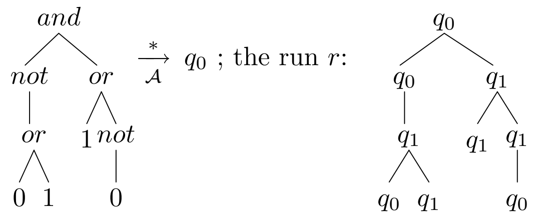
$$t \xrightarrow[\mathcal{A}]{*} q$$

for some final state q in Q_f. Unless it is stated otherwise, we will now refer to the definition with a set of ground transition rules. Considering a reduction starting from a ground term t and leading to a state q with the move relation, it is useful to remember the "history" of the reduction, i.e. to remember in which states the ground subterms of t are reduced. For this, we will adopt the following definitions. Let t be a ground term and $\mathcal{A}$ be a NFTA, a ***run*** r of $\mathcal{A}$ on t is a mapping $r : \mathcal{P}os(t) \to Q$ compatible with Δ, i.e. for every position p in $\mathcal{P}os(t)$, if $t(p) = f \in \mathcal{F}_n$, $r(p) = q$, $r(pi) = q_i$ for each $i \in \{1, \ldots, n\}$, then $f(q_1, \ldots, q_n) \to q \in \Delta$. A run r of $\mathcal{A}$ on t is ***successful*** if $r(\epsilon)$ is a final state. And a ground term t is accepted by a NFTA $\mathcal{A}$ if there is a successful run r of $\mathcal{A}$ on t.

Example 1.1.3. Let $\mathcal{F} = \{or(,), and(,), not(), 0, 1\}$. Consider the automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = \{q_0, q_1\}$, $Q_f = \{q_1\}$, and $\Delta =$

$$\begin{array}{lrcllrcll}
\{ & 0 & \to & q_0 & & 1 & \to & q_1 & \\
 & not(q_0) & \to & q_1 & & not(q_1) & \to & q_0 & \\
 & and(q_0, q_0) & \to & q_0 & & and(q_0, q_1) & \to & q_0 & \\
 & and(q_1, q_0) & \to & q_0 & & and(q_1, q_1) & \to & q_1 & \\
 & or(q_0, q_0) & \to & q_0 & & or(q_0, q_1) & \to & q_1 & \\
 & or(q_1, q_0) & \to & q_1 & & or(q_1, q_1) & \to & q_1 & \}.
\end{array}$$

A ground term over $\mathcal{F}$ can be viewed as a boolean formula without variable and a run on such a ground term can be viewed as the evaluation of the corresponding boolean formula. For instance, we give a reduction for a ground term t and the corresponding run given as a tree



The tree language recognized by $\mathcal{A}$ is the set of true boolean expressions over $\mathcal{F}$.

NFTA with ϵ-rules

Like in the word case, it is convenient to allow ϵ-moves in the reduction of a ground term by an automaton, i.e. the current state is changed but no new symbol of the term is processed. This is done by introducing a new type of rules in the set of transition rules of an automaton. A ***NFTA with ϵ-rules*** is like a NFTA except that now the set of transition rules contains ground transition rules of the form $f(q_1, \ldots, q_n) \rightarrow q$, and ϵ-rules of the form $q \rightarrow q'$. The ability to make ϵ-moves does not allow the NFTA to accept non recognizable sets. But NFTA with ϵ-rules are useful in some constructions and simplify some proofs.

Example 1.1.4. Let $\mathcal{F} = \{cons(,), s(), 0, nil\}$. Consider the automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = \{q_{\mathsf{Nat}}, q_{\mathsf{List}}, q_{\mathsf{List*}}\}$, $Q_f = \{q_{\mathsf{List}}\}$, and $\Delta =$

$$\begin{array}{lrcl rcl} \{ & 0 & \rightarrow & q_{\mathsf{Nat}} & s(q_{\mathsf{Nat}}) & \rightarrow & q_{\mathsf{Nat}} \\ & nil & \rightarrow & q_{\mathsf{List}} & cons(q_{\mathsf{Nat}}, q_{\mathsf{List}}) & \rightarrow & q_{\mathsf{List*}} \\ & q_{\mathsf{List*}} & \rightarrow & q_{\mathsf{List}}\}. & & & \end{array}$$

The recognized tree language is the set of Lisp-like lists of integers. If the final state set Q_f is set to $\{q_{\mathsf{List*}}\}$, then the recognized tree language is the set of non empty Lisp-like lists of integers. The ϵ-rule $q_{\mathsf{List*}} \rightarrow q_{\mathsf{List}}$ says that a non empty list is a list. The reader should recognize the definition of an order-sorted algebra with the sorts Nat, List, and List$*$ (which stands for the non empty lists), and the inclusion List$*$ $\subseteq$ List (see Section 3.4.1).

Theorem 1.1.5 (The equivalence of NFTAs with and without ϵ-rules)**.** *If L is recognized by a NFTA with ϵ-rules, then L is recognized by a NFTA without ϵ-rules.*

Proof. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a NFTA with ϵ-rules. Consider the subset Δ_ϵ consisting of those ϵ-rules in Δ. We denote by *ϵ-closure(q)* the set of all states q' in Q such that there is a reduction of q into q' using rules in Δ_ϵ. We consider that $q \in$ *ϵ-closure(q)*. This computation is a transitive closure computation and can be done in $O(|Q|^3)$. Now let us define the NFTA $\mathcal{A}' = (Q, \mathcal{F}, Q_f, \Delta')$ where Δ' is defined by:

$$\Delta' = \{f(q_1, \ldots, q_n) \rightarrow q' \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta, q' \in \epsilon\textit{-closure(q)}\}$$

Then it may be proved that $t \xrightarrow[\mathcal{A}]{*} q$ iff $t \xrightarrow[\mathcal{A}']{*} q$. □

Unless it is stated otherwise, we will now consider NFTA without ϵ-rules.

Deterministic Finite Tree Automata

Our definition of tree automata corresponds to the notion of nondeterministic finite tree automata. We will now define deterministic tree automata (DFTA) which are a special case of NFTA. It will turn out that, like in the word case, any language recognized by a NFTA can also be recognized by a DFTA.

A tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ is ***deterministic*** (DFTA) if there are no two rules with the same left-hand side (and no ϵ-rule). A DFTA is unambiguous, that is there is at most one run for every ground term, i.e. for every ground term t, there is at most one state q such that $t \xrightarrow[\mathcal{A}]{*} q$. The reader should note that it is possible to define a non deterministic tree automaton which is unambiguous (see Example 1.1.6).

It is also useful to consider tree automata such that there is at least one run for every ground term. This leads to the following definition. A NFTA $\mathcal{A}$

is ***complete*** if there is at least one rule $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$ for all $n \geq 0$, $f \in \mathcal{F}_n$, and $q_1, \ldots, q_n \in Q$. Let us note that for a complete DFTA there is exactly one run for every ground term.

Lastly, for practical reasons, it is usual to consider automata in which unnecessary states are eliminated. A state q is ***accessible*** if there exists a ground term t such that $t \xrightarrow[\mathcal{A}]{*} q$. A NFTA $\mathcal{A}$ is said to be ***reduced*** if all its states are accessible.

Example 1.1.6. The automaton defined in Example 1.1.2 is reduced, not complete, and it is not deterministic because there are two rules of left-hand side $g(q(x))$. Let us also note (see Example 1.1.2) that at least two runs (one is successful) can be defined on the term $g(g(f(g(a), a)))$.

The automaton defined in Example 1.1.3 is a complete and reduced DFTA.

Let $\mathcal{F} = \{g(), a\}$. Consider the automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = \{q_0, q_1, q\}$, $Q_f = \{q_0\}$, and Δ is the following set of transition rules:

$$\begin{array}{lrcl@{\qquad}rcl} \{ & a & \rightarrow & q_0 & g(q_0) & \rightarrow & q_1 \\ & g(q_1) & \rightarrow & q_0 & g(q) & \rightarrow & q_0 \\ & g(q) & \rightarrow & q_1\}. & & & \end{array}$$

This automaton is not deterministic because there are two rules of left-hand side $g(q)$, it is not reduced because state q is not accessible. Nevertheless, one should note that there is at most one run for every ground term t.

Let $\mathcal{F} = \{f(,), g(), a\}$. Consider the automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined in Example 1.1.1 by: $Q = \{q_a, q_g, q_f\}$, $Q_f = \{q_f\}$, and Δ is the following set of transition rules:

$$\begin{array}{lrcl@{\qquad}rcll} \{ & a & \rightarrow & q_a & g(q_a) & \rightarrow & q_g & \\ & g(q_g) & \rightarrow & q_g & f(q_g, q_g) & \rightarrow & q_f & \}. \end{array}$$

This automaton is deterministic and reduced. It is not complete because, for instance, there is no transition rule of left-hand side $f(q_a, q_a)$. It is easy to define a deterministic and complete automaton $\mathcal{A}'$ recognizing the same language by adding a "dead state". The automaton $\mathcal{A}' = (Q', \mathcal{F}, Q_f, \Delta')$ is defined by: $Q' = Q \cup \{\pi\}$, $\Delta' = \Delta \cup$

$$\begin{array}{lrcl@{\qquad}lcll} \{ & g(q_f) & \rightarrow & \pi & g(\pi) & \rightarrow & \pi & \\ & f(q_a, q_a) & \rightarrow & \pi & f(q_a, q_g) & \rightarrow & \pi & \\ & & \ldots & & f(\pi, \pi) & \rightarrow & \pi & \}. \end{array}$$

It is easy to generalize the construction given in Example 1.1.6 of a complete NFTA equivalent to a given NFTA: add a "dead state" π and all transition rules with right-hand side π such that the automaton is complete. The reader should note that this construction could be expensive because it may require $O(|\mathcal{F}| \times |Q|^{Arity(\mathcal{F})})$ new rules where $Arity(\mathcal{F})$ is the maximal arity of symbols in $\mathcal{F}$. Therefore we have the following:

Theorem 1.1.7. *Let L be a recognizable set of ground terms. Then there exists a complete finite tree automaton that accepts L.*

We now give a polynomial algorithm which outputs a reduced NFTA equivalent to a given NFTA as input. The main loop of this algorithm computes the set of accessible states.

```
Reduction Algorithm RED
input: NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
begin
    Set Marked to $\emptyset$ /* Marked is the set of accessible states */
    repeat
        Set Marked to $Marked \cup \{q\}$
        where
            $f \in \mathcal{F}_n$, $q_1, \ldots, q_n \in Marked$, $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$
    until no state can be added to Marked
    Set $Q_r$ to Marked
    Set $Q_{r_f}$ to $Q_f \cap Marked$
    Set $\Delta_r$ to $\{f(q_1, \ldots, q_n) \rightarrow q \in \Delta \mid q, q_1, \ldots, q_n \in Marked\}$
    output: NFTA $\mathcal{A}_r = (Q_r, \mathcal{F}, Q_{r_f}, \Delta_r)$
end
```

Let us recall that the case $n = 0$, in our notation, corresponds to rules of the form $a \rightarrow q$ where a is a constant symbol. Obviously all states in the set *Marked* are accessible, and an easy induction shows that all accessible states are in the set *Marked*. And, the NFTA $\mathcal{A}_r$ accepts the tree language $L(\mathcal{A})$. Consequently we have:

Theorem 1.1.8. *Let L be a recognizable set of ground terms. Then there exists a reduced finite tree automaton that accepts L.*

Now, we consider the reduction of nondeterminism. Since every DFTA is a NFTA, it is clear that the class of recognizable languages includes the class of languages accepted by DFTAs. However it turns out that these classes are equal. We prove that, for every NFTA, we can construct an equivalent DFTA. The proof is similar to the proof of equivalence between DFAs and NFAs in the word case. The proof is based on the "subset construction". Consequently, the number of states of the equivalent DFTA can be exponential in the number of states of the given NFTA (see Example 1.1.11). But, in practice, it often turns out that many states are not accessible. Therefore, we will present in the proof of the following theorem a construction of a DFTA where only the accessible states are considered, i.e. the given algorithm outputs an equivalent and reduced DFTA from a given NFTA as input.

Theorem 1.1.9 (The equivalence of DFTAs and NFTAs)**.** *Let L be a recognizable set of ground terms. Then there exists a DFTA that accepts L.*

Proof. First, we give a theoretical construction of a DFTA equivalent to a NFTA. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a NFTA. Define a DFTA $\mathcal{A}_d = (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$, as follows. The states of Q_d are all the subsets of the state set Q of $\mathcal{A}$. That is, $Q_d = 2^Q$. We denote by s a state of Q_d, i.e. $s = \{q_1, \ldots, q_n\}$ for some states

$q_1, \ldots, q_n \in Q$. We define

$$f(s_1, \ldots, s_n) \to s \in \Delta_d$$
$$\text{iff}$$
$$s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, \exists q_n \in s_n, f(q_1, \ldots, q_n) \to q \in \Delta\}.$$

And Q_{df} is the set of all states in Q_d containing a final state of $\mathcal{A}$. It is easy to prove that $L(\mathcal{A}) = L(\mathcal{A}_d)$. We now give an algorithmic construction where only the accessible states are considered.

```
Determinization Algorithm DET
input: NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
begin
    /* A state $s$ of the equivalent DFTA is in $2^Q$ */
    Set $Q_d$ to $\emptyset$; set $\Delta_d$ to $\emptyset$
    repeat
        Set $Q_d$ to $Q_d \cup \{s\}$; Set $\Delta_d$ to $\Delta_d \cup \{f(s_1, \ldots, s_n) \to s\}$
        where
            $f \in \mathcal{F}_n$, $s_1, \ldots, s_n \in Q_d$,
            $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n, f(q_1, \ldots, q_n) \to q \in \Delta\}$
    until no rule can be added to $\Delta_d$
    Set $Q_{df}$ to $\{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$
    output: DFTA $\mathcal{A}_d = (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$
end
```

It is immediate from the definition of the determinization algorithm that $\mathcal{A}_d$ is a deterministic and reduced tree automaton. In order to prove that $L(\mathcal{A}) = L(\mathcal{A}_d)$, we now prove that:

$$(t \xrightarrow[\mathcal{A}_d]{*} s) \text{ iff } (s = \{q \in Q \mid t \xrightarrow[\mathcal{A}]{*} q\}).$$

The proof is an easy induction on the structure of terms.

- base case: let us consider $t = a \in \mathcal{F}_0$. Then, there is only one rule $a \to s$ in Δ_d where $s = \{q \in Q \mid a \to q \in \Delta\}$.
- induction step: let us consider a term $t = f(t_1, \ldots, t_n)$.
  - First, let us suppose that $t \xrightarrow[\mathcal{A}_d]{*} f(s_1, \ldots, s_n) \to_{\mathcal{A}_d} s$. By induction hypothesis, for each $i \in \{1, \ldots, n\}$, $s_i = \{q \in Q \mid t_i \xrightarrow[\mathcal{A}]{*} q\}$. States s_i are in Q_d, thus a rule $f(s_1, \ldots, s_n) \to s \in \Delta_d$ is added in the set Δ_d by the determinization algorithm and $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n, f(q_1, \ldots, q_n) \to q \in \Delta\}$. Thus, $s = \{q \in Q \mid t \xrightarrow[\mathcal{A}]{*} q\}$.
  - Second, let us consider $s = \{q \in Q \mid t = f(t_1, \ldots, t_n) \xrightarrow[\mathcal{A}]{*} q\}$. Let us consider the state sets s_i defined by $s_i = \{q \in Q \mid t_i \xrightarrow[\mathcal{A}]{*} q\}$. By induction hypothesis, for each $i \in \{1, \ldots, n\}$, $t_i \xrightarrow[\mathcal{A}_d]{*} s_i$. Thus

$s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n, f(q_1, \ldots, q_n) \to q \in \Delta\}$. The rule $f(s_1, \ldots, s_n) \in \Delta_d$ by definition of the state set Δ_d in the determinization algorithm and $t \xrightarrow[\mathcal{A}_d]{*} s$.

$\square$

Example 1.1.10. Let $\mathcal{F} = \{f(,), g(), a\}$. Consider the automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined in Example 1.1.2 by: $Q = \{q, q_g, q_f\}$, $Q_f = \{q_f\}$, and $\Delta =$

$$\begin{array}{lrclrcl} \{ & a & \to & q & g(q) & \to & q \\ & g(q) & \to & q_g & g(q_g) & \to & q_f \\ & f(q,q) & \to & q & \}. & & \end{array}$$

Given $\mathcal{A}$ as input, the determinization algorithm outputs the DFTA $\mathcal{A}_d = (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$ defined by: $Q_d = \{\{q\}, \{q, q_g\}, \{q, q_g, q_f\}\}$, $Q_{df} = \{\{q, q_g, q_f\}\}$, and $\Delta_d =$

$$\begin{array}{llrclcll} & \{ & a & \to & \{q\} & & & \\ & & g(\{q\}) & \to & \{q, q_g\} & & & \\ & & g(\{q, q_g\}) & \to & \{q, q_g, q_f\} & & & \\ & & g(\{q, q_g, q_f\}) & \to & \{q, q_g, q_f\} & \} & & \\ \cup & \{ & f(s_1, s_2) & \to & \{q\} & \mid & s_1, s_2 \in Q_d & \}. \end{array}$$

We now give an example where an exponential blow-up occurs in the determinization process. This example is the same used in the word case.

Example 1.1.11. Let $\mathcal{F} = \{f(), g(), a\}$ and let n be an integer. And let us consider the tree language

$$L = \{t \in T(\mathcal{F}) \mid \text{the symbol at position } \underbrace{1 \ldots 1}_{n} \text{ is } f\}.$$

Let us consider the NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = \{q, q_1, \ldots, q_{n+1}\}$, $Q_f = \{q_{n+1}\}$, and $\Delta =$

$$\begin{array}{lrclrcll} \{ & a & \to & q & f(q) & \to & q & \\ & g(q) & \to & q & f(q) & \to & q_1 & \\ & g(q_1) & \to & q_2 & f(q_1) & \to & q_2 & \\ & & \vdots & & & \vdots & & \\ & g(q_n) & \to & q_{n+1} & f(q_n) & \to & q_{n+1} & \}. \end{array}$$

The NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ accepts the tree language L, and it has $n + 2$ states. Using the subset construction, the equivalent DFTA $\mathcal{A}_d$ has 2^{n+1} states. Any equivalent automaton has to memorize the $n + 1$ last symbols of the input tree. Therefore, it can be proved that any DFTA accepting L has at least 2^{n+1} states. It could also be proved that the automaton $\mathcal{A}_d$ is minimal in the number of states (minimal tree automata are defined in Section 1.5).

If a finite tree automaton is deterministic, we can replace the transition relation Δ by a transition function δ. Therefore, it is sometimes convenient to consider a DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ where

$$\delta : \bigcup_n \mathcal{F}_n \times Q^n \rightarrow Q \ .$$

The computation of such an automaton on a term t as input tree can be viewed as an evaluation of t on finite domain Q. Indeed, define the labeling function $\hat{\delta} : T(\mathcal{F}) \rightarrow Q$ inductively by

$$\hat{\delta}(f(t_1, \dots, t_n)) = \delta(f, \hat{\delta}(t_1), \dots, \hat{\delta}(t_n)) \ .$$

We shall for convenience confuse δ and $\hat{\delta}$.

We now make clear the connections between our definitions and the language theoretical definitions of tree automata and of recognizable tree languages. Indeed, the reader should note that a complete DFTA is just a finite $\mathcal{F}$-algebra $\mathcal{A}$ consisting of a finite carrier $|\mathcal{A}| = Q$ and a distinguished n-ary function $f^{\mathcal{A}} : Q^n \rightarrow Q$ for each n-ary symbol $f \in \mathcal{F}$ together with a specified subset Q_f of Q. A ground term t is accepted by $\mathcal{A}$ if $\delta(t) = q \in Q_f$ where δ is the unique $\mathcal{F}$-algebra homomorphism $\delta : T(\mathcal{F}) \rightarrow \mathcal{A}$.

Example 1.1.12. Let $\mathcal{F} = \{f(,), a\}$ and consider the $\mathcal{F}$-algebra $\mathcal{A}$ with $|\mathcal{A}| = Q = Z_2 = \{0, 1\}$, $f^{\mathcal{A}} = +$ where the sum is formed modulo 2, $a^{\mathcal{A}} = 1$, and let $Q_f = \{0\}$. $\mathcal{A}$ and Q_f defines a DFTA. The recognized tree language is the set of ground terms over $\mathcal{F}$ with an even number of leaves.

Since DFTA and NFTA accept the same sets of tree languages, we shall not distinguish between them unless it becomes necessary, but shall simply refer to both as tree automata (FTA).

1.2 The Pumping Lemma for Recognizable Tree Languages

We now give an example of a tree language which is not recognizable.

Example 1.2.1. Let $\mathcal{F} = \{f(,), g(), a\}$. Let us consider the tree language $L = \{f(g^i(a), g^i(a)) \mid i > 0\}$. Let us suppose that L is recognizable by an automaton $\mathcal{A}$ having k states. Now, consider the term $t = f(g^k(a), g^k(a))$. t belongs to L, therefore there is a successful run of $\mathcal{A}$ on t. As k is the cardinality of the state set, there are two distinct positions along the first branch of the term labeled with the same state. Therefore, one could cut the first branch between these two positions leading to a term $t' = f(g^j(a), g^k(a))$ with $j < k$ such that a successful run of $\mathcal{A}$ can be defined on t'. This leads to a contradiction with $L(\mathcal{A}) = L$.

This (sketch of) proof can be generalized by proving a ***pumping lemma*** for recognizable tree languages. This lemma is extremely useful in proving that certain sets of ground terms are not recognizable. It is also useful for solving decision problems like emptiness and finiteness of a recognizable tree language (see Section 1.7).

Pumping Lemma. *Let L be a recognizable set of ground terms. Then, there exists a constant $k > 0$ satisfying: for every ground term t in L such that $\mathcal{H}eight(t) > k$, there exist a context $C \in \mathcal{C}(\mathcal{F})$, a non trivial context $C' \in \mathcal{C}(\mathcal{F})$, and a ground term u such that $t = C[C'[u]]$ and, for all $n \geq 0$ $C[C'^n[u]] \in L$.*

Proof. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a FTA such that $L = L(\mathcal{A})$ and let $k = |Q|$ be the cardinality of the state set Q. Let us consider a ground term t in L such that $\mathcal{H}eight(t) > k$ and consider a successful run r of $\mathcal{A}$ on t. Now let us consider a path in t of length strictly greater than k. As k is defined to be the cardinality of the state set Q, there are two positions $p_1 < p_2$ along this path such that $r(p_1) = r(p_2) = q$ for some state q. Let u be the ground subterm of t at position p_2. Let u' be the ground subterm of t at position p_1, there exists a non-trivial context C' such that $u' = C'[u]$. Now define the context C such that $t = C[C'[u]]$. Consider a term $C[C'^n[u]]$ for some integer $n > 1$, a successful run can be defined on this term. Indeed suppose that r corresponds to the reduction $t \xrightarrow[\mathcal{A}]{*} q_f$ where q_f is a final state of $\mathcal{A}$, then we have:

$$C[C'^n[u]] \xrightarrow[\mathcal{A}]{*} C[C'^n[q]] \xrightarrow[\mathcal{A}]{*} C[C'^{n-1}[q]] \ldots \xrightarrow[\mathcal{A}]{*} C[q] \xrightarrow[\mathcal{A}]{*} q_f.$$

The same holds when $n = 0$. □

Example 1.2.2. Let $\mathcal{F} = \{f(,), a\}$. Let us consider the tree language $L = \{t \in T(\mathcal{F}) \mid |\mathcal{P}os(t)|$ is a prime number$\}$. We can prove that L is not recognizable. For all $k > 0$, consider a term t in L whose height is greater than k. For all contexts C, non trivial contexts C', and terms u such that $t = C[C'[u]]$, there exists n such that $C[C'^n[u]] \notin L$.

From the Pumping Lemma, we derive conditions for emptiness and finiteness given by the following corollary:

Corollary 1.2.3. *Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a FTA. Then $L(\mathcal{A})$ is non empty if and only if there exists a term t in $L(\mathcal{A})$ with $\mathcal{H}eight(t) \leq |Q|$. Then $L(\mathcal{A})$ is infinite if and only if there exists a term t in $L(\mathcal{A})$ with $|Q| < \mathcal{H}eight(t) \leq 2|Q|$.*

1.3 Closure Properties of Recognizable Tree Languages

A ***closure property*** of a class of (tree) languages is the fact that the class is closed under a particular operation. We are interested in effective closure properties where, given representations for languages in the class, there is an algorithm to construct a representation for the language that results by applying the operation to these languages. Let us note that the equivalence between NFTA and DFTA is effective, thus we may choose the representation that suits us best. Nevertheless, the determinization algorithm may output a DFTA whose number of states is exponential in the number of states of the given NFTA. For the different closure properties, we give effective constructions and we give the properties of the resulting FTA depending on the properties of the given FTA as input. In this section, we consider the Boolean set operations: union, intersection, and complementation. Other operations will be studied in the next sections. Complexity results are given in Section 1.7.

Theorem 1.3.1. *The class of recognizable tree languages is closed under union, under complementation, and under intersection.*

Union

Let L_1 and L_2 be two recognizable tree languages. Thus there are tree automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$ with $L_1 = L(\mathcal{A}_1)$ and $L_2 = L(\mathcal{A}_2)$. Since we may rename states of a tree automaton, without loss of generality, we may suppose that $Q_1 \cap Q_2 = \emptyset$. Now, let us consider the FTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = Q_1 \cup Q_2$, $Q_f = Q_{f1} \cup Q_{f2}$, and $\Delta = \Delta_1 \cup \Delta_2$. The equality between $L(\mathcal{A})$ and $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ is straightforward. Let us note that $\mathcal{A}$ is nondeterministic and not complete, even if $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic and complete.

We now give another construction which preserves determinism. The intuitive idea is to process in parallel a term by the two automata. For this we consider a product automaton. Let us suppose that $\mathcal{A}_1$ and $\mathcal{A}_2$ are complete. And, let us consider the FTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_2 \cup Q_1 \times Q_{f2}$, and $\Delta = \Delta_1 \times \Delta_2$ where

$$\begin{aligned}\Delta_1 \times \Delta_2 = \{ f((q_1, q'_1), \ldots, (q_n, q'_n)) \rightarrow (q, q') \mid \\ f(q_1, \ldots, q_n) \rightarrow q \in \Delta_1 \; f({q'}_1, \ldots, {q'}_n) \rightarrow q' \in \Delta_2 \}\end{aligned}$$

The proof of the equality between $L(\mathcal{A})$ and $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ is left to the reader, but the reader should note that the hypothesis that the two given tree automata are complete is crucial in the proof. Indeed, suppose for instance that a ground term t is accepted by $\mathcal{A}_1$ but not by $\mathcal{A}_2$. Moreover suppose that $\mathcal{A}_2$ is not complete and that there is no run of $\mathcal{A}_2$ on t, then the product automaton does not accept t because there is no run of the product automaton on t. The reader should also note that the construction preserves determinism, i.e. if the two given automata are deterministic, then the product automaton is also deterministic.

Complementation

Let L be a recognizable tree language. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a complete DFTA such that $L(\mathcal{A}) = L$. Now, complement the final state set to recognize the complement of L. That is, let $\mathcal{A}^c = (Q, \mathcal{F}, Q^c_f, \Delta)$ with $Q^c_f = Q \setminus Q_f$, the DFTA $\mathcal{A}^c$ recognizes the complement of set L in $T(\mathcal{F})$.

If the input automaton $\mathcal{A}$ is a NFTA, then first apply the determinization algorithm, and second complement the final state set. This could lead to an exponential blow-up.

Intersection

Closure under intersection follows from closure under union and complementation because

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}.$$

where we denote by $\overline{L}$ the complement of set L in $T(\mathcal{F})$. But if the recognizable tree languages are defined by NFTA, we have to use the complementation construction, therefore the determinization process is used leading to an exponential blow-up. Consequently, we now give a direct construction which does not use the determinization algorithm. Let $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$ be FTA such that $L(\mathcal{A}_1) = L_1$ and $L(\mathcal{A}_2) = L_2$. And, consider the FTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_{f2}$, and $\Delta = \Delta_1 \times \Delta_2$. $\mathcal{A}$ recognizes $L_1 \cap L_2$. Moreover the reader should note that $\mathcal{A}$ is deterministic if $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic.

1.4 Tree Homomorphisms

We now consider tree transformations and study the closure properties under these tree transformations. In this section we are interested with tree transformations preserving the structure of trees. Thus, we restrict ourselves to tree homomorphisms. Tree homomorphisms are a generalization of homomorphisms for words (considered as unary terms) to the case of arbitrary ranked alphabets. In the word case, it is known that the class of regular sets is closed under homomorphisms and inverse homomorphisms. The situation is different in the tree case because whereas recognizable tree languages are closed under inverse homomorphisms, they are closed only under a subclass of homomorphisms, i.e. linear homomorphisms (duplication of terms is forbidden). First, we define tree homomorphisms.

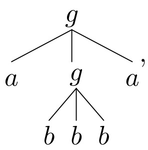
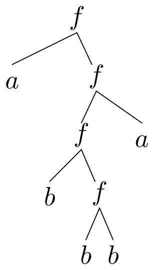
Let $\mathcal{F}$ and $\mathcal{F}'$ be two sets of function symbols, possibly not disjoint. For each $n > 0$ such that $\mathcal{F}$ contains a symbol of arity n, we define a set of variables $\mathcal{X}_n = \{x_1, \ldots, x_n\}$ disjoint from $\mathcal{F}$ and $\mathcal{F}'$.

Let $h_{\mathcal{F}}$ be a mapping which, with $f \in \mathcal{F}$ of arity n, associates a term $t_f \in T(\mathcal{F}', \mathcal{X}_n)$. The ***tree homomorphism*** $h : T(\mathcal{F}) \rightarrow T(\mathcal{F}')$ determined by $h_{\mathcal{F}}$ is defined as follows:

- $h(a) = t_a \in T(\mathcal{F}')$ for each $a \in \mathcal{F}$ of arity 0,
- $h(f(t_1, \ldots, t_n)) = t_f\{x_1 \leftarrow h(t_1), \ldots, x_n \leftarrow h(t_n)\}$

where $t_f\{x_1 \leftarrow h(t_1), \ldots, x_n \leftarrow h(t_n)\}$ is the result of applying the substitution $\{x_1 \leftarrow h(t_1), \ldots, x_n \leftarrow h(t_n)\}$ to the term t_f.

Example 1.4.1. Let $\mathcal{F} = \{g(,,), a, b\}$ and $\mathcal{F}' = \{f(,), a, b\}$. Let us consider the tree homomorphism h determined by $h_{\mathcal{F}}$ defined by: $h_{\mathcal{F}}(g) = f(x_1, f(x_2, x_3))$, $h_{\mathcal{F}}(a) = a$ and $h_{\mathcal{F}}(b) = b$. For instance, we have:

If $t =$  , then $h(t) =$ 

The homomorphism h defines a transformation from ternary trees into binary trees.

Let us now consider $\mathcal{F} = \{and(,), or(,), not(), 0, 1\}$ and $\mathcal{F}' = \{or(,), not(), 0, 1\}$. Let us consider the tree homomorphism h determined by $h_\mathcal{F}$ defined by: $h_\mathcal{F}(and) = not(or(not(x_1), not(x_2))$, and $h_\mathcal{F}$ is the identity otherwise. This homomorphism transforms a boolean formula in an equivalent boolean formula which does not contain the function symbol *and*.

A tree homomorphism is ***linear*** if for each $f \in \mathcal{F}$ of arity n, $h_\mathcal{F}(f) = t_f$ is a linear term in $T(\mathcal{F}', \mathcal{X}_n)$. The following example shows that tree homomorphisms do not always preserve recognizability.

Example 1.4.2. Let $\mathcal{F} = \{f(), g(), a\}$ and $\mathcal{F}' = \{f'(,), g(), a\}$. Let us consider the tree homomorphism h determined by $h_\mathcal{F}$ defined by: $h_\mathcal{F}(f) = f'(x_1, x_1)$, $h_\mathcal{F}(g) = g(x_1)$, and $h_\mathcal{F}(a) = a$. h is not linear. Let $L = \{f(g^i(a)) \mid i \geq 0\}$, then L is a recognizable tree language. $h(L) = \{f'(g^i(a), g^i(a)) \mid i \geq 0\}$ is not recognizable (see Example 1.2.1).

Theorem 1.4.3 (Linear homomorphisms preserve recognizability)**.** *Let h be a linear tree homomorphism and L be a recognizable tree language, then $h(L)$ is a recognizable tree language.*

Proof. Let L be a recognizable tree language. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a reduced DFTA such that $L(\mathcal{A}) = L$. Let h be a linear tree homomorphism from $T(\mathcal{F})$ into $T(\mathcal{F}')$ determined by a mapping $h_\mathcal{F}$.

First, let us define a NFTA $\mathcal{A}' = (Q', \mathcal{F}', Q'_f, \Delta')$. Let us consider a rule $r = f(q_1, \ldots, q_n) \rightarrow q$ in Δ and consider the linear term $t_f = h_\mathcal{F}(f) \in T(\mathcal{F}', \mathcal{X}_n)$ and the set of positions $\mathcal{P}os(t_f)$. We define a set of states $Q^r = \{q^r_p \mid p \in \mathcal{P}os(t_f)\}$, and we define a set of rules Δ_r as follows: for all positions p in $\mathcal{P}os(t_f)$

- if $t_f(p) = g \in \mathcal{F}'_k$, then $g(q^r_{p_1}, \ldots, q^r_{p_k}) \rightarrow q^r_p \in \Delta_r$,
- if $t_f(p) = x_i$, then $q_i \rightarrow q^r_p \in \Delta_r$,
- $q^r_\epsilon \rightarrow q \in \Delta_r$.

The preceding construction is made for each rule in Δ. We suppose that all the state sets Q^r are disjoint and that they are disjoint from Q. Now define $\mathcal{A}'$ by:

- $Q' = Q \cup \bigcup_{r \in \Delta} Q^r$,
- $Q'_f = Q_f$,

- $\Delta' = \bigcup_{r \in \Delta} \Delta_r$.

Second, we have to prove that $h(L) = L(\mathcal{A}')$.

$h(L) \subseteq L(\mathcal{A}')$. We prove that if $t \xrightarrow[\mathcal{A}]{*} q$ then $h(t) \xrightarrow[\mathcal{A}']{*} q$ by induction on the length of the reduction of ground term $t \in T(\mathcal{F})$ by automaton $\mathcal{A}$.

- Base case. Suppose that $t \rightarrow_{\mathcal{A}} q$. Then $t = a \in \mathcal{F}_0$ and $a \rightarrow q \in \Delta$. Then there is a reduction $h(a) = t_a \xrightarrow[\mathcal{A}']{*} q$ using the rules in the set $\Delta_{a \rightarrow q}$.
- Induction step.
  Suppose that $t = f(u_1, \ldots, u_n)$, then $h(t) = t_f\{x_1 \leftarrow h(u_1), \ldots, x_n \leftarrow h(u_n)\}$. Moreover suppose that $t \xrightarrow[\mathcal{A}]{*} f(q_1, \ldots, q_n) \rightarrow_{\mathcal{A}} q$. By induction hypothesis, we have $h(u_i) \xrightarrow[\mathcal{A}']{*} q_i$, for each i in $\{1, \ldots, n\}$. Then there is a reduction $t_f\{x_1 \leftarrow q_1, \ldots, x_n \leftarrow q_n\} \xrightarrow[\mathcal{A}']{*} q$ using the rules in the set $\Delta_{f(q_1,\ldots,q_n) \rightarrow q}$.

$h(L) \supseteq L(\mathcal{A}')$**.** We prove that if $t' \xrightarrow[\mathcal{A}']{*} q \in Q$ then $t' = h(t)$ with $t \xrightarrow[\mathcal{A}]{*} q$ for some $t \in T(\mathcal{F})$. The proof is by induction on the number of states in Q occurring along the reduction $t' \xrightarrow[\mathcal{A}']{*} q \in Q$.

- Base case. Suppose that $t' \xrightarrow[\mathcal{A}']{*} q \in Q$ and no state in Q apart from q occurs in the reduction. Then, because the state sets Q^r are disjoint, only rules of some Δ^r can be used in the reduction. Thus, t' is ground, $t' = h_{\mathcal{F}}(f)$ for some symbol $f \in \mathcal{F}$, and $r = f(q_1, \ldots, q_n) \rightarrow q$. Because the automaton is reduced, there is some ground term t with $\mathcal{H}ead(t) = f$ such that $t' = h(t)$ and $t \xrightarrow[\mathcal{A}]{*} q$.
- Induction step. Suppose that

  $$t' \xrightarrow[\mathcal{A}']{*} v\{x'_1 \leftarrow q_1, \ldots, x'_m \leftarrow q_m\} \xrightarrow[\mathcal{A}']{*} q$$

  where v is a linear term in $T(\mathcal{F}', \{x'_1, \ldots, x'_m\})$, $t' = v\{x'_1 \leftarrow u'_1, \ldots, x'_m \leftarrow u'_m\}$, $u'_i \xrightarrow[\mathcal{A}']{*} q_i \in Q$, and no state in Q apart from q occurs in the reduction of $v\{x'_1 \leftarrow q_1, \ldots, x'_m \leftarrow q_m\}$ to q. The reader should note that different variables can be substituted by the same state. Then, because the state sets Q^r are disjoint, only rules of some Δ^r can be used in the reduction of $v\{x'_1 \leftarrow q_1, \ldots, x'_m \leftarrow q_m\}$ to q. Thus, there exists some linear term t_f such that $v\{x'_1 \leftarrow q_1, \ldots, x'_m \leftarrow q_m\} = t_f\{x_1 \leftarrow q_1, \ldots, x_n \leftarrow q_n\}$ for some symbol $f \in \mathcal{F}_n$ and $r = f(q_1, \ldots, q_n) \rightarrow q \in \Delta$. By induction hypothesis, there are terms $u_1, \ldots, u_m$ in L such that $u'_i = h(u_i)$ and $u_i \xrightarrow[\mathcal{A}]{*} q_i$ for each i in $\{1, \ldots, m\}$. Now consider the term $t = f(v_1, \ldots, v_n)$, where $v_i = u_i$ if x_i occurs in t_f and v_i is some term such that $v_i \xrightarrow[\mathcal{A}]{*} q_i$ otherwise (terms v_i always exist because $\mathcal{A}$ is reduced). We have $h(t) = t_f\{x_1 \leftarrow h(v_1), \ldots, x_n \leftarrow h(v_n)\}$, $h(t) = v\{x'_1 \leftarrow h(u_1), \ldots, x'_m \leftarrow$

$h(u_m)\}$, $h(t) = t'$. Moreover, by definition of the v_i and by induction hypothesis, we have $t \xrightarrow[\mathcal{A}]{*} q$. Note that if q_i occurs more than once, you can substitute q_i by any term satisfying the conditions. The proof does not work for the non linear case because you have to check that different occurrences of some state q_i corresponding to the same variable $x_j \in \mathcal{V}ar(t_f)$ can only be substituted by equal terms.

□

Only linear tree homomorphisms preserve recognizability. But, for linear and non linear homomorphisms, we have:

Theorem 1.4.4 (Inverse homomorphisms preserve recognizability)**.** *Let h be a tree homomorphism and L be a recognizable tree language, then $h^{-1}(L)$ is a recognizable tree language.*

Proof. Let h be a tree homomorphism from $T(\mathcal{F})$ into $T(\mathcal{F}')$ determined by a mapping $h_{\mathcal{F}}$. Let $\mathcal{A}' = (Q', \mathcal{F}', Q'_f, \Delta')$ be a complete DFTA such that $L(\mathcal{A}') = L$. We define a DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ by $Q = Q' \cup \{s\}$ where $s \notin Q'$, $Q_f = Q'_f$ and Δ is defined by the following:

- for $a \in \mathcal{F}_0$, if $t_a \xrightarrow[\mathcal{A}']{*} q$ then $a \rightarrow q \in \Delta$;
- for $f \in \mathcal{F}_n$ where $n > 0$, for $p_1, \ldots, p_n \in Q$, if $t_f\{x_1 \leftarrow p_1, \ldots, x_n \leftarrow p_n\} \xrightarrow[\mathcal{A}']{*} q$ then $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$ where $q_i = p_i$ if x_i occurs in t_f and $q_i = s$ otherwise;
- for $a \in \mathcal{F}_0$, $a \rightarrow s \in \Delta$;
- for $f \in \mathcal{F}_n$ where $n > 0$, $f(s, \ldots, s) \rightarrow s \in \Delta$.

The rule set Δ is computable. The proof of the equivalence $t \xrightarrow[\mathcal{A}]{*} q$ if and only if $h(t) \xrightarrow[\mathcal{A}']{*} q$ is left to the reader. □

It can be proved that the class of recognizable tree languages is the smallest non trivial class of tree languages closed by linear tree homomorphisms and inverse tree homomorphisms. Tree homomorphisms do not in general preserve recognizability, therefore let us consider the following problem: given as instance a recognizable tree language L and a tree homomorphism h, is the set $h(L)$ recognizable? It is not known whether this problem is decidable.

As a conclusion, we consider different special types of tree homomorphisms. These homomorphisms will be used in the next sections in order to simplify some proofs and will be useful in Chapter 6. Let h be a tree homomorphism determined by $h_{\mathcal{F}}$. The tree homomorphism h is said to be:

- ***ϵ-free*** (or non erasing) if for each symbol $f \in \mathcal{F}$, t_f is not reduced to a variable.
- ***symbol to symbol*** if for each symbol $f \in \mathcal{F}$, $\mathcal{H}eight(t_f) = 1$. The reader should note that with our definitions a symbol to symbol tree homomorphism is ϵ-free. A linear symbol to symbol tree homomorphism changes the label of the input symbol, possibly erases some subtrees and possibly modifies order of subtrees.

- ***complete*** if for each symbol $f \in \mathcal{F}_n$, $\mathcal{V}ar(t_f) = \mathcal{X}_n$.
- a ***delabeling*** if h is a complete, linear, symbol to symbol tree homomorphism. Such a delabeling only changes the label of the input symbol and possibly order of subtrees.
- ***alphabetic*** (or a relabeling) if for each symbol $f \in \mathcal{F}_n$, $t_f = g(x_1, \dots, x_n)$, where $g \in \mathcal{F}'_n$.

As a corollary of Theorem 1.4.3, alphabetic tree homomorphisms, delabelings and linear, symbol to symbol tree homomorphisms preserve recognizability. It can be proved that for these classes of tree homomorphisms, given h and a FTA $\mathcal{A}$ such that $L(\mathcal{A}) = L$ as instance, a FTA for the recognizable tree language $h(L)$ can be constructed in linear time. The same holds for $h^{-1}(L)$.

Example 1.4.5. Let $\mathcal{F} = \{f(,), g(), a\}$ and $\mathcal{F}' = \{f'(,), g'(), a'\}$. Let us consider some tree homomorphisms h determined by different $h_{\mathcal{F}}$.

- $h_{\mathcal{F}}(f) = x_1$, $h_{\mathcal{F}}(g) = f'(x_1, x_1)$, and $h_{\mathcal{F}}(a) = a'$ defines a tree homomorphism which is not linear, not ϵ-free, and not complete.
- $h_{\mathcal{F}}(f) = g'(x_1)$, $h_{\mathcal{F}}(g) = f'(x_1, x_1)$, and $h_{\mathcal{F}}(a) = a'$ defines a tree homomorphism which is a non linear symbol to symbol tree homomorphism, and it is not complete.
- $h_{\mathcal{F}}(f) = f'(x_2, x_1)$, $h_{\mathcal{F}}(g) = g'(x_1)$, and $h_{\mathcal{F}}(a) = a'$ defines a tree homomorphism which is a delabeling.
- $h_{\mathcal{F}}(f) = f'(x_1, x_2)$, $h_{\mathcal{F}}(g) = g'(x_1)$, and $h_{\mathcal{F}}(a) = a'$ defines a tree homomorphism which is an alphabetic tree homomorphism.

1.5 Minimizing Tree Automata

In this section, we prove that, like in the word case, there exists a unique minimal automaton in the number of states for a given recognizable tree language.

A Myhill-Nerode Theorem for Tree Languages

The ***Myhill-Nerode Theorem*** is a classical result in the theory of finite automata. This theorem gives a characterization of the recognizable sets and it has numerous applications. A consequence of this theorem, among other consequences, is that there is essentially a unique minimum state DFA for every recognizable language over finite alphabet. The Myhill-Nerode Theorem generalizes in a straightforward way to automata on finite trees.

An equivalence relation $\equiv$ on $T(\mathcal{F})$ is a ***congruence*** on $T(\mathcal{F})$ if for every $f \in \mathcal{F}_n$

$$u_i \equiv v_i \; 1 \leq i \leq n \Rightarrow f(u_1, \dots, u_n) \equiv f(v_1, \dots, v_n) \; .$$

It is of ***finite index*** if there are only finitely many $\equiv$-classes. Equivalently a congruence is an equivalence relation closed under context, i.e. for all contexts

$C \in \mathcal{C}(\mathcal{F})$, if $u \equiv v$, then $C[u] \equiv C[v]$. For a given tree language L, let us define the congruence $\equiv_L$ on $T(\mathcal{F})$ by: $u \equiv_L v$ if for all contexts $C \in \mathcal{C}(\mathcal{F})$,

$$C[u] \in L \text{ iff } C[v] \in L.$$

We are now ready to give the Theorem:

Myhill-Nerode Theorem. *The following three statements are equivalent:*

(i) L is a recognizable tree language

(ii) L is the union of some equivalence classes of a congruence of finite index

(iii) the relation $\equiv_L$ is a congruence of finite index.

Proof.

- *(i) ⇒ (ii)* Assume that L is recognized by some complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$. We consider δ as a transition function. Let us consider the relation $\equiv_{\mathcal{A}}$ defined on $T(\mathcal{F})$ by: $u \equiv_{\mathcal{A}} v$ if $\delta(u) = \delta(v)$. Clearly $\equiv_{\mathcal{A}}$ is a congruence relation and it is of finite index, since the number of equivalence classes is at most the number of states in Q. Furthermore, L is the union of those equivalence classes that include a term u such that $\delta(u)$ is a final state.

- *(ii) ⇒ (iii)* Let us denote by $\sim$ the congruence of finite index. And let us assume that $u \sim v$. By an easy induction on the structure of terms, it can be proved that $C[u] \sim C[v]$ for all contexts $C \in \mathcal{C}(\mathcal{F})$. Now, L is the union of some equivalence classes of $\sim$, thus we have $C[u] \in L$ iff $C[v] \in L$. Thus $u \equiv_L v$, and the equivalence class of u in $\sim$ is contained in the equivalence class of u in $\equiv_L$. Consequently, the index of $\equiv_L$ is lower than or equal to the index of $\sim$ which is finite.

- *(iii) ⇒ (i)* Let Q_{min} be the finite set of equivalence classes of $\equiv_L$. And let us denote by $[u]$ the equivalence class of a term u. Let the transition function δ_{min} be defined by:

  $$\delta_{min}(f, [u_1], \ldots, [u_n]) = [f(u_1, \ldots, u_n)].$$

  The definition of δ_{min} is consistent because $\equiv_L$ is a congruence. And let $Q_{min_f} = \{[u] \mid u \in L\}$. The DFTA $\mathcal{A}_{min} = (Q_{min}, \mathcal{F}, Q_{min_f}, \delta_{min})$ recognizes the tree language L.

□

As a corollary of the Myhill-Nerode Theorem, we can deduce an other algebraic characterization of recognizable tree languages. This characterization is a reformulation of the definition of recognizability. A set of ground terms L is recognizable if and only if there exist a finite $\mathcal{F}$-algebra $\mathcal{A}$, an $\mathcal{F}$-algebra homomorphism $\phi : T(\mathcal{F}) \rightarrow \mathcal{A}$ and a subset A' of the carrier $|\mathcal{A}|$ of $\mathcal{A}$ such that $L = \phi^{-1}(A')$.

Minimization of Tree Automata

First, we prove the existence and uniqueness of the minimum DFTA for a recognizable tree language. It is a consequence of the Myhill-Nerode Theorem because of the following result:

Corollary 1.5.1. *The minimum DFTA recognizing a recognizable tree language L is unique up to a renaming of the states and is given by $\mathcal{A}_{min}$ in the proof of the Myhill-Nerode Theorem.*

Proof. Assume that L is recognized by some DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$. The relation $\equiv_{\mathcal{A}}$ is a refinement of $\equiv_L$ (see the proof of the Myhill-Nerode Theorem). Therefore the number of states of $\mathcal{A}$ is greater than or equal to the number of states of $\mathcal{A}_{min}$. If equality holds, $\mathcal{A}$ is reduced, i.e. all states are accessible, because otherwise a state could be removed leading to a contradiction. Let q be a state in Q and let u be such that $\delta(u) = q$. The state q can be identified with the state $\delta_{min}(u)$. This identification is consistent and defines a one to one correspondence between Q and Q_{min}. □

Second, we give a minimization algorithm for finding the minimum state DFTA equivalent to a given reduced DFTA. We identify an equivalence relation and the sequence of its equivalence classes.

```
Minimization Algorithm MIN
input: complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
begin
    Set $P$ to $\{Q_f, Q - Q_f\}$ /* P is the initial equivalence relation*/
    repeat
        $P' = P$
        /* Refine equivalence $P$ in $P'$ */
        $qP'q'$ if
            $qPq'$ and
            $\forall f \in \mathcal{F}_n \forall q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n \in Q$
            $\delta(f(q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)) P \delta(f(q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n))$
    until $P' = P$
    Set $Q_{min}$ to the set of equivalence classes of $P$
    /* we denote by $[q]$ the equivalence class of state $q$ w.r.t. $P$ */
    Set $\delta_{min}$ to $\{(f, [q_1], \ldots, [q_n]) \rightarrow [f(q_1, \ldots, q_n)]\}$
    Set $Q_{min_f}$ to $\{[q] \mid q \in Q_f\}$
    output: DFTA $\mathcal{A}_{min} = (Q_{min}, \mathcal{F}, Q_{min_f}, \delta_{min})$
end
```

The DFTA constructed by the algorithm $\mathcal{MIN}$ is the minimum state DFTA for its tree language. Indeed, let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ the DFTA to which is applied the algorithm and let $L = L(\mathcal{A})$. Let $\mathcal{A}_{min}$ be the output of the algorithm. It is easy to show that the definition of $\mathcal{A}_{min}$ is consistent and that $L = L(\mathcal{A}_{min})$. Now, by contradiction, we can prove that $\mathcal{A}_{min}$ has no more states than the number of equivalence classes of $\equiv_L$.

1.6 Top Down Tree Automata

The tree automata that we have defined in the previous sections are also known as bottom-up tree automata because these automata start their computation at the leaves of trees. In this section we define top-down tree automata. Such an automaton starts its computation at the root in an initial state and then simultaneously works down the paths of the tree level by level. The tree automaton accepts a tree if a run built up in this fashion can be defined. It appears that top-down tree automata and bottom-up tree automata have the same expressive power. An important difference between bottom-up tree automata and top-down automata appears in the question of determinism since deterministic top-down tree automata are strictly less powerful than nondeterministic ones and therefore are strictly less powerful than bottom-up tree automata. Intuitively, it is due to the following: tree properties specified by deterministic top-down tree automata can depend only on path properties. We now make precise these remarks, but first formally define top-down tree automata.

A nondeterministic ***top-down*** finite Tree Automaton (top-down NFTA) over $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ where Q is a set of states (states are unary symbols), $I \subseteq Q$ is a set of initial states, and Δ is a set of rewrite rules of the following type :

$$q(f(x_1, \ldots, x_n)) \rightarrow f(q_1(x_1), \ldots, q_n(x_n)),$$

where $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in \mathcal{X}$.

When $n = 0$, i.e. when the symbol is a constant symbol a, a transition rule of top-down NFTA is of the form $q(a) \rightarrow a$. A top-down automaton starts at the root and moves downward, associating along a run a state with each subterm inductively. We do not formally define the move relation $\rightarrow_{\mathcal{A}}$ defined by a top-down NFTA because the definition is easily deduced from the corresponding definition for bottom-up NFTA. The tree language $L(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all ground terms t for which there is an initial state q in I such that

$$q(t) \xrightarrow[\mathcal{A}]{*} t.$$

The expressive power of bottom-up and top-down tree automata is the same. Indeed, we have the following Theorem:

Theorem 1.6.1 (The equivalence of top-down and bottom-up NFTAs)**.** *The class of languages accepted by top-down NFTAs is exactly the class of recognizable tree languages.*

Proof. The proof is left to the reader. **Hint.** Reverse the arrows and exchange the sets of initial and final states. □

Top-down and bottom-up tree automata have the same expressive power because they define the same classes of tree languages. Nevertheless they do not have the same behavior from an algorithmic point of view because nondeterminism can not be reduced in the class of top-down tree automata.

Proposition 1.6.2 (Top-down NFTAs and top-down DFTAs)**.** *A top-down finite Tree Automaton* $(Q, \mathcal{F}, I, \Delta)$ *is deterministic (top-down DFTA) if there is one initial state and no two rules with the same left-hand side. Top-down DFTAs are strictly less powerful than top-down NFTAs, i.e. there exists a recognizable tree language which is not accepted by a top-down DFTA.*

Proof. Let $\mathcal{F} = \{f(,), a, b\}$. And let us consider the recognizable tree language $T = \{f(a,b), f(b,a)\}$. Now let us suppose there exists a top-down DFTA that accepts T, the automaton should accept the term $f(a,a)$ leading to a contradiction. Obviously the tree language $T = \{f(a,b), f(b,a)\}$ is recognizable by a finite union of top-down DFTA but there is a recognizable tree language which is not accepted by a finite union of top-down DFTA (see Exercise 1.2). □

The class of languages defined by top-down DFTA corresponds to the class of ***path-closed*** tree languages (see Exercise 1.6).

1.7 Decision Problems and their Complexity

In this section, we study some decision problems and their complexity. The size of an automaton will be the size of its representation. More formally:

Definition 1.7.1. *Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a NFTA over $\mathcal{F}$. The size of a rule $f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow q(f(x_1, \ldots, x_n))$ is $arity(f) + 2$. The size of $\mathcal{A}$ noted $\|A\|$, is defined by:*

$$\|\mathcal{A}\| = |Q| + \sum_{f(q_1(x_1),\ldots,q_n(x_n)) \rightarrow q(f(x_1,\ldots,x_n)) \in \Delta} (arity(f) + 2).$$

We will work in the frame of RAM machines, with uniform measure.

Membership

Instance A ground term.

Answer "yes" if and only if the term is recognized by a given automaton.

Let us first remark that, in our model, for a given deterministic automaton, a run on a tree can be computed in $O(\|t\|)$. The complexity of the problem is:

Theorem 1.7.2. *The membership problem is ALOGTIME-complete.*

Uniform Membership

Instance A tree automaton and a ground term.

Answer "yes" if and only if the term is recognized by the given automaton.

Theorem 1.7.3. *The uniform membership problem can be decided in linear time for DFTA, in polynomial time for NFTA.*

Proof. In the deterministic case, from a term t and the automaton $\|\mathcal{A}\|$, we can compute a run in $O(\|t\| + \|\mathcal{A}\|)$. In the nondeterministic case, the idea is similar to the word case: the algorithm determinizes along the computation, i.e. for each node of the term, we compute the set of reached states. The complexity of this algorithm will be in $O(\|t\| \times \|\mathcal{A}\|)$.

□

The uniform membership problem has been proved LOGSPACE-complete for deterministic top-down tree automata, LOGCFL-complete for NFTA under log-space reductions. For DFTA, it has been proven LOGDCFL, but the precise complexity remains open.

Emptiness

Instance A tree automaton

Answer “yes” if and only if the recognized language is empty.

Theorem 1.7.4. *It can be decided in linear time whether the language accepted by a finite tree automaton is empty.*

Proof. The minimal height of accepted terms can be bounded by the number of states using Corollary 1.2.3; so, as membership is decidable, emptiness is decidable. Of course, this approach does not provide a practicable algorithm. To get an efficient algorithm, it suffices to notice that a NFTA accepts at least one tree if and only if there is an accessible final state. In other words, the language recognized by a **reduced** automaton is empty if and only if the set of final states is non empty. Reducing an automaton can be done in $O(|Q| \times \|\mathcal{A}\|)$ by the reduction algorithm given in Section 1.1. Actually, this algorithm can be improved by choosing an adequate data structure in order to get a linear algorithm (see Exercise 1.18). This linear least fixpoint computation holds in several frameworks. For example, it can be viewed as the satisfiability test of a set of propositional Horn formulae. The reduction is easy and linear: each state q can be associated with a propositional variable X_q and each rule $r : f(q_1, \dots, q_n) \rightarrow q$ can be associated with a propositional Horn formula $F_r = X_q \vee \neg X_{q_1} \vee \dots \vee \neg X_{q_n}$. It is straightforward that satisfiability of $\{F_r\} \cup \{\neg X_q / q \in Q_f\}$ is equivalent to emptiness of the language recognized by $(Q, \mathcal{F}, Q_f, \Delta)$. So, as satisfiability of a set of propositional Horn formulae can be decided in linear time, we get a linear algorithm for testing emptiness for NFTA. ☐

The emptiness problem is **P-complete** with respect to logspace reductions, even when restricted to deterministic tree automata. The proof can easily be done since the problem is very close to *the solvable path systems* problem which is known to be P-complete (see Exercise 1.19).

Intersection non-emptiness

Instance A finite sequence of tree automata.

Answer “yes” if and only if there is at least one term recognized by each automaton of the sequence.

Theorem 1.7.5. *The intersection problem for tree automata is EXPTIME-complete.*

Proof. By constructing the product automata for the n automata, and then testing non-emptiness, we get an algorithm in $O(\|\mathcal{A}_1\| \times \dots \times \|\mathcal{A}_n\|)$. The proof of EXPTIME-hardness is based on simulation of an alternating linear space-bounded Turing machine. Roughly speaking, with such a machine and an input of length n can be associated polynomially n tree automata whose intersection corresponds to the set of accepting computations on the input. It is worth noting that the result holds for deterministic top down tree automata as well as for deterministic bottom-up ones. ☐

Finiteness

Instance A tree automaton

Answer “yes” if and only if the recognized language is finite.

Theorem 1.7.6. *Finiteness can be decided in polynomial time.*

Proof. Let us consider a NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$. Deciding finiteness of $\mathcal{A}$ is direct by Corollary 1.2.3: it suffices to find an accepted term t s.t. $|Q| < \|t\| \leq 2 * |Q|$. A more efficient way to test finiteness is to check the existence of a loop: the language is infinite if and only if there is a loop on some useful state, i.e. there exist an accessible state q and contexts C and C' such that $C[q] \xrightarrow[\mathcal{A}]{*} q$ and $C'[q] \xrightarrow[\mathcal{A}]{*} q'$ for some final state q'. Computing accessible and coaccessible states can be done in $O(|Q| \times \|\mathcal{A}\|)$ or in $O(\|\mathcal{A}\|)$ by using an ad hoc representation of the automaton. For a given q, deciding if there is a loop on q can be done in $O(\|\mathcal{A}\|)$. So, finiteness can be decided in $O(|Q| \times \|\mathcal{A}\|)$. ☐

Emptiness of the Complement

Instance A tree automaton.

Answer “yes” if and only if every term is accepted by the automaton

Deciding whether a deterministic tree automaton recognizes the set of all terms is polynomial for a fixed alphabet: we just have to check whether the automaton is complete (which can be done in $O(|\mathcal{F}| \times |Q|^{Arity(\mathcal{F})})$) and then it remains only to check that all accessible states are final. For nondeterministic automata, the following result proves in some sense that determinization with its exponential cost is unavoidable:

Theorem 1.7.7. *The problem whether a tree automaton accepts the set of all terms is EXPTIME-complete for nondeterministic tree automata.*

Proof. The proof of this theorem is once more based on simulation of a linear space bounded alternating Turing machine: indeed, the complement of the accepting computations on an input w can be coded polynomially in a recognizable tree language. ☐

Equivalence

Instance Two tree automata

Answer “yes” if and only if the automata recognize the same language.

Theorem 1.7.8. *Equivalence is decidable for tree automata.*

Proof. Clearly, as the class of recognizable sets is effectively closed under complementation and intersection, and as emptiness is decidable, equivalence is decidable. For two deterministic complete automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we get by these means an algorithm in $O(\|\mathcal{A}_1\| \times \|\mathcal{A}_2\|)$. (Another way is to compare the minimal automata). For nondeterministic ones, this approach leads to an exponential algorithm. ☐

As we have proved that deciding whether an automaton recognizes the set of all ground terms is EXPTIME-hard, we get immediately:

Corollary 1.7.9. *The inclusion problem and the equivalence problem for NFTAs are EXPTIME-complete.*

Singleton Set Property

Instance A tree automaton

Answer "yes" if and only if the recognized language is a singleton set.

Theorem 1.7.10. *The singleton set property is decidable in polynomial time.*

Proof. There are several ways to get a polynomial algorithm for this property. A first one would be to first check non-emptiness of $L(\mathcal{A})$ and then "extract" from $\mathcal{A}$ a DFA $\mathcal{B}$ whose size is smaller than $\|\mathcal{A}\|$ and which accepts a single term recognized by $\mathcal{A}$. Then it remains to check emptiness of $L(\mathcal{A}) \cap \overline{L(\mathcal{B})}$. This can be done in polynomial time, even if $\mathcal{B}$ is non complete.

Another way is: for each state of a bottom-up tree automaton $\mathcal{A}$, compute, up to 2, the number $C(q)$ of terms leading to state q. This can be done in a straightforward way when $\mathcal{A}$ is deterministic; when $\mathcal{A}$ is non deterministic, this can be also done in polynomial time:

```
Singleton Set Test Algorithm
input: NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
begin
  Set $C(q)$ to 0, for every $q$ in $Q$
  /* $C(q) \in \{0,1,2\}$ is the number, up to 2, of terms leading to state $q$ */
  /* if $C(q) = 1$ then $T(q)$ is a representation of the accepted tree */
  repeat
    for each rule $f(q_1, \ldots, q_n) \rightarrow q \in \Delta$ do
      Case $\wedge_j C(q_j) >= 1$ and $C(q_i) = 2$ for some i: Set $C(q)$ to 2
      Case $\wedge_j C(q_j) = 1$ and $C(q) = 0$: Set $C(q)$ to 1, $T(q)$ to $f(q_1, ...q_n)$
      Case $\wedge_j C(q_j) = 1$, $C(q) = 1$ and $Diff(T(q), f(q_1, \ldots, q_n))$:
        Set $C(q)$ to 2
      Others null
      where $Diff(f(q_1, ..., q_n), g(q'_1, ..., q'_n))$ defined by:
        /* Diff can be computed polynomially by using memorization. */
        if $(f \neq g)$ then return true
        elseif $Diff(T(q_i), T(q'_i)$ for some $q_i$ then return True
        else return False
  until $C$ can not be changed
  output:
  /*$L(\mathcal{A})$ is empty */
  if $\wedge_{q \in Q_f} C(q) = 0$ then return False
  /* two terms in $L(\mathcal{A})$ accepted in the same state or two different states */
  elseif $\exists q \in Q_f\ C(q) = 2$ then return False
  elseif $\exists q, q' \in Q_f\ C(q) = C(q') = 1$ and $Diff(T(q), T(q'))$ then return False
  /* in all other cases $L(\mathcal{A})$ is a singleton set*/
```

```
        else return True.
    end
```

□

Other complexity results for "classical" problems can be found in the exercises. E.g., let us cite the following problem whose proof is sketched in Exercise 1.12

Ground Instance Intersection Problem

Instance A term t, a tree automaton $\mathcal{A}$.

Answer "yes" if and only if there is at least a ground instance of t which is accepted by $\mathcal{A}$.

Theorem 1.7.11. *The Ground Instance Intersection Problem for tree automata is P when t is linear, NP-complete when t is non linear and $\mathcal{A}$ deterministic, EXPTIME-complete when t is non linear and $\mathcal{A}$ non deterministic.*

1.8 Exercises

Starred exercises are discussed in the bibliographic notes.

Exercise 1.1. Let $\mathcal{F} = \{f(,), g(), a\}$. Define a top-down NFTA, a NFTA and a DFTA for the set $G(t)$ of ground instances of term $t = f(f(a, x), g(y))$ which is defined by $G(t) = \{f(f(a, u), g(v)) \mid u, v \in T(\mathcal{F})\}$. Is it possible to define a top-down DFTA for this language?

Exercise 1.2. Let $\mathcal{F} = \{f(,), g(), a\}$. Define a top-down NFTA, a NFTA and a DFTA for the set $M(t)$ of terms which have a ground instance of term $t = f(a, g(x))$ as a subterm, that is $M(t) = \{C[f(a, g(u))] \mid C \in \mathcal{C}(\mathcal{F}),\ u \in T(\mathcal{F})\}$. Is it possible to define a top-down DFTA for this language? A finite union of top-down DFTA ?

Exercise 1.3. Let $\mathcal{F} = \{g(), a\}$. Is the set of ground terms whose height is even recognizable? Let $\mathcal{F} = \{f(,), g(), a\}$. Is the set of ground terms whose height is even recognizable?

Exercise 1.4. Let $\mathcal{F} = \{f(,), a\}$. Prove that the set $L = \{f(t, t) \mid t \in T(\mathcal{F})\}$ is not recognizable. Let $\mathcal{F}$ be any ranked alphabet which contains at least one constant symbol a and one binary symbol $f(,)$. Prove that the set $L = \{f(t, t) \mid t \in T(\mathcal{F})\}$ is not recognizable.

Exercise 1.5. Prove the equivalence between top-down NFTA and NFTA.

Exercise 1.6. Let t be a ground term, the path language $\pi(t)$ is defined inductively by:

- if $t \in \mathcal{F}_0$, then $\pi(t) = t$
- if $t = f(t_1, \ldots, t_n)$, then $\pi(t) = \bigcup_{i=1}^{i=n} \{fiw \mid w \in \pi(t_i)\}$

Let L be a tree language, the path language of L is defined as $\pi(L) = \cup_{t \in L} \pi(t)$, the path closure of L is defined as $pathclosure(L) = \{t \mid \pi(t) \subseteq \pi(L)\}$. A tree language L is path-closed if $pathclosure(L) = L$.

1. Prove that if L is a recognizable tree language, then $\pi(L)$ is a recognizable language
2. Prove that if L is a recognizable tree language, then $\mathit{pathclosure}(L)$ is a recognizable language
3. Prove that a recognizable tree language is path-closed if and only if it is recognizable by a top-down DFTA.
4. Is it decidable whether a recognizable tree language defined by a NFTA is path-closed ?

Exercise 1.7. Let $\mathcal{F} = \{f(,), g(), a\}$ and $\mathcal{F}' = \{f'(,), g(), a\}$. Let us consider the tree homomorphism h determined by $h_{\mathcal{F}}$ defined by: $h_{\mathcal{F}}(f) = f'(x_1, x_2)$, $h_{\mathcal{F}}(g) = f'(x_1, x_1)$, and $h_{\mathcal{F}}(a) = a$. Is $h(T(\mathcal{F}))$ recognizable? Let $L_1 = \{g^i(a) \mid i \geq 0\}$, then L_1 is a recognizable tree language, is $h(L_1)$ recognizable? Let L_2 be the recognizable tree language defined by $L_2 = L(\mathcal{A})$ where $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ is defined by: $Q = \{q_a, q_g, q_f\}$, $Q_f = \{q_f\}$, and Δ is the following set of transition rules:

$$\begin{array}{lrclrcl}
\{ & a & \rightarrow & q_a & g(q_a) & \rightarrow & q_g \\
 & f(q_a, q_a) & \rightarrow & q_f & f(q_g, q_g) & \rightarrow & q_f \\
 & f(q_a, q_g) & \rightarrow & q_f & f(q_g, q_a) & \rightarrow & q_f \\
 & f(q_a, q_f) & \rightarrow & q_f & f(q_f, q_a) & \rightarrow & q_f \\
 & f(q_g, q_f) & \rightarrow & q_f & f(q_f, q_g) & \rightarrow & q_f \\
 & f(q_f, q_f) & \rightarrow & q_f \quad \}. & & &
\end{array}$$

Is $h(L_2)$ recognizable?

Exercise 1.8. Let $\mathcal{F}_1 = \{\mathit{or}(,), \mathit{and}(,), \mathit{not}(), 0, 1, x\}$. A ground term over $\mathcal{F}$ can be viewed as a boolean formula over variable x. Define a DFTA which recognizes the set of satisfiable boolean formulae over x. Let $\mathcal{F}_n = \{\mathit{or}(,), \mathit{and}(,), \mathit{not}(), 0, 1, x_1, \ldots, x_n\}$. A ground term over $\mathcal{F}$ can be viewed as a boolean formula over variables $x_1, \ldots, x_n$. Define a DFTA which recognizes the set of satisfiable boolean formulae over $x_1, \ldots, x_n$.

Exercise 1.9. Let t be a linear term in $T(\mathcal{F}, \mathcal{X})$. Prove that the set $G(t)$ of ground instances of term t is recognizable. Let R be a finite set of linear terms in $T(\mathcal{F}, \mathcal{X})$. Prove that the set $G(R)$ of ground instances of set R is recognizable.

Exercise 1.10. * Let R be a finite set of linear terms in $T(\mathcal{F}, \mathcal{X})$. We define the set $\mathit{Red}(R)$ of reducible terms for R to be the set of ground terms which have a ground instance of some term in R as a subterm.

1. Prove that the set $\mathit{Red}(R)$ is recognizable.
2. Prove that the number of states of a DFTA recognizing $\mathit{Red}(R)$ can be at least 2^{n-1} where n is the size (number of nodes) of R. Hint: Consider the set reduced to the pattern $h(f(x_1, f(x_2, f(x_3), \ldots, (f(x_{p-1}, f(a, x_p) \cdots)$.
3. Let us now suppose that R is a finite set of **ground** terms. Prove that we can construct a DFTA recognizing $\mathit{Red}(R)$ whose number of states is at most $n+2$ where n is the number of different subterms of R.

Exercise 1.11. * Let R be a finite set of linear terms in $T(\mathcal{F}, \mathcal{X})$. A term t is inductively reducible for R if all the ground instances of term t are reducible for R. Prove that inductive reducibility of a linear term t for a set of linear terms R is decidable.

Exercise 1.12. *

We consider the following decision problem:

Instance t a term in $T(\mathcal{F}, \mathcal{X})$ and $\mathcal{A}$ a NFTA

Answer "yes" if and only if at least one ground instance of t is accepted by A.

1. Let us first suppose that t is linear; prove that the property is P.

   Hint: a NFTA for the set of ground instances of t can ce computed polynomially (see Exercise 1.9

2. Let us now suppose that t is non linear but that $\mathcal{A}$ is deterministic.

   (a) Prove that the property is NP. Hint: we just have to guess a substitution of the variables of t by states.

   (b) Prove that the property is NP-hard.

   Hint: just consider a term t which represents a boolean formula and $\mathcal{A}$ a DFTA which accepts valid formulas.

3. Let us now suppose that t is non linear and that $\mathcal{A}$ is non deterministic.

   Prove that the property is $EXPTIME-$complete.

   Hint: use the EXPTIME-hardness of intersection non-emptiness.

Exercise 1.13. * We consider the following two problems. First, given as instance a recognizable tree language L and a tree homomorphism h, is the set $h(L)$ recognizable? Second, given as instance a set R of terms in $T(\mathcal{F}, \mathcal{X})$, is the set $Red(R)$ recognizable? Prove that if the first problem is decidable, the second problem is easily shown decidable.

Exercise 1.14. Let $\mathcal{F} = \{f(,), a, b\}$.

1. Let us consider the set of ground terms L_1 defined by the following two conditions:

   - $f(a, b) \in L_1$,
   - $t \in L_1 \Rightarrow f(a, f(t, b)) \in L_1$.

   Prove that the set L_1 is recognizable.

2. Prove that the set $L_2 = \{t \in T(\mathcal{F}) \mid |t|_a = |t|_b\}$ is not recognizable where $|t|_a$ (respectively $|t|_b$) denotes the number of a (respectively the number of b) in t.

3. Let L be a recognizable tree language over $\mathcal{F}$. Let us suppose that f is a commutative symbol. Let $C(L)$ be the congruence closure of set L for the set of equations $C = \{f(x, y) = f(y, x)\}$. Prove that $C(L)$ is recognizable.

4. Let L be a recognizable tree language over $\mathcal{F}$. Let us suppose that f is a commutative and associative symbol. Let $AC(L)$ be the congruence closure of set L for the set of equations $AC = \{f(x, y) = f(y, x); f(x, f(y, z)) = f(f(x, y), z)\}$. Prove that in general $AC(L)$ is not recognizable.

5. Let L be a recognizable tree language over $\mathcal{F}$. Let us suppose that f is an associative symbol. Let $A(L)$ be the congruence closure of set L for the set of equations $A = \{f(x, f(y, z)) = f(f(x, y), z)\}$. Prove that in general $A(L)$ is not recognizable.

Exercise 1.15. * Consider the *complement problem*:

- **Instance** A term $t \in T(\mathcal{F}, \mathcal{X})$ and terms $t_1, \dots, t_n$,
- **Question** There is a ground instance of t which is not an instance of any t_i.

Prove that the complement problem is decidable whenever term t and all terms t_i are linear. Extend the proof to handle the case where t is a term (not necessarily linear).

Exercise 1.16. * Let $\mathcal{F}$ be a ranked alphabet and suppose that $\mathcal{F}$ contains some symbols which are commutative and associative. The set of ground AC-instances of a term t is the AC-congruence closure of set $G(t)$. Prove that the set of ground AC-instances of a linear term is recognizable. The reader should note that the set of ground AC-instances of a set of linear terms is not recognizable (see Exercice 1.14).

Prove that the *AC-complement problem* is decidable where the AC-complement problem is defined by:

- **Instance** A linear term $t \in T(\mathcal{F}, \mathcal{X})$ and linear terms $t_1, \ldots, t_n$,
- **Question** There is a ground AC-instance of t which is not an AC-instance of any t_i.

Exercise 1.17. * Let $\mathcal{F}$ be a ranked alphabet and $\mathcal{X}$ be a countable set of variables. Let S be a rewrite system on $T(\mathcal{F}, \mathcal{X})$ (the reader is referred to [DJ90]) and L be a set of ground terms. We denote by $S^*(L)$ the set of reductions of terms in L by S and by $S(L)$ the set of ground S-normal forms of set L. Formally,

$$S^*(L) = \{t \in T(\mathcal{F}) \mid \exists u \in L \; u \xrightarrow{*} t\},$$

$$S(L) = \{t \in T(\mathcal{F}) \mid t \in IRR(S) \text{ and } \exists u \in L \; u \xrightarrow{*} t\} = IRR(S) \cap S^*(L)$$

where $IRR(S)$ denotes the set of ground irreducible terms for S. We consider the two following decision problems:

(1rst order reachability)

- **Instance** A rewrite system S, two ground terms u and v,
- **Question** $v \in S^*(\{u\})$.

(2nd order reachability)

- **Instance** A rewrite system S, two recognizable tree languages L and L',
- **Question** $S^*(L) \subseteq L'$.

1. Let us suppose that rewrite system S satisfies:

   (PreservRec) If L is recognizable, then $S^*(L)$ is recognizable.

   What can be said about the two reachability decision problems? Give a sufficient condition on rewrite system S satisfying (PreservRec) such that S satisfies (NormalFormRec) where (NormalFormRec) is defined by:

   (NormalFormRec) If L is recognizable, then $S(L)$ is recognizable.

2. Let $\mathcal{F} = \{f(,), g(), h(), a\}$. Let $L = \{f(t_1, t_2) \mid t_1, t_2 \in T(\{g(), h(), a\}\}$, and S is the following set of rewrite rules:

$$\{ \begin{array}{rcl@{\qquad}rcl} f(g(x), h(y)) & \to & f(x, y) & f(h(x), g(y)) & \to & f(x, y) \\ g(h(x)) & \to & x & h(g(x)) & \to & x \\ f(a, x) & \to & x & f(x, a) & \to & x \end{array} \}$$

   Are the sets L, $S^*(L)$, and $S(L)$ recognizable?

3. Let $\mathcal{F} = \{f(,), g(), h(), a\}$. Let $L = \{g(h^n(a)) \mid n \geq 0\}$, and S is the following set of rewrite rules:

$$\{ \; g(x) \to f(x, x) \; \}$$

   Are the sets L, $S^*(L)$, and $S(L)$ recognizable?

4. Let us suppose now that rewrite system S is linear and monadic, i.e. all rewrite rules are of one of the following three types:

$$\begin{array}{llll} (1) & l & \rightarrow & a & , a \in \mathcal{F}_0 \\ (2) & l & \rightarrow & x & , x \in \mathcal{V}ar(l) \\ (3) & l & \rightarrow & f(x_1, \ldots, x_p) & , x_1, \ldots, x_p \in \mathcal{V}ar(l), f \in \mathcal{F}_p \end{array}$$

   where l is a linear term (no variable occurs more than once in t) whose height is greater than 1. Prove that a linear and monadic rewrite system satisfies (PreservRec). Prove that (PreservRec) is false if the right-hand side of rules of type (3) may be non linear.

Exercise 1.18. Design a linear-time algorithm for testing emptiness of the language recognized by a tree automaton:

Instance A tree automaton

Answer "yes" if and only if the language recognized is empty.

Hint: Choose a suitable data structure for the automaton. For example, a state could be associated with the list of the "adresses" of the rules whose left-hand side contain it (eventually, a rule can be repeated); each rule could be just represented by a counter initialized at the arity of the corresponding symbol and by the state of the right-hand side. Activating a state will decrement the counters of the corresponding rules. When the counter of a rule becomes null, the rule can be applied: the right-hand side state can be activated.

Exercise 1.19. The Solvable Path Problem is the following:

Instance a finite set X and three sets $R \subset X \times X \times X, X_s \subset X$ and $X_t \subset X$.

Answer "yes" if and only if $X_t \cap A$ is non empty, where A is the least subset of X such that $X_s \subset A$ and if $y, z \in A$ and $(x, y, z) \in R$, then $x \in A$.

Prove that this $P - complete$ problem is log-space reducible to the emptiness problem for tree automata.

Exercise 1.20. A *flat automaton* is a tree automaton which has the following property: there is an ordering $\geq$ on the states and a particular state $q_\top$ such that the transition rules have one of the following forms:

1. $f(q_\top, \ldots, q_\top) \rightarrow q_\top$
2. $f(q_1, \ldots, q_n) \rightarrow q$ with $q > q_i$ for every i
3. $f(q_\top, \ldots, q_\top, q, q_\top, \ldots, q_\top) \rightarrow q$

Moreover, we assume that all terms are accepted in the state $q_\top$. (The automaton is called *flat* because there are no "nested loop").

Prove that the intersection of two flat automata is a finite union of automata whose size is linear in the sum of the original automata. (This contrasts with the construction of Theorem 1.3.1 in which the intersection automaton's size is the product of the sizes of its components).

Deduce from the above result that the intersection non-emptiness problem for flat automata is in NP (compare with Theorem 1.7.5).

1.9 Bibliographic Notes

Tree automata were introduced by Doner [Don65, Don70] and Thatcher and Wright [TW65, TW68]. Their goal was to prove the decidability of the weak second order theory of multiple successors. The original definitions are based on the algebraic approach and involve heavy use of universal algebra and/or category theory.

Many of the basic results presented in this chapter are the straightforward generalization of the corresponding results for finite automata. It is difficult to attribute a particular result to any one paper. Thus, we only give a list of some important contributions consisting of the above mentioned papers of Doner, Thatcher and Wright and also Eilenberg and Wright [EW67], Thatcher [Tha70], Brainerd [Bra68, Bra69], Arbib and Give'on [AG68]. All the results of this chapter and a more complete and detailed list of references can be found in the textbook of Gécseg and Steinby [GS84] and their survey [GS96]. For an overview of the notion of recognizability in general algebraic structures see Courcelle [Cou89] and the fundamental paper of Mezei and Wright [MW67]. In Nivat and Podelski [NP89] and [Pod92], the theory of recognizable tree languages is reduced to the theory of recognizable sets in an infinitely generated free monoid.

The results of Sections 1.1, 1.2, and 1.3 were noted in many of the papers mentioned above, but, in this textbook, we present these results in the style of the undergraduate textbook on finite automata by Hopcroft and Ullman [HU79]. Tree homomorphisms were defined as a special case of tree transducers, see Thatcher [Tha73]. The reader is referred to the bibliographic notes in Chapter 6 of the present textbook for detailed references. The reader should note that our proof of preservation of recognizability by tree homomorphisms and inverse tree homomorphisms is a direct construction using FTA. A more classical proof can be found in [GS84] and uses regular tree grammars (see Chapter 2).

Minimal tree recognizers and Nerode's congruence appear in Brainerd [Bra68, Bra69], Arbib and Give'on [AG68], and Eilenberg and Wright [EW67]. The proof we presented here is by Kozen [Koz92] (see also Fülöp and Vágvölgyi [FV89]). Top-down tree automata were first defined by Rabin [Rab69]. Deterministic top-down tree automata define the class of path-closed tree languages (see Exercice 1.6). An alternative definition of deterministic top-down tree automata was given in [NP97] leading to "homogeneous" tree languages, also a minimization algorithm was given. Residual finite automata were defined in [CGL$^+$03] via bottom-up or top-down congruence closure. Residual finite automata have a canonical form even in the non deterministic case.

Some results of Sections 1.7 are "folklore" results. Complexity results for the membership problem and the uniform membership problem could be found in [Loh01]. Other interesting complexity results for tree automata can be found in Seidl [Sei89], [Sei90]. The EXPTIME-hardness of the problem of intersection non-emptiness is often used; this problem is close to problems of type inference and an idea of the proof can be found in [FSVY91]. A proof for deterministic top-down automata can be found in [Sei94b]. A detailed proof in the deterministic bottom-up case as well as some other complexity results are in [Vea97a], [Vea97b].

Numerous exercises of the present chapter illustrate applications of tree automata theory to automated deduction and to the theory of rewriting systems. These applications are studied in more details in Section 3.4. Results about tree automata and rewrite systems are collected in Gilleron and Tison [GT95]. Let S be a term rewrite system (see for example Dershowitz and Jouannaud [DJ90] for a survey on rewrite systems), if S is left-linear the set $IRR(S)$ of irreducible ground terms w.r.t. S is a recognizable tree language. This result first appears in Gallier and Book [GB85] and is the subject of Exercise 1.10. However not every recognizable tree language is the set of irreducible terms w.r.t. a rewrite system S (see Fülöp and Vágvölgyi [FV88]). It was proved that the problem whether, given a rewrite system S as instance, the set of irreducible terms is recognizable is decidable (Kucherov [Kuc91]). The problem of preservation of regularity by tree homomorphisms is not known decidable. Exercise 1.13 shows connections between preservation of regularity for tree homomorphisms and recognizability of sets of irreducible terms for rewrite systems.

The notion of inductive reducibility (or ground reducibility) was introduced in automated deduction. A term t is S-inductively (or S-ground) reducible for S if all the ground instances of term t are reducible for S. Inductive reducibility is decidable for a linear term t and a left-linear rewrite system S. This is Exercise 1.11, see also Section 3.4.2. Inductive reducibility is decidable for finite S (see Plaisted [Pla85]). Complement problems are also introduced in automated deduction. They are the subject of Exercises 1.15 and 1.16. The complement problem for linear terms was proved decidable by Lassez and Marriott [LM87] and the AC-complement problem by Lugiez and Moysset [LM94].

The reachability problem is defined in Exercise 1.17. It is well known that this problem is undecidable in general. It is decidable for rewrite systems preserving recognizability, i.e. such that for every recognizable tree language L, the set of reductions of terms in L by S is recognizable. This is true for linear and monadic rewrite systems (right-hand sides have depth less than 1). This result was obtained by K. Salomaa [Sal88] and is the matter of Exercise 1.17. This is true also for linear and semi-monadic (variables in the right-hand sides have depth at most 1) rewrite systems, Coquidé et al. [CDGV94]. Other interesting results can be found in [Jac96] and [NT99].

Chapter 2

Regular Grammars and Regular Expressions

2.1 Tree Grammar

In the previous chapter, we have studied tree languages from the acceptor point of view, using tree automata and defining recognizable languages. In this chapter we study languages from the generative point of view, using regular tree grammars and defining regular tree languages. We shall see that the two notions are equivalent and that many properties and concepts on regular word languages smoothly generalize to regular tree languages, and that algebraic characterizations of regular languages do exist for tree languages. Actually, this is not surprising since tree languages can be seen as word languages on an infinite alphabet of contexts.

2.1.1 Definitions

When we write programs, we often have to know how to produce the elements of the data structures that we use. For instance, a definition of the lists of integers in a functional language like ML is similar to the following definition:

$$\begin{array}{l} Nat = 0 \mid s(Nat) \\ List = nil \mid cons(Nat, List) \end{array}$$

This definition is nothing but a tree grammar in disguise, more precisely the set of lists of integers is the tree language generated by the grammar with axiom $List$, non-terminal symbols $List, Nat$, terminal symbols $0, s, nil, cons$ and rules

$$\begin{array}{lcl} Nat & \rightarrow & 0 \\ Nat & \rightarrow & s(Nat) \\ List & \rightarrow & nil \\ List & \rightarrow & cons(Nat, List) \end{array}$$

Tree grammars are similar to word grammars except that basic objects are trees, therefore terminals and non-terminals may have an arity greater than 0. More precisely, a ***tree grammar*** $G = (S, N, \mathcal{F}, R)$ is composed of an ***axiom*** S, a set N of ***non-terminal*** symbols with $S \in N$, a set $\mathcal{F}$ of ***terminal*** symbols, a set R of ***production rules*** of the form $\alpha \rightarrow \beta$ where α, β are trees of $T(\mathcal{F} \cup N \cup \mathcal{X})$ where $\mathcal{X}$ is a set of dummy variables and α contains at least one non-terminal. Moreover we require that $\mathcal{F} \cap N = \emptyset$, that each element of $N \cup \mathcal{F}$ has a fixed arity and that the arity of the axiom S is 0. In this chapter, we shall concentrate on ***regular tree grammars*** where a regular tree grammar $G = (S, N, \mathcal{F}, R)$ is a tree grammar such that all non-terminal symbols have arity 0 and production rules have the form $A \rightarrow \beta$, with A a non-terminal of N and β a tree of $T(\mathcal{F} \cup N)$.

Example 2.1.1. The grammar G with axiom $List$, non-terminals $List, Nat$ terminals $0, nil, s(), cons(,)$, rules

$$\begin{array}{l} List \rightarrow nil \\ List \rightarrow cons(Nat, List) \\ Nat \rightarrow 0 \\ Nat \rightarrow s(Nat) \end{array}$$

is a regular tree grammar.

A tree grammar is used to build terms from the axiom, using the corresponding ***derivation relation***. Basically the idea is to replace a non-terminal A by the right-hand side α of a rule $A \rightarrow \alpha$. More precisely, given a regular tree grammar $G = (S, N, \mathcal{F}, R)$, the derivation relation $\rightarrow_G$ associated to G is a relation on pairs of terms of $T(\mathcal{F} \cup N)$ such that $s \rightarrow_G t$ if and only if there are a rule $A \rightarrow \alpha \in R$ and a context C such that $s = C[A]$ and $t = C[\alpha]$. The ***language generated*** by G, denoted by $L(G)$, is the set of terms of $T(\mathcal{F})$ which can be reached by successive derivations starting from the axiom, i.e. $L(G) = \{s \in T_{\mathcal{F}} \mid S \xrightarrow{+}_G s\}$ with $\xrightarrow{+}$ the transitive closure of $\rightarrow_G$. We write $\rightarrow$ instead of $\rightarrow_G$ when the grammar G is clear from the context. A ***regular tree language*** is a language generated by a regular tree grammar.

Example 2.1.2. Let G be the grammar of the previous example, then a derivation of $cons(s(0), nil)$ from $List$ is

$$\begin{array}{ll} List \rightarrow_G cons(Nat, List) \rightarrow_G cons(s(Nat), List) & \rightarrow_G cons(s(Nat), nil) \\ & \rightarrow_G cons(s(0), nil) \end{array}$$

and the language generated by G is the set of lists of non-negative integers.

From the example, we can see that trees are generated top-down by replacing a leaf by some other term. When A is a non-terminal of a regular tree grammar G, we denote by $L_G(A)$ the language generated by the grammar G' identical to G but with A as axiom. When there is no ambiguity on the grammar referred to, we drop the subscript G. We say that two grammars G and G' are ***equivalent*** when they generate the same language. Grammars can contain useless rules or non-terminals and we want to get rid of these while preserving the generated language. A non-terminal is ***reachable*** if there is a derivation from the axiom containing this non-terminal. A non-terminal A is ***productive*** if $L_G(A)$ is non-empty. A regular tree grammar is ***reduced*** if and only if all its non-terminals are reachable and productive. We have the following result:

Proposition 2.1.3. *Each regular tree grammar is equivalent to a reduced regular tree grammar.*

Proof. Given a grammar $G = (S, N, \mathcal{F}, R)$, we can compute the set of reachable non-terminals and the set of productive non-terminals using the sequences $(Reach)_n$ and $(Prod)_n$ which are defined in the following way.

$$\begin{aligned}
Prod_0 &= \emptyset \\
Prod_n &= Prod_{n-1} \cup \{A \in N \mid \exists (A \rightarrow \alpha) \in R \text{ s.t.} \\
&\qquad\qquad \text{each non-terminal of } \alpha \text{ is in } Prod_{n-1}\} \\
Reach_0 &= \{S\} \\
Reach_n &= Reach_{n-1} \cup \{A \in N \mid \exists (A' \rightarrow \alpha) \in R \text{ s.t.} \\
&\qquad\qquad A' \in Reach_{n-1} \text{ and } A \text{ occurs in } \alpha\}
\end{aligned}$$

For each sequence, there is an index such that all elements of the sequence with greater index are identical and this element is the set of productive (resp. reachable) non-terminals of G. Each regular tree grammar is equivalent to a reduced tree grammar which is computed by the following cleaning algorithm.

Computation of an equivalent reduced grammar
input: a regular tree grammar $G = (S, N, \mathcal{F}, R)$.

1. Compute the set of productive non-terminals $N_{Prod} = \bigcup_{n \geq 0} Prod_n$ for G and let $G' = (S, N_{Prod}, \mathcal{F}, R')$ where R' is the subset of R involving rules containing only productive non-terminals.
2. Compute the set of reachable non-terminals $N_{Reach} = \bigcup_{n \geq 0} Reach_n$ for G' (not G) and let $G'' = (S, N_{Reach}, \mathcal{F}, R'')$ where R'' is the subset of R' involving rules containing only reachable non-terminals.

output: G''

The equivalence of G, G' and G'' is left to the reader. Moreover each non-terminal A of G'' must appear in a derivation $S \xrightarrow{*}_{G''} C[A] \xrightarrow{*}_{G''} C[s]$ which proves that G'' is reduced. The reader should notice that exchanging the two steps of the computation may result in a grammar which is not reduced (see Exercise 2.3).

□

Actually, we shall use even simpler grammars, i.e. ***normalized*** regular tree grammar, where the production rules have the form $A \rightarrow f(A_1, \ldots, A_n)$ or $A \rightarrow a$ where f, a are symbols of $\mathcal{F}$ and $A, A_1, \ldots, A_n$ are non-terminals. The following result shows that this is not a restriction.

Proposition 2.1.4. *Each regular tree grammar is equivalent to a normalized regular tree grammar.*

Proof. Replace a rule $A \rightarrow f(s_1, \ldots, s_n)$ by $A \rightarrow f(A_1, \ldots, A_n)$ with $A_i = s_i$ if $s_i \in N$ otherwise A_i is a new non-terminal. In the last case add the rule $A_i \rightarrow s_i$. Iterate this process until one gets a (necessarily equivalent) grammar with rules of the form $A \to f(A_1, \dots, A_n)$ or $A \to a$ or $A_1 \to A_2$. The last rules are replaced by the rules $A_1 \to \alpha$ for all $\alpha \notin N$ such that $A_1 \xrightarrow{+} A_i$ and $A_i \to \alpha \in R$ (these $A_i's$ are easily computed using a transitive closure algorithm). □

From now on, we assume that all grammars are normalized, unless this is stated otherwise explicitly.

2.1.2 Regularity and Recognizabilty

Given some normalized regular tree grammar $G = (S, N, \mathcal{F}, R_G)$, we show how to build a top-down tree automaton which recognizes $L(G)$. We define $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ by

- $Q = \{q_A \mid A \in N\}$
- $I = \{q_S\}$
- $q_A(f(x_1, \dots, x_n)) \to f(q_{A_1}(x_1), \dots, q_{A_n}(x_n)) \in \Delta$ if and only if $A \to f(A_1, \dots, A_n) \in R_G$.

A standard proof by induction on derivation length yields $L(G) = L(\mathcal{A})$. Therefore we have proved that the languages generated by regular tree grammar are recognizable languages.

The next question to ask is whether recognizable tree languages can be generated by regular tree grammars. If L is a regular tree language, there exists a top-down tree automata $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ such that $L = L(\mathcal{A})$. We define $G = (S, N, \mathcal{F}, R_G)$ with S a new symbol, $N = \{A_q \mid q \in Q\}$, $R_G = \{A_q \to f(A_{q_1}, \dots, A_{q_n}) \mid q(f(x_1, \dots, x_n)) \to f(q_1(x_1), \dots, q_n(x_n)) \in R\} \cup \{S \to A_I \mid A_I \in I\}$. A standard proof by induction on derivation length yields $L(G) = L(\mathcal{A})$.

Combining these two properties, we get the equivalence between recognizability and regularity.

Theorem 2.1.5. *A tree language is recognizable if and only if it is a regular tree language.*

2.2 Regular Expressions. Kleene's Theorem for Tree Languages

Going back to our example of lists of non-negative integers, we can write the sets defined by the non-terminals Nat and $List$ as follows.

$$\begin{aligned} Nat &= \{0, s(0), s(s(0)), \dots\} \\ List &= \{nil, cons(\_, nil), cons(\_, cons(\_, nil)), \dots\} \end{aligned}$$

where $\_$ stands for any element of Nat. There is some regularity in each set which reminds of the regularity obtained with regular word expressions constructed with the union, concatenation and iteration operators. Therefore we can try to use the same idea to denote the sets Nat and $List$. However, since we are dealing with trees and not words, we must put some information to indicate where concatenation and iteration must take place. This is done by using a new symbol which behaves as a constant. Moreover, since we have two independent iterations, the first one for Nat and the second one for $List$, we shall use two different new symbols $\square_1$ and $\square_2$ and a natural extension of regular word expressions leads us to denote the sets Nat and $List$ as follows.

$$\begin{aligned}&Nat = s(\square_1)^{*,\square_1} \ ._{\square_1}\ 0\\ &List = nil + cons(\ (s(\square_1)^{*,\square_1}\ ._{\square_1}\ 0)\ ,\square_2)^{*,\square_2}\ ._{\square_2}\ nil\end{aligned}$$

Actually the first term nil in the second equality is redundant and a shorter (but slightly less natural) expression yields the same language.

We are going to show that this is a general phenomenon and that we can define a notion of regular expressions for trees and that Kleene's theorem for words can be generalized to trees. Like in the example, we must introduce a particular set of constants $\mathcal{K}$ which are used to indicate the positions where concatenation and iteration take place in trees. This explains why the syntax of regular tree expressions is more cumbersome than the syntax of word regular expressions. These new constants are usually denoted by $\square_1, \square_2, \ldots$ Therefore, in this section, we consider trees constructed on $\mathcal{F} \cup \mathcal{K}$ where $\mathcal{K}$ is a distinguished finite set of symbols of arity 0 disjoint from $\mathcal{F}$.

2.2.1 Substitution and Iteration

First, we have to generalize the notion of substitution to languages, replacing some $\square_i$ by a tree of some language L_i. The main difference with term substitution is that different occurrences of the same constant $\square_i$ can be replaced by different terms of L_i. Given a tree t of $T(\mathcal{F} \cup \mathcal{K})$, $\square_1, \ldots, \square_n$ symbols of $\mathcal{K}$ and $L_1, \ldots, L_n$ languages of $T(\mathcal{F} \cup \mathcal{K})$, the ***tree substitution*** (substitution for short) of $\square_1, \ldots, \square_n$ by $L_1, \ldots, L_n$ in t, denoted by $t\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\}$, is the tree language defined by the following identities.

- $\square_i\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = L_i$ for $i = 1, \ldots, n$,
- $a\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = \{a\}$ for all $a \in \mathcal{F} \cup \mathcal{K}$ such that arity of a is 0 and $a \neq \square_1, \ldots, a \neq \square_n$,
- $f(s_1, \ldots, s_n)\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} =$
$$\{f(t_1, \ldots, t_n) \mid t_i \in s_i\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\}\}$$

Example 2.2.1. Let $\mathcal{F} = \{0, nil, s(), cons(,)\}$ and $\mathcal{K} = \{\square_1, \square_2\}$, let

$$t = cons(\square_1, cons(\square_1, \square_2))$$

and let

$$L_1 = \{0, s(0)\}$$

then

$$\begin{aligned} t\{\square_1 \leftarrow L\} \quad = \quad & \{cons(0, cons(0, \square_2)),\\ & cons(0, cons(s(0), \square_2)),\\ & cons(s(0), cons(0, \square_2)),\\ & cons(s(0), cons(s(0), \square_2))\} \end{aligned}$$

Symbols of $\mathcal{K}$ are mainly used to distinguish places where the substitution must take place, and they are usually not relevant. For instance, if t is a tree on the alphabet $\mathcal{F} \cup \{\square\}$ and L be a language of trees on the alphabet $\mathcal{F}$, then the trees of $t\{\square \leftarrow L\}$ don't contain the symbol $\square$.

The substitution operation generalizes to languages in a straightforward way. When $L, L_1, \ldots, L_n$ are languages of $T(\mathcal{F} \cup \mathcal{K})$ and $\square_1, \ldots, \square_n$ are elements of $\mathcal{K}$, we define $L\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\}$ to be the set $\bigcup_{t \in L} \{\ t\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\}\}$.

Now, we can define the concatenation operation for tree languages. Given L and M two languages of $T_{\mathcal{F} \cup \mathcal{K}}$, and $\square$ be an element of $\mathcal{K}$, the ***concatenation*** of M to L through $\square$, denoted by $L \,._\square\, M$, is the set of trees obtained by substituting the occurrence of $\square$ in trees of L by trees of M, i.e. $L \,._\square\, M = \bigcup_{t \in L} \{t\{\square \leftarrow M\}\}$.

To define the closure of a language, we must define the sequence of successive iterations. Given L a language of $T(\mathcal{F} \cup \mathcal{K})$ and $\square$ an element of $\mathcal{K}$, the sequence $L^{n,\square}$ is defined by the equalities.

- $L^{0,\ \square} = \{\square\}$
- $L^{n+1,\ \square} = L^{n,\ \square} \cup L \,._\square\, L^{n,\ \square}$

The ***closure*** $L^{*,\square}$ of L is the union of all $L^{n,\ \square}$ for non-negative n, i.e., $L^{*,\square} = \cup_{n \geq 0} L^{n,\square}$. From the definition, one gets that $\{\square\} \subseteq L^{*,\square}$ for any L.

Example 2.2.2. Let $\mathcal{F} = \{0, nil, s(), cons(,)\}$, let $L = \{0, cons(0, \square)\}$ and $M = \{nil, cons(s(0), \square)\}$, then

$$\begin{array}{rcl} L \,._\square\, M & = & \{0, cons(0, nil), cons(0, cons(s(0), \square))\} \\ L^{*,\square} & = & \{\square\} \cup \\ & & \{0, cons(0, \square)\} \cup \\ & & \{0, cons(0, \square), cons(0, cons(0, \square))\} \cup \ldots \end{array}$$

We prove now that the substitution and concatenation operations yield regular languages when they are applied to regular languages.

Proposition 2.2.3. *Let L be a regular tree language on $\mathcal{F} \cup \mathcal{K}$, let $L_1, \ldots, L_n$ be regular tree languages on $\mathcal{F} \cup \mathcal{K}$, let $\square_1, \ldots, \square_n \in \mathcal{K}$, then $L\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\}$ is a regular tree language.*

Proof. Since L is regular, there exists some normalized regular tree grammar $G = (S, N, \mathcal{F} \cup \mathcal{K}, R)$ such that $L = L(G)$, and for each $i = 1, \ldots, n$ there exists a normalized grammar $G_i = (S_i, N_i, \mathcal{F} \cup \mathcal{K}, R_i)$ such that $L_i = L(G_i)$. We can assume that the sets of non-terminals are pairwise disjoint. The idea of the proof is to construct a grammar G' which starts by generating trees like G but replaces the generation of a symbol $\square_i$ by the generation of a tree of L_i via a branching towards the axiom of G_i. More precisely, we show that $L\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = L(G')$ where $G' = (S, N', \mathcal{F} \cup \mathcal{K}, R')$ such that

- $N' = N \cup N_1 \cup \ldots \cup N_n$,
- R' contains the rules of R_i and the rules of R but the rules $A \rightarrow \square_i$ which are replaced by the rules $A \rightarrow S_i$, where S_i is the axiom of L_i.

A straightforward induction on the height of trees proves that G' generates each tree of $L\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$.

The converse is to prove that $L(G') \subseteq L\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$. This is achieved by proving the following property by induction on the derivation length.

$$\begin{gathered} A \xrightarrow{+} s' \text{ where } s' \in T(\mathcal{F} \cup \mathcal{K}) \text{ using the rules of } G' \\ \text{if and only if} \\ \text{there is some } s \text{ such that } A \xrightarrow{+} s \text{ using the rules of } G \text{ and} \\ s' \in s\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}. \end{gathered}$$

- base case: $A \rightarrow s$ in one step. Therefore this derivation is a derivation of the grammar G and no $\Box_i$ occurs in s, yielding $s \in L\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$
- induction step: we assume that the property is true for any terminal and derivation of length less than n. Let A be such that $A \rightarrow s'$ in n steps. This derivation can be decomposed as $A \rightarrow s_1 \xrightarrow{+} s'$. We distinguish several cases depending on the rule used in the derivation $A \rightarrow s_1$.
  - the rule is $A \rightarrow f(A_1, \ldots, A_m)$, therefore $s' = f(t_1, \ldots, t_m)$ and $t_i \in L(A_i)\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$, therefore $s' \in L(A)\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$,
  - the rule is $A \rightarrow S_i$, therefore $A \rightarrow \Box_i \in R$ and $s' \in L_i$ and $s' \in L(A)\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$.
  - the rule $A \rightarrow a$ with $a \in \mathcal{F}$, a of arity 0, $a \neq \Box_1, \ldots, a \neq \Box_n$ are not considered since no further derivation can be done.

□

The following proposition states that regular languages are stable also under closure.

Proposition 2.2.4. *Let L be a regular tree language of $T(\mathcal{F} \cup \mathcal{K})$, let $\Box \in \mathcal{K}$, then $L^{*,\Box}$ is a regular tree language of $T(\mathcal{F} \cup \mathcal{K})$.*

Proof. There exists a normalized regular grammar $G = (S, N, \mathcal{F} \cup \mathcal{K}, R)$ such that $L = L(G)$ and we obtain from G a grammar $G' = (S', N \cup \{S'\}, \mathcal{F} \cup \mathcal{K}, R')$ for $L^{*,\Box}$ by replacing rules leading to $\Box$ such as $A \rightarrow \Box$ by rules $A \rightarrow S'$ leading to the (new) axiom. Moreover we add the rule $S' \rightarrow \Box$ to generate $\{\Box\} = L^{0,\Box}$ and the rule $S' \rightarrow S$ to generate $L^{i,\Box}$ for $i > 0$. By construction G' generates the elements of $L^{*,\Box}$.

Conversely a proof by induction on the length on the derivation proves that $L(G') \subseteq L^{*,\Box}$. □

2.2.2 Regular Expressions and Regular Tree Languages

Now, we can define regular tree expression in the flavor of regular word expression using the $+, ._\Box, ^{*,\Box}$ operators.

Definition 2.2.5. *The set $Regexp(\mathcal{F}, \mathcal{K})$ of* ***regular tree expressions*** *on $\mathcal{F}$ and $\mathcal{K}$ is the smallest set such that:*

- *the empty set* $\emptyset$ *is in* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$
- *if* $a \in \mathcal{F}_0 \cup \mathcal{K}$ *is a constant, then* $a \in \mathit{Regexp}(\mathcal{F},\mathcal{K})$,
- *if* $f \in \mathcal{F}_n$ *has arity* $n > 0$ *and* $E_1, \ldots, E_n$ *are regular expressions of* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$ *then* $f(E_1, \ldots, E_n)$ *is a regular expression of* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$,
- *if* E_1, E_2 *are regular expressions of* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$ *then* $(E_1 + E_2)$ *is a regular expression of* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$,
- *if* E_1, E_2 *are regular expressions of* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$ *and* $\square$ *is an element of* $\mathcal{K}$ *then* $E_1 \,._{\square}\, E_2$ *is a regular expression of* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$,
- *if* E *is a regular expression of* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$ *and* $\square$ *is an element of* $\mathcal{K}$ *then* $E^{*,\square}$ *is a regular expression of* $\mathit{Regexp}(\mathcal{F},\mathcal{K})$.

Each regular expression E represents a set of terms of $T(\mathcal{F} \cup \mathcal{K})$ which we denote $[\![E]\!]$ and which is formally defined by the following equalities.

- $[\![\emptyset]\!] = \emptyset$,
- $[\![a]\!] = \{a\}$ for $a \in \mathcal{F}_0 \cup \mathcal{K}$,
- $[\![f(E_1, \ldots, E_n)]\!] = \{f(s_1, \ldots, s_n) \mid s_1 \in [\![E_1]\!], \ldots, s_n \in [\![E_n]\!]\}$,
- $[\![E_1 + E_2]\!] = [\![E_1]\!] \cup [\![E_2]\!]$,
- $[\![E_1 ._{\square}\, E_2]\!] = [\![E_1]\!]\{\square \leftarrow [\![E_2]\!]\}$,
- $[\![E^{*,\square}]\!] = [\![E]\!]^{*,\square}$

Example 2.2.6. Let $\mathcal{F} = \{0, nil, s(), cons(,)\}$ and $\square \in \mathcal{K}$ then

$$(cons(0, \square)^{*,\square})._{\square} nil$$

is a regular expression of $\mathit{Regexp}(\mathcal{F},\mathcal{K})$ which denotes the set of lists of zeros:

$$\{nil, cons(0, nil), cons(0, cons(0, nil)), \ldots\}$$

In the remainder of this section, we compare the relative expressive power of regular expressions and regular languages. It is easy to prove that for each regular expression E, the set $[\![E]\!]$ is a regular tree language. The proof is done by structural induction on E. The first three cases are obvious and the two last cases are consequences of Propositions 2.2.4 and 2.2.3. The converse, i.e. a regular tree language can be denoted by a regular expression, is more involved and the proof is similar to the proof of Kleene's theorem for word languages. Let us state the result first.

Proposition 2.2.7. *Let* $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ *be a bottom-up tree automaton, then there exists a regular expression* E *of* $\mathit{Regexp}(\mathcal{F}, Q)$ *such that* $L(\mathcal{A}) = [\![E]\!]$.

The occurrence of symbols of Q in the regular expression denoting $L(\mathcal{A})$ doesn't cause any trouble since a regular expression of $\mathit{Regexp}(\mathcal{F}, Q)$ can denote a language of $T_{\mathcal{F}}$.

Proof. The proof is similar to the proof for word languages and word automata. For each $1 \leq i, j, \leq |Q|, K \subseteq Q$, we define the set $T(i, j, K)$ as the set of trees t of $T(\mathcal{F} \cup K)$ such that there is a run r of $\mathcal{A}$ on t satisfying the following properties:

- $r(\epsilon) = q_i$,
- $r(p) \in \{q_1, \ldots, q_j\}$ for all $p \neq \epsilon$ labelled by a function symbol.

Roughly speaking, a term is in $T(i, j, K)$ if we can reach q_i at the root by using only states in $\{q_1, \ldots, q_j\}$ when we assume that the leaves are states of K. By definition, $L(\mathcal{A})$, the language accepted by $\mathcal{A}$, is the union of the $T(i, |Q|, \emptyset)$'s for i such that q_i is a final state: these terms are the terms of $T(\mathcal{F})$ such that there is a successful run using any possible state of Q. Now, we prove by induction on j that $T(i, j, K)$ can be denoted by a regular expression of *Regexp*$(\mathcal{F}, Q)$.

- Base case $j = 0$. The set $T(i, 0, K)$ is the set of trees t where the root is labelled by q_i, the leaves are in $\mathcal{F} \cup K$ and no internal node is labelled by some q. Therefore there exist $a_1, \ldots, a_n, a \in \mathcal{F} \cup K$ such that $t = f(a_1, \ldots, a_n)$ or $t = a$, hence $T(i, 0, K)$ is finite and can be denoted by a regular expression of *Regexp*$(\mathcal{F} \cup Q)$.
- Induction case. Let us assume that for any $i', K' \subseteq Q$ and $0 \leq j' < j$, the set $T(i', j', K')$ can be denoted by a regular expression. We can write the following equality:

  $$\begin{aligned} T(i,j,K) = \;& T(i, j-1, K) \\ & \cup \\ & T(i, j-1, K \cup \{q_j\}) \;.q_j\; T(j, j-1, K \cup \{q_j\})^{\,*,q_j} \;.q_j\; T(j, j-1, K) \end{aligned}$$

  The inclusion of $T(i, j, K)$ in the right-hand side of the equality can be easily seen from Figure 2.2.2.

  The converse inclusion is also not difficult. By definition:
  $T(i, j-1, K) \subseteq T(i, j, K)$
  and an easy proof by induction on the number of occurrences of q_j yields:
  $T(i, j-1, K \cup \{q_j\}) \;.q_j\; T(j, j-1, K \cup \{q_j\})^{\,*,q_j} \;.q_j\; T(j, j-1, K) \subseteq T(i, j, K)$

  By induction hypothesis, each set of the right-hand side of the equality defining $T(i, j, K)$ can be denoted by a regular expression of *Regex*$(\mathcal{F} \cup Q)$. This yields the desired result because the union of these sets is represented by the sum of the corresponding expressions.

□

Since we have already seen that regular expressions denote recognizable tree languages and that recognizable languages are regular, we can state Kleene's theorem for tree languages.

Theorem 2.2.8. *A tree language is recognizable if and only if it can be denoted by a regular tree expression.*

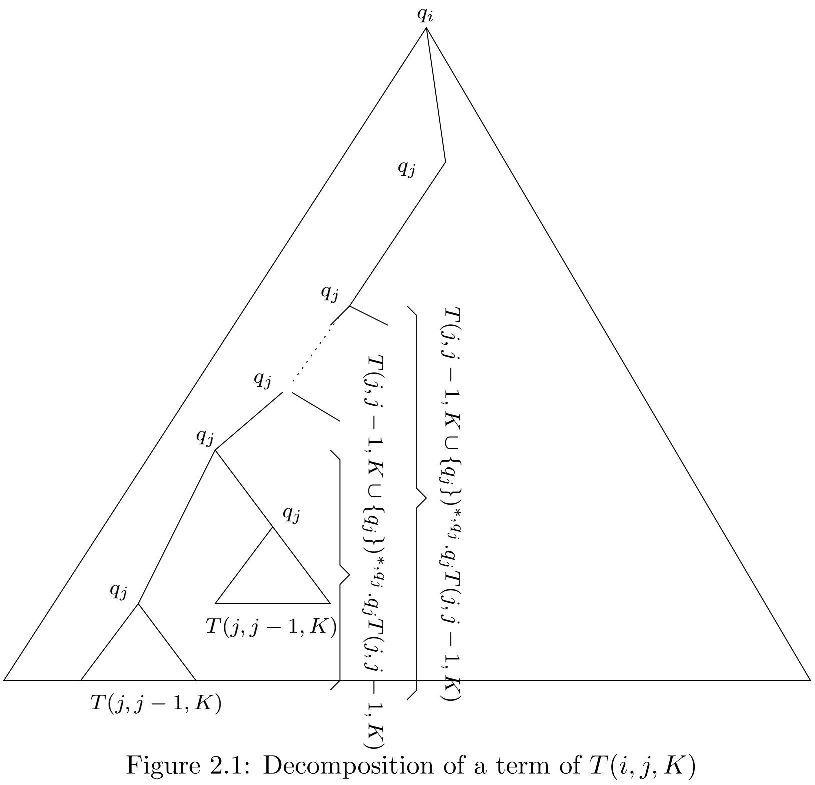

q_i
q_j
q_j
q_j
q_j
q_j
q_j
$T(j, j-1, K)$
$T(j, j-1, K)$
$T(j, j-1, K \cup \{q_j\})^{*,q_j} .q_j T(j, j-1, K)$
$T(j, j-1, K \cup \{q_j\})^{*,q_j} .q_j T(j, j-1, K)$

Figure 2.1: Decomposition of a term of $T(i, j, K)$

2.3 Regular Equations

Looking at our example of the set of lists of non-negative integers, we can realize that these lists can be defined by equations instead of grammar rules. For instance, denoting set union by +, we could replace the grammar given in Section 2.1.1 by the following equations.

$$\begin{array}{l} Nat = 0 \ + \ s(Nat) \\ List = nil \ + \ cons(Nat, List) \end{array}$$

where the variables are $List$ and Nat. To get the usual lists of non-negative numbers, we must restrict ourselves to the least fixed-point solution of this set of equations. Systems of language equations do not always have a solution nor does a least solution always exists. Therefore we shall study ***regular equation systems*** defined as follows.

Definition 2.3.1. *Let $X_1, \dots, X_n$ be variables denoting sets of trees, for $1 \leq j \leq p$, $1 \leq i \leq m_j$, let s_i^j be a term over $\mathcal{F} \cup \{X_1, \dots, X_n\}$, then a regular equation system S is a set of equations of the form:*

$$\begin{array}{rcl} X_1 & = & s_1^1 + \dots + s_{m_1}^1 \\ & \vdots & \\ X_p & = & s_1^p + \dots + s_{m_p}^p \end{array}$$

A solution of S is any n-tuple $(L_1, \dots, L_n)$ of languages of $T(\mathcal{F})$ such that

$$\begin{array}{l} L_1 = s_1^1\{X_1 \leftarrow L_1, \dots, X_n \leftarrow L_n\} \cup \dots \cup s_{m_1}^1\{X_1 \leftarrow L_1, \dots, X_n \leftarrow L_n\} \\ \vdots \\ L_p = s_1^p\{X_1 \leftarrow L_1, \dots, X_n \leftarrow L_n\} \cup \dots \cup s_{m_p}^p\{X_1 \leftarrow L_1, \dots, X_n \leftarrow L_n\} \end{array}$$

Since equations with the same left-hand side can be merged into one equation, and since we can add equations $X_k = X_k$ without changing the set of solutions of a system, we assume in the following that $p = n$.

The ordering $\subseteq$ is defined on $T(\mathcal{F})^n$ by

$$(L_1, \dots, L_n) \subseteq (L'_1, \dots, L'_n) \quad \text{iff } L_i \subseteq L'_i \text{ for all } i = 1, \dots, n$$

By definition $(\emptyset, \dots, \emptyset)$ is the smallest element of $\subseteq$ and each increasing sequence has an upper bound. To a system of equations, we associate the fixed-point operator $\mathcal{TS} : T(\mathcal{F})^n \rightarrow T(\mathcal{F})^n$ defined by:

$$\mathcal{TS}(L_1, \dots, L_n) = (L'_1, \dots, L'_n)$$

where

$$\begin{array}{l} L'_1 = L_1 \cup s_1^1\{X_1 \leftarrow L_1, \dots, X_n \leftarrow L_n\} \cup \dots \cup s_{m_1}^1\{X_1 \leftarrow L_1, \dots, X_n \leftarrow L_n\} \\ \vdots \\ L'_n = L_n \cup s_1^n\{X_1 \leftarrow L_1, \dots, X_n \leftarrow L_n\} \cup \dots \cup s_{m_n}^n\{X_1 \leftarrow L_1, \dots, X_n \leftarrow L_n\} \end{array}$$

Example 2.3.2. Let S be

$$\begin{array}{rcl} Nat & = & 0 + s(Nat) \\ List & = & nil + cons(Nat, List) \end{array}$$

then

$$\mathcal{TS}(\emptyset,\emptyset) = (\{0\}, \{nil\})$$
$$\mathcal{TS}^2(\emptyset,\emptyset) = (\{0, s(0)\}, \{nil, cons(0, nil)\})$$

Using a classical approach we use the fixed-point operator to compute the least fixed-point solution of a system of equations.

Proposition 2.3.3. *The fixed-point operator* $\mathcal{TS}$ *is continuous and its least fixed-point* $\mathcal{TS}^\omega(\emptyset,\dots,\emptyset)$ *is the least solution of* S*.*

Proof. We show that $\mathcal{TS}$ is continuous in order to use Knaster-Tarski's theorem on continuous operators. By construction, $\mathcal{TS}$ is monotonous, and the last point is to prove that if $S_1 \subseteq S_2 \subseteq \dots$ is an increasing sequence of n-tuples of languages, the equality $\mathcal{TS}(\bigcup_{i\geq 1} S_i) = \bigcup_{i\geq 1} \mathcal{TS}(S_i)$ holds. By definition, each S_i can be written as $(S_1^i, \dots, S_n^i)$.

- We have that $\bigcup_{i\geq 1} \mathcal{TS}(S_i) \subseteq \mathcal{TS}\big(\bigcup_{i\geq 1}(S_i)\big)$ holds since the sequence $S_1 \subseteq S_2 \subseteq \dots$ is increasing and the operator $\mathcal{TS}$ is monotonous.
- Conversely we must prove $\mathcal{TS}(\bigcup_{i\geq 1} S_i) \subseteq \bigcup_{i\geq 1} \mathcal{TS}(S_i)$.

  Let $v = (v^1, \dots, v^n) \in \mathcal{TS}(\bigcup_{i\geq 1} S_i)$. Then for each $k = 1, \dots, n$ either $v^k \in \bigcup_{i\geq 1} S_i$ hence $v^k \in S_{l_k}$ for some l_k, or there is some $u = (u^1, \dots, u^n) \in \bigcup_{i\geq 1} S_i$ such that $v^k = s^k_{j_k}\{X_1 \leftarrow u^1, \dots, X_n \leftarrow u^n\}$. Since the sequence $(S_{i,\ i\geq 1})$ is increasing we have that $u \in S_{l_k}$ for some l_k. Therefore $v^k \in \mathcal{TS}(S_L) \subseteq \bigcup_{i\geq 1} \mathcal{TS}(S_i)$ for $L = \max\{l_k \mid k = 1, \dots, n\}$.

□

We have introduced systems of regular equations to get an algebraic characterization of regular tree languages stated in the following theorem.

Theorem 2.3.4. *The least fixed-point solution of a system of regular equations is a tuple of regular tree languages. Conversely each regular tree language is a component of the least solution of a system of regular equations.*

Proof. Let S be a system of regular equations. Let $G_i = (X_i, \{X_1, \dots, X_n\}, \mathcal{F}, R)$ where $R = \cup_{k=1,\dots,n}\{X_k \to s_k^1, \dots, X_k \to s_k^{j_k}\}$ if the k^{th} equation of S is $X_k = s_k^1 + \dots + s_k^{j_k}$. We show that $L(G_i)$ is the i^{th} component of $(L_1, \dots, L_n)$ the least fixed-point solution of S.

- We prove that $\mathcal{TS}^p(\emptyset, \dots, \emptyset) \subseteq (L(G_1), \dots, L(G_n))$ by induction on p.

  Let us assume that this property holds for all $p' \leq p$. Let $u = (u_1, \dots, u_n)$ be an element of $TS^{p+1}(\emptyset, \dots, \emptyset) = \mathcal{TS}(\mathcal{TS}^p(\emptyset, \dots, \emptyset))$. For each i in $1, \dots, n$, either $u^i \in \mathcal{TS}^p(\emptyset, \dots, \emptyset)$ and $u_i \in L(G_i)$ by induction hypothesis, or there exist $v^i = (v_1^i, \dots, v_n^i) \in TS^p(\emptyset, \dots, \emptyset)$ and s_i^j such that $u_i = s_i^j\{X_1 \to v_1^i, \dots, X_n \to v_n^i\}$. By induction hypothesis $v_j^i \in L(G_j)$ for $j = 1, \dots, n$ therefore $u_i \in L(G_i)$.

- We prove now that $(L(X_1), \ldots, L(X_n)) \subseteq \mathcal{TS}^\omega(\emptyset, \ldots, \emptyset)$ by induction on derivation length.

  Let us assume that for each $i = 1, \ldots, n$, for each $p' \leq p$, if $X_i \rightarrow^{p'} u_i$ then $u_i \in \mathcal{TS}^{p'}(\emptyset, \ldots, \emptyset)$. Let $X_i \rightarrow^{p+1} u_i$, then $X_i \rightarrow s_i^j(X_1, \ldots, X_n) \rightarrow^p v_i$ with $u_i = s_i^j(v_1, \ldots, v_n)$ and $X_j \rightarrow^{p'} v_j$ for some $p' \leq p$. By induction hypothesis $v_j \in \mathcal{TS}^{p'}(\emptyset, \ldots, \emptyset)$ which yields that $u_i \in \mathcal{TS}^{p+1}(\emptyset, \ldots, \emptyset)$.

Conversely, given a regular grammar $G = (S, \{A_1, \ldots, A_n\}, \mathcal{F}, R)$, with $R = \{A_1 \rightarrow s_1^1, \ldots, A_1 \rightarrow s_{p_1}^1, \ldots, A_n \rightarrow s_1^n, \ldots, A_n \rightarrow s_{p_n}^n\}$, a similar proof yields that the least solution of the system

$$\begin{array}{rcl} A_1 & = & s_1^1 + \ldots + s_{p_1}^1 \\ & & \ldots \\ A_n & = & s_1^n + \ldots + s_{p_n}^n \end{array}$$

is $(L(A_1), \ldots, L(A_n))$. □

Example 2.3.5. The grammar with axiom *List*, non-terminals *List*, *Nat* terminals $0, s(), nil, cons(,)$ and rules

$$\begin{array}{rcl} List & \rightarrow & nil \\ List & \rightarrow & cons(Nat, List) \\ Nat & \rightarrow & 0 \\ Nat & \rightarrow & s(Nat) \end{array}$$

generates the second component of the least solution of the system given in Example 2.3.2.

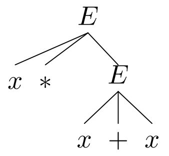
2.4 Context-free Word Languages and Regular Tree Languages

Context-free word languages and regular tree languages are strongly related. This is not surprising since derivation trees of context-free languages and derivations of tree grammars look alike. For instance let us consider the context-free language of arithmetic expressions on $+$,$*$ and a variable x. A context-free word grammar generating this set is $E \rightarrow x \mid E + E \mid E * E$ where E is the axiom. The generation of a word from the axiom can be described by a derivation tree which has the axiom at the root and where the generated word can be read by picking up the leaves of the tree from the left to the right (computing what we call the yield of the tree). The rules for constructing derivation trees show some regularity, which suggests that this set of trees is regular. The aim of this section is to show that this is true indeed. However, there are some traps which must be avoided when linking tree and word languages. First, we describe how to relate words and trees. The symbols of $\mathcal{F}$ are used to build trees but also words (by taking a symbol of $\mathcal{F}$ as a letter). The ***Yield*** operator computes a word from a tree by concatenating the leaves of the tree from the left to the right. More precisely, it is defined as follows.

$$Yield(a) = a \quad \text{if } a \in \mathcal{F}_0$$
$$Yield(f(s_1,\ldots,s_n)) = Yield(s_1)\ldots Yield(s_n) \quad \text{if } f \in \mathcal{F}_n,\ s_i \in T(\mathcal{F}).$$

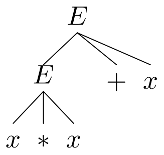
Example 2.4.1. Let $\mathcal{F} = \{x, +, *, E(,,)\}$ and let

$s =$



then $Yield(s) = x * x + x$ which is a word on $\{x, *, +\}$. Note that $*$ and $+$ are not the usual binary operator but syntactical symbols of arity 0. If

$t =$



then $Yield(t) = x * x + x$.

We recall that a ***context-free!word grammar*** G is a tuple (S, N, T, R) where S is the axiom, N the set of non-terminal letters, T the set of terminal letters, R the set of production rules of the form $A \to \alpha$ with $A \in N, \alpha \in (T \cup N)^*$. The usual definition of derivation trees of context free word languages allow nodes labelled by a non-terminal A to have a variable number of sons, which is equal to the length of the right-hand side α of the rule $A \to \alpha$ used to build the derivation tree at this node.

Since tree languages are defined for signatures where each symbol has a fixed arity, we introduce a new symbol (A, m) for each $A \in N$ such that there is a rule $A \to \alpha$ with α of length m. Let $\mathcal{G}$ be the set composed of these new symbols and of the symbols of T. The set of derivation trees issued from $a \in \mathcal{G}$, denoted by $D(G, a)$ is the smallest set such that:

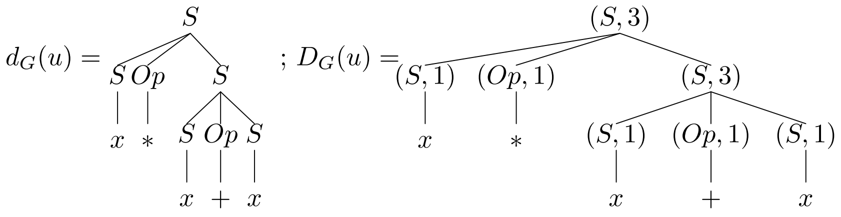
- $D(G, a) = \{a\}$ if $a \in T$,
- $(a, 0)(\epsilon) \in D(G, a)$ if $a \to \epsilon \in R$ where ϵ is the empty word,
- $(a, p)(t_1, \ldots, t_p) \in D(G, (a, p))$ if $t_1 \in D(G, a_1), \ldots, t_p \in D(G, a_p)$ and $(a \to a_1 \ldots a_p) \in R$ where $a_i \in \mathcal{G}$.

The set of ***derivation trees*** of G is $D(G) = \cup_{(S,i)\in\mathcal{G}} D(G, (S, i))$.

Example 2.4.2. Let $T = \{x, +, *\}$ and let G be the context free word grammar with axiom S, non terminal Op, and rules

$$\begin{array}{l} S \rightarrow S\ Op\ S \\ S \rightarrow x \\ Op \rightarrow + \\ Op \rightarrow * \end{array}$$

Let the word $u = x * x + x$, a derivation tree for u with G is $d_G(u)$, and the same derivation tree with our notations is $D_G(u) \in D(G, S)$



By definition, the language generated by a context-free word grammar G is the set of words computed by applying the *Yield* operator to derivation trees of G. The next theorem states how context-free word languages and regular tree languages are related.

Theorem 2.4.3. *The following statements hold.*

1. *Let G be a context-free word grammar, then the set of derivation trees of $L(G)$ is a regular tree language.*
2. *Let L be a regular tree language then $Yield(L)$ is a context-free word language.*
3. *There exists a regular tree language which is not the set of derivation trees of a context-free language.*

Proof. We give the proofs of the three statements.

1. Let $G = (S, N, T, R)$ be a context-free word language. We consider the tree grammar $G' = (S, N, \mathcal{F}, R')$ such that
    - the axiom and the set of non-terminal symbols of G and G' are the same,
    - $\mathcal{F} = T \cup \{\epsilon\} \cup \{(A, n) \mid A \in N,\ \exists A \rightarrow \alpha \in R \text{ with } \alpha \text{ of length } n\}$,
    - if $A \rightarrow \epsilon \in R$ then $A \rightarrow (A, 0)(\epsilon) \in R'$
    - if $(A \rightarrow a_1 \ldots a_p) \in R$ then $A \rightarrow (A, p)(a_1, \ldots, a_p) \in R'$

    Then $L(G) = \{\mathit{Yield}(s) \mid s \in L(G')\}$. The proof is a standard induction on derivation length. It is interesting to remark that there may and usually do exist several tree languages (not necessarily regular) such that the corresponding word language obtained via the *Yield* operator is a given context-free word language.

2. Let G be a normalized tree grammar (S, X, N, R). We build the word context-free grammar $G' = (S, X, N, R')$ such that a rule $X \rightarrow X_1 \ldots X_n$ (resp. $X \rightarrow a$) is in R' if and only if the rule $X \rightarrow f(X_1, \ldots, X_n)$ (resp. $X \rightarrow a$) is in R for some f. It is straightforward to prove by induction on the length of derivation that $L(G') = \mathit{Yield}(L(G))$.

3. Let G be the regular tree grammar with axiom S, non-terminals S, G_1, G_2, terminals $s(,), g(), a, b$ and rules $S \rightarrow s(G_1, G_2)$, $G_1 \rightarrow g(a)$, and $G_2 \rightarrow g(b)$. The language $L(G)$ consists of the single tree $s(g(a), g(b))$.

   Assume that $L(G)$ is the set of derivation trees of some context-free word grammar. To generate the first node of the tree, one must have a rule $S \rightarrow G\ G$ where S is the axiom, and rules $G \rightarrow a$, $G \rightarrow b$ (to get the inner nodes). Therefore the tree $S(G(a), G(a))$ should be in $L(G)$ which is not the case.

   We have shown that the class of derivation tree languages is a strict subset of the class of regular tree languages. It can be shown (see Exercise 2.5) that the class of derivation tree languages is the class of local languages. Moreover, every regular tree language is the image of a local tree language by an alphabetic homomorphism. For instance, on the running example, we can consider the context-free word grammar $S \rightarrow G_1\ G_2$ where S is the axiom, and rules $G_1 \rightarrow a$, $G_2 \rightarrow b$. The derivation tree language is the single tree $S(G_1(a), G_2(b))$. Now, $L(G)$ can be obtained by applying the tree homomorphism h determined by $h_{\mathcal{F}}$ defined by: $h_{\mathcal{F}}(S) = s(x_1, x_2)$, $h_{\mathcal{F}}(G_1) = g(x_1)$ and $h_{\mathcal{F}}(G_2) = g(x_1)$.

□

The class of derivation trees of context-free languages is a strict subclass of the class of recognizable tree languages. Sets of derivation trees correspond to local tree languages and the class of recognizable tree languages is the image of the class of local languages by alphabetic homomorphisms (see Exercice 2.5).

2.5 Beyond Regular Tree Languages: Context-free Tree Languages

For word language, the story doesn't end with regular languages but there is a strict hierarchy.

$$\text{regular} \subset \text{context-free} \subset \text{recursively enumerable}$$

Recursively enumerable tree languages are languages generated by tree grammars as defined in the beginning of the chapter, and this class is far too general to have good properties. Actually, any Turing machine can be simulated by a one rule rewrite system which shows how powerful tree grammars are (any grammar rule can be seen as a rewrite rule by considering both terminals and non-terminals as syntactical symbols). Therefore, most of the research has been done on context-free tree languages which we describe now.

2.5.1 Context-free Tree Languages

A ***context-free!tree grammar*** is a tree grammar $G = (S, N, \mathcal{F}, R)$ where the rules have the form $X(x_1, \ldots, x_n) \rightarrow t$ with t a tree of $T(\mathcal{F} \cup N \cup \{x_1, \ldots, x_n\})$, $x_1, \ldots, x_n \in \mathcal{X}$ where $\mathcal{X}$ is a set of reserved variables with $\mathcal{X} \cap (\mathcal{F} \cup N) = \emptyset$, X a non-terminal of arity n.

Example 2.5.1. The grammar of axiom *Prog*, set of non-terminals $\{Prog, Nat, Fact()\}$, set of terminals $\{0, s, if(,), eq(,), not(), times(,), dec()\}$ and rules

$$\begin{array}{lcl} Prog & \rightarrow & Fact(Nat) \\ Nat & \rightarrow & 0 \\ Nat & \rightarrow & s(Nat) \\ Fact(x) & \rightarrow & if(eq(x,0), s(0)) \\ Fact(x) & \rightarrow & if(not(eq(x,0)), times(x, Fact(dec(x)))) \end{array}$$

where $\mathcal{X} = \{x\}$ is a context-free tree grammar. The reader can easily see that the last rule is the classical definition of the factorial function.

The derivation relation associated to a context-free tree grammar G is a generalization of the derivation relation for regular tree grammar. The derivation relation $\rightarrow$ is a relation on pairs of terms of $T(\mathcal{F} \cup N)$ such that $s \rightarrow t$ iff there is a rule $X(x_1, \ldots, x_n) \rightarrow \alpha \in R$, a context C such that $s = C[X(t_1, \ldots, t_n)]$ and $t = C[\alpha\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}]$. For instance, the previous grammar can yield the sequence of derivations

$$Prog \rightarrow Fact(Nat) \rightarrow Fact(0) \rightarrow if(eq(0,0), s(0))$$

The language generated by G, denoted by $L(G)$ is the set of terms of $T(\mathcal{F})$ which can be reached by successive derivations starting from the axiom. Such languages are called ***context-free!tree languages***. Context-free tree languages are closed under union, concatenation and closure. Like in the word case, one can define pushdown tree automata which recognize exactly the set of context-free tree languages. We discuss only IO and OI grammars and we refer the reader to the bibliographic notes for more information.

2.5.2 IO and OI Tree Grammars

Context-free tree grammars have been extensively studied in connection with the theory of recursive program scheme. A non-terminal F can be seen as a function name and production rules $F(x_1, \ldots, x_n) \rightarrow t$ define the function. Recursive definitions are allowed since t may contain occurrences of F. Since we know that such recursive definitions may not give the same results depending on the evaluation strategy, IO and OI tree grammars have been introduced to account for such differences.

A context-free grammar is ***IO*** (for innermost-outermost) if we restrict legal derivations to derivations where the innermost terminals are derived first. This restriction corresponds to call-by-value evaluation. A context-free grammar is ***OI*** (for outermost-innermost) if we restrict legal derivations to derivations where the outermost terminals are derived first. This corresponds to call-by-name evaluation. Therefore, given one context-free grammar G, we can define *IO-G* and *OI-G* and the next example shows that the languages generated by these grammars may be different.

Example 2.5.2. Let G be the context-free grammar with axiom *Exp*, non-terminals $\{Exp, Nat, Dup\}$, terminals $\{double, s, 0\}$) and rules

$Exp \to Dup(Nat)$
$Nat \to s(Nat)$
$Nat \to 0$
$Dup(x) \to double(x,x)$
Then outermost-innermost derivations have the form

$$Exp \to Dup(Nat) \to double(Nat, Nat) \xrightarrow{*} double(s^n(0), s^m(0))$$

while innermost-outermost derivations have the form

$$Exp \to Dup(Nat) \xrightarrow{*} Dup(s^n(0)) \to double(s^n(0), s^n(0))$$

Therefore $L(OI\text{-}G) = \{double(s^n(0), s^m(0)) \mid n, m \in \mathbb{N}\}$ and $L(IO\text{-}G) = \{double(s^n(0), s^n(0)) \mid n \in \mathbb{N}\}$.

A tree language L is *IO* if there is some context-free grammar G such that $L = L(IO\text{-}G)$. The next theorem shows the relation between $L(IO\text{-}G)$, $L(OI\text{-}G)$ and $L(G)$.

Theorem 2.5.3. *The following inclusion holds:* $L(IO\text{-}G) \subseteq L(OI\text{-}G) = L(G)$

Example 2.5.2 shows that the inclusion can be strict. *IO*-languages are closed under intersection with regular languages and union, but the closure under concatenation requires another definition of concatenation: all occurrences of a constant generated by a non right-linear rule are replaced by the *same* term, as shown by the next example.

Example 2.5.4. Let G be the context-free grammar with axiom Exp, non-terminals $\{Exp, Nat, Fct\}$, terminals $\{\Box, f(\_,\_,\_)\}$ and rules
$Exp \to Fct(Nat, Nat)$
$Nat \to \Box$
$Fct(x,y) \to f(x,x,y)$
and let $L = IO\text{-}G$ and $M = \{0,1\}$, then $L._{\Box}M$ contains $f(0,0,0)$,$f(0,0,1)$, $f(1,1,0)$, $f(1,1,1)$ but not $f(1,0,1)$ nor $f(0,1,1)$.

There is a lot of work on the extension of results on context-free word grammars and languages to context-free tree grammars and languages. Unfortunately, many constructions and theorem can't be lifted to the tree case. Usually the failure is due to non-linearity which expresses that the same subtrees must occur at different positions in the tree. A similar phenomenon occurred when we stated results on recognizable languages and tree homomorphisms: the inverse image of a recognizable tree language by a tree homorphism is recognizable, but the assumption that the homomorphism is linear is needed to show that the direct image is recognizable.

2.6 Exercises

Exercise 2.1. Let $\mathcal{F} = \{f(,), g(), a\}$. Consider the automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ defined by: $Q = \{q, q_g, q_f\}$, $Q_f = \{q_f\}$, and $\Delta =$

$$\begin{array}{lrclrcl} \{ & a & \to & q(a) & g(q(x)) & \to & q(g(x)) \\ & g(q(x)) & \to & q_g(g(x)) & g(q_g(x)) & \to & q_f(g(x)) \\ & f(q(x),q(y)) & \to & q(f(x,y)) & \}. & & \end{array}$$

Define a regular tree grammar generating $L(\mathcal{A})$.

Exercise 2.2.

1. Prove the equivalence of a regular tree grammar and of the reduced regular tree grammar computed by algorithm of proposition 2.1.3.
2. Let $\mathcal{F} = \{f(,), g(), a\}$. Let G be the regular tree grammar with axiom X, non-terminal A, and rules
$$\begin{aligned} X &\to f(g(A), A) \\ A &\to g(g(A)) \end{aligned}$$
Define a top-down NFTA, a NFTA and a DFTA for $L(G)$. Is it possible to define a top-down DFTA for this language?

Exercise 2.3. Let $\mathcal{F} = \{f(,), a\}$. Let G be the regular tree grammar with axiom X, non-terminals A, B, C and rules

$$\begin{aligned} X &\to C \\ X &\to a \\ X &\to A \\ A &\to f(A, B) \\ B &\to a \end{aligned}$$

Compute the reduced regular tree grammar equivalent to G applying the algorithm defined in the proof of Proposition 2.1.3. Now, consider the same algorithm, but first apply step 2 and then step 1. Is the output of this algorithm reduced? equivalent to G?

Exercise 2.4.

1. Prove Theorem 1.4.3 using regular tree grammars.
2. Prove Theorem 1.4.4 using regular tree grammars.

Exercise 2.5. (Local languages) Let $\mathcal{F}$ be a signature, let t be a term of $T(\mathcal{F})$, then we define $fork(t)$ as follows:

- $fork(a) = \emptyset$, for each constant symbol a;
- $fork(f(t_1, \ldots, t_n)) = \{f(\mathcal{H}ead(t_1), \ldots, \mathcal{H}ead(t_n))\} \cup \bigcup_{i=1}^{i=n} fork(t_i)$

A tree language L is ***local*** if and only if there exist a set $\mathcal{F}' \subseteq \mathcal{F}$ and a set $G \subseteq fork(T(\mathcal{F}))$ such that $t \in L$ iff $root(t) \in \mathcal{F}'$ and $fork(t) \subseteq G$. Prove that every local tree language is a regular tree language. Prove that a language is local iff it is the set of derivation trees of a context-free word language. Prove that every regular tree language is the image of a local tree language by an alphabetic homomorphism.

Exercise 2.6. The pumping lemma for context-free word languages states:

> for each context-free language L, there is some constant $k \geq 1$ such that each $z \in L$ of length greater than or equal to k can be written $z = uvwxy$ such that vx is not the empty word, vwx has length less than or equal to k, and for each $n \geq 0$, the word uv^nwx^ny is in L.

Prove this result using the pumping lemma for tree languages and the results of this chapter.

Exercise 2.7. Another possible definition for the iteration of a language is:

- $L^{0,\,\square} = \{\square\}$
- $L^{n+1,\,\square} = L^{n,\,\square} \cup L^{n,\,\square} \,._{\square}\, L$

(Unfortunately that definition was given in the previous version of TATA)

1. Show that this definition may generate non-regular tree languages. Hint: one binary symbol $f(\ ,\ )$ and $\square$ are enough.

2. Are the two definitions equivalent (i.e. generate the same languages) if Σ consists of unary symbols and constants only?

Exercise 2.8. Let $\mathcal{F}$ be a ranked alphabet, let t be a term of $T(\mathcal{F})$, then we define the word language $Branch(t)$ as follows:

- $Branch(a) = a$, for each constant symbol a;
- $Branch(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{i=n} \{fu \mid u \in Branch(t_i)\}$

Let L be a regular tree language, prove that $Branch(L) = \bigcup_{t \in L} Branch(t)$ is a regular word language. What about the converse?

Exercise 2.9. 1. Let $\mathcal{F}$ be a ranked alphabet such that $\mathcal{F}_0 = \{a, b\}$. Find a regular tree language L such that $Yield(L) = \{a^n b^n \mid n \geq 0\}$. Find a non regular tree language L such that $Yield(L) = \{a^n b^n \mid n \geq 0\}$.

2. Same questions with $Yield(L) = \{u \in \mathcal{F}_0^* \mid |u|_a = |u|_b\}$ where $|u|_a$ (respectively $|u|_b$) denotes the number of a (respectively the number of b) in u.
3. Let $\mathcal{F}$ be a ranked alphabet such that $\mathcal{F}_0 = \{a, b, c\}$, let $A_1 = \{a^n b^n c^p \mid n, p \geq 0\}$, and let $A_2 = \{a^n b^p c^p \mid n, p \geq 0\}$. Find regular tree languages such that $Yield(L_1) = A_1$ and $Yield(L_2) = A_2$. Does there exist a regular tree language such that $Yield(L) = A_1 \cap A_2$.

Exercise 2.10. 1. Let G be the context free word grammar with axiom X, terminals a, b, and rules

$$\begin{aligned} &X \to XX \\ &X \to aXb \\ &X \to \epsilon \end{aligned}$$

where ϵ stands for the empty word. What is the word language $L(G)$? Give a derivation tree for $u = aabbab$.

2. Let G' be the context free word grammar in Greibach normal form with axiom X, non terminals X', Y', Z' terminals a, b, and rules l

$$\begin{aligned} &X' \to aX'Y' \\ &X' \to aY' \\ &X' \to aX'Z' \\ &X' \to aZ' \\ &Y' \to bX' \\ &Z' \to b \end{aligned}$$

prove that $L(G') = L(G)$. Give a derivation tree for $u = aabbab$.

3. Find a context free word grammar G'' such that $L(G'') = A_1 \cup A_2$ (A_1 and A_2 are defined in Exercise 2.9). Give two derivation trees for $u = abc$.

Exercise 2.11. Let $\mathcal{F}$ be a ranked alphabet.

1. Let L and L' be two regular tree languages. Compare the sets $Yield(L \cap L')$ and $Yield(L) \cap Yield(L')$.
2. Let A be a subset of $\mathcal{F}_0$. Prove that $T(\mathcal{F}, A) = T(\mathcal{F} \cap A)$ is a regular tree language. Let L be a regular tree language over $\mathcal{F}$, compare the sets $Yield(L \cap T(\mathcal{F}, A))$ and $Yield(L) \cap Yield(T(\mathcal{F}, A))$.
3. Let R be a regular word language over $\mathcal{F}_0$. Let $T(\mathcal{F}, R) = \{t \in T(\mathcal{F}) \mid Yield(t) \in R\}$. Prove that $T(\mathcal{F}, R)$ is a regular tree language. Let L be a regular tree language over $\mathcal{F}$, compare the sets $Yield(L \cap T(\mathcal{F}, R))$ and $Yield(L) \cap Yield(T(\mathcal{F}, R))$. As a consequence of the results obtained in the present exercise, what could be said about the intersection of a context free word language and of a regular word language?

2.7 Bibliographic notes

This chapter only scratches the topic of tree grammars and related topics. A useful reference on algebraic aspects of regular tree language is [GS84] which contains a lot of classical results on these features. There is a huge litterature on tree grammars and related topics, which is also relevant for the chapter on tree transducers, see the references given in this chapter. Systems of equations can be generalized to formal tree series with similar results [BR82, Boz99, Boz01, Kui99, Kui01]. The notion of pushdown tree automaton has been introduced by Guessarian [Gue83] and generalized to formal tree series by Kuich [Kui01] The reader may consult [Eng82, ES78] for IO and OI grammars. The connection between recursive program scheme and formalisms for regular tree languages is also well-known, see [Cou86] for instance. We should mention that some open problems like equivalence of deterministic tree grammars are now solved using the result of Senizergues on the equivalence of deterministic pushdown word automata [Sén97].

Chapter 3

Logic, Automata and Relations

3.1 Introduction

As early as in the 50s, automata, and in particular tree automata, played an important role in the development of verification. Several well-known logicians, such as A. Church, J.R. Büchi, Elgott, MacNaughton, M. Rabin and others contributed to what is called "the trinity" by Trakhtenbrot: Logic, Automata and Verification (of Boolean circuits).

The idea is simple: given a formula ϕ with free variables $x_1, \ldots, x_n$ and a domain of interpretation D, ϕ defines the subset of D^n containing all assignments of the free variables $x_1, \ldots, x_n$ that satisfy ϕ. Hence formulas in this case are just a way of defining subsets of D^n (also called n-ary relations on D). In case $n = 1$ (and, as we will see, also for $n > 1$), finite automata provide another way of defining subsets of D^n. In 1960, Büchi realized that these two ways of defining relations over the free monoid $\{0, \ldots, n\}^*$ coincide when the logic is the *sequential calculus*, also called *weak second-order monadic logic with one successor*, WS1S. This result was extended to tree automata: Doner, Thatcher and Wright showed that the definability in the weak second-order monadic logic with k successors, WSkS coincides with the recognizability by a finite tree automaton. These results imply in particular the decidability of WSkS, following the decision results on tree automata (see chapter 1).

These ideas are the basis of several decision techniques for various logics some of which will be listed in Section 3.4. In order to illustrate this correspondence, consider Presburger's arithmetic: the atomic formulas are equalities and inequalities $s = t$ or $s \geq t$ where s, t are sums of variables and constants. For instance $x+y+y = z+z+z+1+1$, also written $x+2y = 3z+2$, is an atomic formula. In other words, atomic formulas are linear Diophantine (in)equations. Then atomic formulas can be combined using any logical connectives among $\wedge, \vee, \neg$ and quantifications $\forall, \exists$. For instance $\forall x.(\forall y.\neg(x = 2y)) \Rightarrow (\exists y.x = 2y + 1)$ is a (true) formula of Presburger's arithmetic. Formulas are interpreted in the natural numbers (non-negative integers), each symbol having its expected meaning. A *solution* of a formula $\phi(x)$ whose only free variable is x, is an assignment of x to a natural number n such that $\phi(n)$ holds true in the interpretation. For

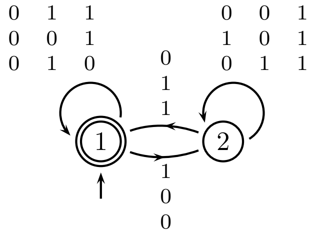


Figure 3.1: The automaton which accepts the solutions of $x = y + z$

instance, if $\phi(x)$ is the formula $\exists y.x = 2y$, its solutions are the even numbers.

Writing integers in base 2, they can be viewed as elements of the free monoid $\{0, 1\}^*$, i.e. words of 0s and 1s. The representation of a natural number is not unique as $01 = 1$, for instance. Tuples of natural numbers are displayed by stacking their representations in base 2 and aligning on the right, then completing with some 0s on the left in order to get a rectangle of bits. For instance the pair (13,6) is represented as $\begin{smallmatrix}1&1&0&1\\0&1&1&0\end{smallmatrix}$ (or $\begin{smallmatrix}0&1&1&0&1\\0&0&1&1&0\end{smallmatrix}$ as well). Hence, we can see the solutions of a formula as a subset of $(\{0, 1\}^n)^*$ where n is the number of free variables of the formula.

It is not difficult to see that the set of solutions of any atomic formula is recognized by a finite word automaton working on the alphabet $\{0, 1\}^n$. For instance, the solutions of $x = y + z$ are recognized by the automaton of Figure 3.1.

Then, and that is probably one of the key ideas, each logical connective corresponds to a basic operation on automata (here word automata): $\vee$ is a union, $\wedge$ and intersection, $\neg$ a complement, $\exists x$ a *projection* (an operation which will be defined in Section 3.2.4). It follows that the set of solutions of any Presburger formula is recognized by a finite automaton.

In particular, a closed formula (without free variable), holds true in the interpretation if the initial state of the automaton is also final. It holds false otherwise. Therefore, this gives both a decision technique for Presburger formulas by computing automata and an effective representation of the set of solutions for open formulas.

The example of Presburger's arithmetic we just sketched is not isolated. That is one of the purposes of this chapter to show how to relate finite tree automata and formulas.

In general, the problem with these techniques is to design an appropriate notion of automaton, which is able to recognize the solutions of atomic formulas and which has the desired closure and decision properties. We have to cite here the famous *Rabin automata* which work on infinite trees and which have indeed the closure and decidability properties, allowing to decide the full second-order monadic logic with k successors (a result due to M. Rabin, 1969). It is however out of the scope of this book to survey automata techniques in logic and computer science. We restrict our attention to finite automata on finite trees and refer to the excellent surveys [Rab77, Tho90] for more details on other applications of automata to logic.

We start this chapter by reviewing some possible definitions of automata on pairs (or, more generally, tuples) of finite trees in Section 3.2. We define in this way several notions of recognizability for relations, which are not necessary unary, extending the frame of chapter 1. This extension is necessary since, automata recognizing the solutions of formulas actually recognize n-tuples of solutions, if there are n free variables in the formula.

The most natural way of defining a notion of recognizability on tuples is to consider products of recognizable sets. Though this happens to be sometimes sufficient, this notion is often too weak. For instance the example of Figure 3.1 could not be defined as a product of recognizable sets. Rather, we stacked the words and recognized these codings. Such a construction can be generalized to trees (we have to overlap instead of stacking) and gives rise to a second notion of recognizability. We will also introduce a third class called "Ground Tree Transducers" which is weaker than the second class above but enjoys stronger closure properties, for instance by iteration. Its usefulness will become evident in Section 3.4.

Next, in Section 3.3, we introduce the weak second-order monadic logic with k successor and show Thatcher and Wright's theorem which relates this logic with finite tree automata. This is a modest insight into the relations between logic and automata.

Finally in Section 3.4 we survey a number of applications, mostly issued from Term Rewriting or Constraint Solving. We do not detail this part (we give references instead). The goal is to show how the simple techniques developed before can be applied to various questions, with a special emphasis on decision problems. We consider the theories of *sort constraints* in Section 3.4.1, the theory of *linear encompassment* in Section 3.4.2, the theory of ground term rewriting in Section 3.4.3 and reduction strategies in orthogonal term rewriting in Section 3.4.4. Other examples are given as exercises in Section 3.5 or considered in chapters 4 and 5.

3.2 Automata on Tuples of Finite Trees

3.2.1 Three Notions of Recognizability

Let $Rec_\times$ be the subset of n-ary relations on $T(\mathcal{F})$ which are finite unions of products $S_1 \times \cdots \times S_n$ where $S_1, \ldots, S_n$ are recognizable subsets of $T(\mathcal{F})$. This notion of recognizability of pairs is the simplest one can imagine. Automata for such relations consist of pairs of tree automata which work independently. This notion is however quite weak, as e.g. the diagonal

$$\Delta = \{(t,t) \mid t \in T(\mathcal{F})\}$$

does not belong to $Rec_\times$. Actually a relation $R \in Rec_\times$ does not really relate its components!

The second notion of recognizability is used in the correspondence with WSkS and is strictly stronger than the above one. Roughly, it consists in overlapping the components of a n-tuple, yielding a term on a product alphabet. Then define *Rec* as the set of sets of pairs of terms whose overlapping coding is recognized by a tree automaton on the product alphabet.

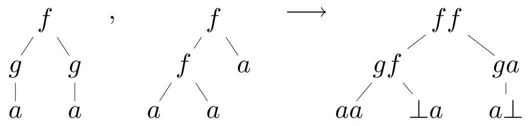


Figure 3.2: The overlap of two terms

Let us first define more precisely the notion of "coding". (This is illustrated by an example on Figure 3.2). We let $\mathcal{F}' = (\mathcal{F} \cup \{\bot\})^n$, where $\bot$ is a new symbol. This is the idea of "stacking" the symbols, as in the introductory example of Presburger's arithmetic. Let k be the maximal arity of a function symbol in $\mathcal{F}$. Assuming $\bot$ has arity 0, the arities of function symbols in $\mathcal{F}'$ are defined by $a(f_1 \dots f_n) = \max(a(f_1), \dots, a(f_n))$.

The coding of two terms $t_1, t_2 \in T(\mathcal{F})$ is defined by induction:

$$[f(t_1, \dots, t_n), g(u_1, \dots, u_m)] \stackrel{\text{def}}{=} fg([t_1, u_1], \dots [t_m, u_m], [t_{m+1}, \bot], \dots, [t_n, \bot])$$

if $n \geq m$ and

$$[f(t_1, \dots, t_n), g(u_1, \dots, u_m)] \stackrel{\text{def}}{=} fg([t_1, u_1], \dots [t_n, u_n], [\bot, u_{n+1}], \dots, [\bot, u_m])$$

if $m \geq n$.

More generally, the coding of n terms $f_1(t_1^1, \dots, t_1^{k_1}), \dots, f_n(t_1^n, \dots, t_n^{k_n})$ is defined as

$$f_1 \dots f_n([t_1^1, \dots, t_n^1], \dots, [t_1^m, \dots, t_n^m])$$

where m is the maximal arity of $f_1, \dots, f_n \in \mathcal{F}$ and t_i^j is, by convention, $\bot$ when $j > k_i$.

Definition 3.2.1. *Rec is the set of relations $R \subseteq T(\mathcal{F})^n$ such that*

$$\{[t_1, \dots, t_n] \mid (t_1, \dots, t_n) \in R\}$$

is recognized by a finite tree automaton on the alphabet $\mathcal{F}' = (\mathcal{F} \cup \{\bot\})^n$.

For example, consider the diagonal Δ, it is in *Rec* since its coding is recognized by the bottom-up tree automaton whose only state is q (also a final state) and transitions are the rules $ff(q, \dots, q) \to q$ for all symbols $f \in \mathcal{F}$.

One drawback of this second notion of recognizability is that it is not closed under iteration. More precisely, there is a binary relation R which belongs to *Rec* and whose transitive closure is not in *Rec* (see Section 3.5). For this reason, a third class of recognizable sets of pairs of trees was introduced: the *Ground Tree Transducers* (GTT for short).

Definition 3.2.2. *A GTT is a pair of bottom-up tree automata $(\mathcal{A}_1, \mathcal{A}_2)$ working on the same alphabet. Their sets of states may share some symbols (the* synchronization states*).*

A pair (t, t') is recognized by a GTT $(\mathcal{A}_1, \mathcal{A}_2)$ if there is a context $C \in \mathcal{C}^n(\mathcal{F})$ such that $t = C[t_1, \dots, t_n]$, $t' = C[t'_1, \dots, t'_n]$ and there are states $q_1, \dots, q_n$ of both automata such that, for all i, $t_i \xrightarrow[\mathcal{A}_1]{*} q_i$ and $t'_i \xrightarrow[\mathcal{A}_2]{*} q_i$. We write $L(\mathcal{A}_1, \mathcal{A}_2)$ the language accepted by the GTT $(\mathcal{A}_1, \mathcal{A}_2)$, i.e. the set of pairs of terms which are recognized.

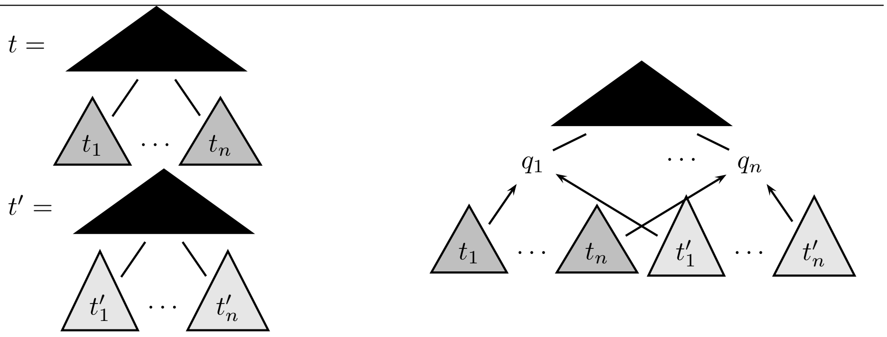


Figure 3.3: GTT acceptance

The recognizability by a GTT is depicted on Figure 3.3. For instance, Δ is accepted by a GTT. Another typical example is the binary relation "one step parallel rewriting" for term rewriting system whose left members are linear and whose right hand sides are ground (see Section 3.4.3).

3.2.2 Examples of The Three Notions of Recognizability

The first example illustrates $Rec_\times$. It will be developed in a more general framework in Section 3.4.2.

Example 3.2.3. Consider the alphabet $\mathcal{F} = \{f(,), g(), a\}$ where f is binary, g is unary and a is a constant. Let P be the predicate which is true on t if there are terms t_1, t_2 such that $f(g(t_1), t_2)$ is a subterm of t. Then the solutions of $P(x) \wedge P(y)$ define a relation in $Rec_\times$, using twice the following automaton:

$$\begin{array}{lclrclrcl}
Q & = & \{q_f, q_g, q_\top\} & & & & & & \\
Q_f & = & \{q_f\} & & & & & & \\
T & = & \{ & a & \rightarrow & q_\top & f(q_\top, q_\top) & \rightarrow & q_\top \\
 & & & g(q_\top) & \rightarrow & q_\top & f(q_f, q_\top) & \rightarrow & q_f \\
 & & & g(q_f) & \rightarrow & q_f & f(q_g, q_\top) & \rightarrow & q_f \\
 & & & g(q_\top) & \rightarrow & q_g & f(q_\top, q_f) & \rightarrow & q_f\}
\end{array}$$

For instance the pair $(g(f(g(a), g(a))), f(g(g(a)), a))$ is accepted by the pair of automata.

The second example illustrates *Rec*. Again, it is a first account of the developments of Section 3.4.4.

Example 3.2.4. Let $\mathcal{F} = \{f(,), g(), a, \Omega\}$ where f is binary, g is unary, a and Ω are constants. Let R be the set of terms (t, u) such that u can be obtained from t by replacing each occurrence of Ω by some term in $T(\mathcal{F})$ (each occurrence of Ω needs not to be replaced with the same term). Using the notations of Chapter 2

$$R(t, u) \iff u \in t ._\Omega T(\mathcal{F})$$

R is recognized by the following automaton (on codings of pairs):

$$\begin{array}{lcl rcl rcl}
Q & = & \{q, q'\} & & & & & & \\
Q_f & = & \{q'\} & & & & & & \\
T & = & \{ & \perp a & \rightarrow & q & \perp f(q,q) & \rightarrow & q \\
 & & & \perp g(q) & \rightarrow & q & \Omega f(q,q) & \rightarrow & q' \\
 & & & \perp \Omega & \rightarrow & q & ff(q',q') & \rightarrow & q' \\
 & & & aa & \rightarrow & q' & gg(q') & \rightarrow & q' \\
 & & & \Omega\Omega & \rightarrow & q' & \Omega g(q) & \rightarrow & q' \\
 & & & \Omega a & \rightarrow & q'\} & & &
\end{array}$$

For instance, the pair $(f(g(\Omega), g(\Omega)), f(g(g(a)), g(\Omega)))$ is accepted by the automaton: the overlap of the two terms yields

$$[tu] = ff(gg(\Omega g(\perp a)), gg(\Omega\Omega))$$

And the reduction:

$$\begin{array}{rcl}
[tu] & \xrightarrow{*} & ff(gg(\Omega g(q)), gg(q')) \\
 & \xrightarrow{*} & ff(gg(q'), q') \\
 & \rightarrow & ff(q', q') \\
 & \rightarrow & q'
\end{array}$$

The last example illustrates the recognition by a GTT. It comes from the theory of rewriting; further developments and explanations on this theory are given in Section 3.4.3.

Example 3.2.5. Let $\mathcal{F} = \{\times, +, 0, 1\}$. Let $\mathcal{R}$ be the rewrite system $0 \times x \rightarrow 0$. The many-steps reduction relation defined by $\mathcal{R}$: $\xrightarrow[\mathcal{R}]{*}$ is recognized by the GTT$(\mathcal{A}_1, \mathcal{A}_2)$ defined as follows (+ and × are used in infix notation to meet their usual reading):

$$\begin{array}{lcl lcl rcl}
T_1 & = & \{ & 0 & \rightarrow & q_\top & q_\top + q_\top & \rightarrow & q_\top \\
 & & & 1 & \rightarrow & q_\top & q_\top \times q_\top & \rightarrow & q_\top \\
 & & & 0 & \rightarrow & q_0 & q_0 \times q_\top & \rightarrow & q_0\} \\
T_2 & = & \{ & 0 & \rightarrow & q_0\} & & &
\end{array}$$

Then, for instance, the pair $(1 + ((0 \times 1) \times 1), 1 + 0)$ is accepted by the GTT since

$$1 + ((0 \times 1) \times 1) \xrightarrow[\mathcal{A}_1]{*} 1 + (q_0 \times q_\top) \times q_\top \xrightarrow[\mathcal{A}_1]{} 1 + (q_0 \times q_\top) \xrightarrow[\mathcal{A}_1]{} 1 + q_0$$

on one hand and $1 + 0 \xrightarrow[\mathcal{A}_2]{} 1 + q_0$ on the other hand.

3.2.3 Comparisons Between the Three Classes

We study here the inclusion relations between the three classes: $Rec_\times, Rec, GTT$.

Proposition 3.2.6. *$Rec_\times \subset Rec$ and the inclusion is strict.*

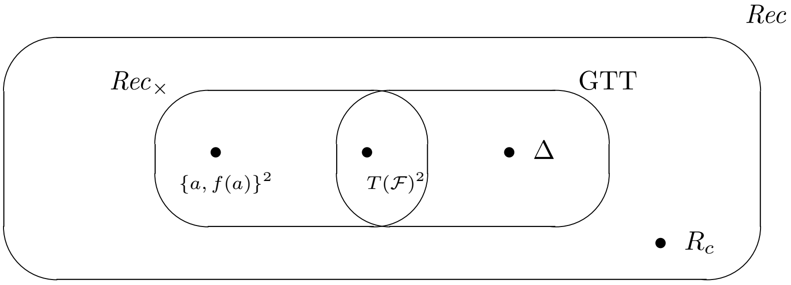


Figure 3.4: The relations between the three classes

Proof. To show that any relation in $Rec_\times$ is also in Rec, we have to construct from two automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_1^f, R_1), \mathcal{A}_2 = (\mathcal{F}, Q_2, Q_2^f, R_2)$ an automaton which recognizes the overlaps of the terms in the languages. We define such an automaton $\mathcal{A} = (Q, (\mathcal{F} \cup \{\perp\})^2, Q^f, R)$ by: $Q = (Q_1 \cup \{q_\perp\}) \times (Q_2 \cup \{q_\perp\})$, $Q^f = Q_1^f \times Q_2^f$ and R is the set of rules:

- $f \perp ((q_1, q_\perp), \ldots, (q_n, q_\perp)) \;\rightarrow\; (q, q_\perp)$ if $f(q_1, \ldots, q_n) \rightarrow q \in R_1$
- $\perp f((q_\perp, q_1), \ldots, (q_\perp, q_n)) \;\rightarrow\; (q_\perp, q)$ if $f(q_1, \ldots, q_n) \rightarrow q \in R_2$
- $fg((q_1, q_1'), \ldots, (q_m, q_m'), (q_{m+1}, q_\perp), \ldots, (q_n, q_\perp)) \rightarrow (q, q')$ if $f(q_1, \ldots, q_n) \rightarrow q \in R_1$ and $g(q_1', \ldots, q_m') \rightarrow q' \in R_2$ and $n \geq m$
- $fg((q_1, q_1'), \ldots, (q_n, q_n'), (q_\perp, q_{n+1}), \ldots, (q_\perp, q_m)) \rightarrow (q, q')$ if $f(q_1, \ldots, q_n) \rightarrow q \in R_1$ and $g(q_1', \ldots, q_m') \rightarrow q' \in R_2$ and $m \geq n$

The proof that $\mathcal{A}$ indeed accepts $L(\mathcal{A}_1) \times L(\mathcal{A}_2)$ is left to the reader.

Now, the inclusion is strict since e.g. $\Delta \in Rec \setminus Rec_\times$. □

Proposition 3.2.7. *GTT $\subset$ Rec and the inclusion is strict.*

Proof. Let $(\mathcal{A}_1, \mathcal{A}_2)$ be a GTT accepting R. We have to construct an automaton $\mathcal{A}$ which accepts the codings of pairs in R.

Let $\mathcal{A}_0 = (Q_0, \mathcal{F}, Q_0^f, T_0)$ be the automaton constructed in the proof of Proposition 3.2.6. $[t, u] \xrightarrow[\mathcal{A}_0]{*} (q_1, q_2)$ if and only if $t \xrightarrow[\mathcal{A}_1]{*} q_1$ and $u \xrightarrow[\mathcal{A}_2]{*} q_2$. Now we let $\mathcal{A} = (Q_0 \cup \{q_f\}, \mathcal{F}, Q_f = \{q_f\}, T)$. T consists of T_0 plus the following rules:

$$(q, q) \quad \rightarrow \quad q_f \qquad ff(q_f, \ldots, q_f) \quad \rightarrow \quad q_f$$

For every symbol $f \in F$ and every state $q \in Q_0$.

If (t, u) is accepted by the GTT, then

$$t \xrightarrow[\mathcal{A}_1]{*} C[q_1, \ldots, q_n]_{p_1, \ldots, p_n} \xleftarrow[\mathcal{A}_2]{*} u.$$

Then

$$[t, u] \xrightarrow[\mathcal{A}_0]{*} [C, C][(q_1, q_1), \ldots, (q_n, q_n)]_{p_1, \ldots, p_n} \xrightarrow[\mathcal{A}]{*} [C, C][q_f, \ldots, q_f]_{p_1, \ldots, p_n} \xrightarrow[\mathcal{A}]{*} q_f$$

Conversely, if $[t, u]$ is accepted by $\mathcal{A}$ then $[t, u] \xrightarrow[\mathcal{A}]{*} q_f$. By definition of $\mathcal{A}$, there should be a sequence:

$$[t, u] \xrightarrow[\mathcal{A}]{*} C[(q_1, q_1), \ldots, (q_n, q_n)]_{p_1,\ldots,p_n} \xrightarrow[\mathcal{A}]{*} C[q_f, \ldots, q_f]_{p_1,\ldots,p_n} \xrightarrow[\mathcal{A}]{*} q_f$$

Indeed, we let p_i be the positions at which one of the ϵ-transitions steps $(q, q) \rightarrow q_f$ is applied. $(n \geq 0)$. Now, $C[q_f, \ldots, q_f]_{p_1,\ldots,p_m} q_f$ if and only if C can be written $[C_1, C_1]$ (the proof is left to the reader).

Concerning the strictness of the inclusion, it will be a consequence of Propositions 3.2.6 and 3.2.8. ☐

Proposition 3.2.8. *GTT* $\not\subseteq$ *Rec*$_\times$ *and Rec*$_\times$ $\not\subseteq$ *GTT.*

Proof. Δ is accepted by a GTT (with no state and no transition) but it does not belong to *Rec*$_\times$. On the other hand, if $\mathcal{F} = \{f, a\}$, then $\{a, f(a)\}^2$ is in *Rec*$_\times$ (it is the product of two finite languages) but it is not accepted by any GTT since any GTT accepts at least Δ. ☐

Finally, there is an example of a relation R_c which is in *Rec* and not in the union *Rec*$_\times \cup$ GTT; consider for instance the alphabet $\{a(), b(), 0\}$ and the one step reduction relation associated with the rewrite system $a(x) \rightarrow x$. In other words,

$$(u, v) \in R_c \Longleftrightarrow \exists C \in \mathcal{C}(\mathcal{F}), \exists t \in T(\mathcal{F}), u = C[a(t)] \wedge v = C[t]$$

It is left as an exercise to prove that $R_c \in$ *Rec* $\setminus$ (*Rec*$_\times \cup$ GTT).

3.2.4 Closure Properties for *Rec*$_\times$ and *Rec*; Cylindrification and Projection

Let us start with the classical closure properties.

Proposition 3.2.9. *Rec*$_\times$ *and Rec are closed under Boolean operations.*

The proof of this proposition is straightforward and left as an exercise.

These relations are also closed under *cylindrification* and *projection*. Let us first define these operations which are specific to automata on tuples:

Definition 3.2.10. *If* $R \subseteq T(\mathcal{F})^n$ $(n \geq 1)$ *and* $1 \leq i \leq n$ *then the* ith projection *of* R *is the relation* $R_i \subseteq T(\mathcal{F})^{n-1}$ *defined by*

$$R_i(t_1, \ldots, t_{n-1}) \Leftrightarrow \exists t \in T(\mathcal{F}) \;\; R(t_1, \ldots, t_{i-1}, t, t_i, \ldots, t_{n-1})$$

When $n = 1$, $T(\mathcal{F})^{n-1}$ is by convention a singleton set $\{\top\}$ (so as to keep the property that $T(\mathcal{F})^{n+1} = T(\mathcal{F}) \times T(\mathcal{F})^n$). $\{\top\}$ is assumed to be a neutral element w.r.t. Cartesian product. In such a situation, a relation $R \subseteq T(\mathcal{F})^0$ is either $\emptyset$ or $\{\top\}$ (it is a propositional variable).

Definition 3.2.11. *If* $R \subseteq T(\mathcal{F})^n$ $(n \geq 0)$ *and* $1 \leq i \leq n + 1$, *then the* ith cylindrification *of* R *is the relation* $R^i \subseteq T(\mathcal{F})^{n+1}$ *defined by*

$$R^i(t_1, \ldots, t_{i-1}, t, t_i, \ldots, t_n) \Leftrightarrow R(t_1, \ldots, t_{i-1}, t_i, \ldots, t_n)$$

Proposition 3.2.12. *$Rec_\times$ and Rec are effectively closed under projection and cylindrification. Actually, ith projection can be computed in linear time and the ith cylindrification of $\mathcal{A}$ can be computed in linear time (assuming that the size of the alphabet is constant).*

Proof. For $Rec_\times$, this property is easy: projection on the ith component simply amounts to remove the ith automaton. Cylindrification on the ith component simply amounts to insert as a ith automaton, an automaton accepting all terms.

Assume that $R \in Rec$. The ith projection of R is simply its image by the following linear tree homomorphism:

$$h_i([f_1,\ldots,f_n](t_1,\ldots,t_k)) \stackrel{\text{def}}{=} [f_1\ldots f_{i-1}f_{i+1}\ldots f_n](h_i(t_1),\ldots,h_i(t_m))$$

in which m is the arity of $[f_1\ldots f_{i-1}f_{i+1}\ldots f_n]$ (which is smaller or equal to k). Hence, by Theorem 1.4.3, the ith projection of R is recognizable (and we can extract from the proof a linear construction of the automaton).

Similarly, the ith cylindrification is obtained as an inverse homomorphic image, hence is recognizable thanks to Theorem 1.4.4.

□

Note that using the above construction, the projection of a deterministic automaton may be non-deterministic (see exercises)

Example 3.2.13. Let $\mathcal{F} = \{f, g, a\}$ where f is binary, g is unary and a is a constant. Consider the following automaton $\mathcal{A}$ on $\mathcal{F}' = (\mathcal{F} \cup \{\perp\})^2$: The set of states is $\{q_1, q_2, q_3, q_4, q_5\}$ and the set of final states is $\{q_3\}$[1]

$$\begin{array}{rcl@{\qquad}rcl}
a\perp & \rightarrow & q_1 & f\perp(q_1,q_1) & \rightarrow & q_1\\
g\perp(q_1) & \rightarrow & q_1 & fg(q_2,q_1) & \rightarrow & q_3\\
ga(q_1) & \rightarrow & q_2 & f\perp(q_4,q_1) & \rightarrow & q_4\\
g\perp(q_1) & \rightarrow & q_4 & fa(q_4,q_1) & \rightarrow & q_2\\
gg(q_3) & \rightarrow & q_3 & ff(q_3,q_3) & \rightarrow & q_3\\
aa & \rightarrow & q_5 & ff(q_3,q_5) & \rightarrow & q_3\\
gg(q_5) & \rightarrow & q_5 & ff(q_5,q_3) & \rightarrow & q_3\\
ff(q_5,q_5) & \rightarrow & q_5 & & &
\end{array}$$

The first projection of this automaton gives:

$$\begin{array}{rcl@{\qquad}rcl}
a & \rightarrow & q_2 & g(q_3) & \rightarrow & q_3\\
a & \rightarrow & q_5 & g(q_5) & \rightarrow & q_5\\
g(q_2) & \rightarrow & q_3 & f(q_3,q_3) & \rightarrow & q_3\\
f(q_3,q_5) & \rightarrow & q_3 & f(q_5,q_5) & \rightarrow & q_5\\
f(q_5,q_3) & \rightarrow & q_3 & & &
\end{array}$$

Which accepts the terms containing $g(a)$ as a subterm[2].

---

[1] This automaton accepts the set of pairs of terms (u, v) such that u can be rewritten in one or more steps to v by the rewrite system $f(g(x), y) \rightarrow g(a)$.

[2] i.e. the terms that are obtained by applying at least one rewriting step using $f(g(x), y) \rightarrow g(a)$

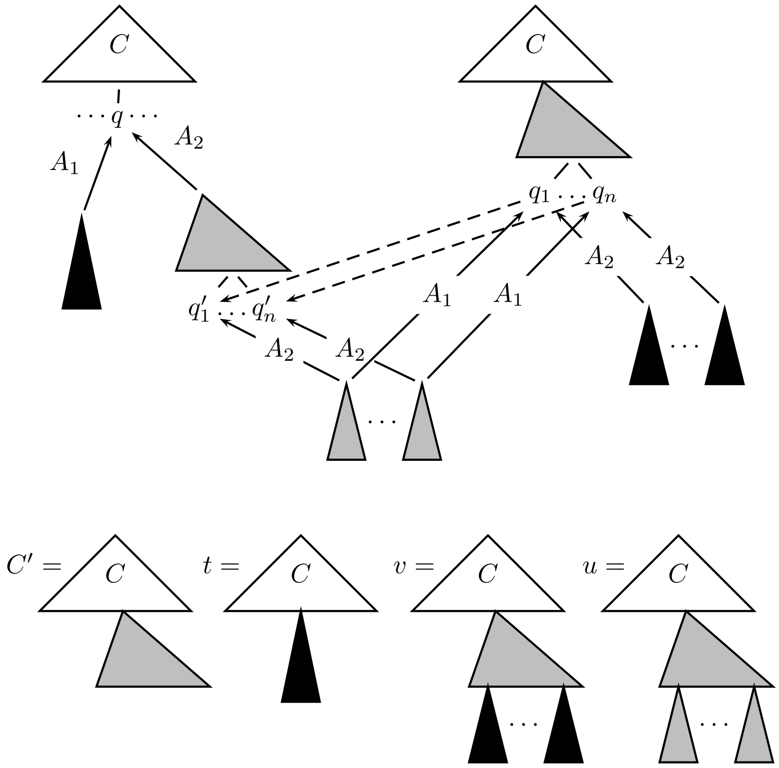


Figure 3.5: The proof of Theorem 3.2.14

3.2.5 Closure of GTT by Composition and Iteration

Theorem 3.2.14. *If $R \subseteq T(\mathcal{F})^2$ is recognized by a GTT, then its transitive closure R^* is also recognized by a GTT.*

The detailed proof is technical, so let us show it on a picture and explain it informally.

We consider two terms (t, v) and (v, u) which are both accepted by the GTT and we wish that (t, u) is also accepted. For simplicity, consider only one state q such that $t \xrightarrow[\mathcal{A}_1]{*} C[q] \xleftarrow[\mathcal{A}_2]{*} v$ and $v \xrightarrow[\mathcal{A}_1]{*} C'[q_1, \dots, q_n] \xleftarrow[\mathcal{A}_2]{*} u$. There are actually two cases: C can be "bigger" than C' or "smaller". Assume it is smaller. Then q is reached at a position inside C': $C' = C[C'']_p$. The situation is depicted on Figure 3.5. Along the reduction of v to q by $\mathcal{A}_2$, we enter a configuration $C''[q'_1, \dots, q'_n]$. The idea now is to add to $\mathcal{A}_2$ ϵ-transitions from q_i to q'_i. In this way, as can easily be seen on Figure 3.5, we get a reduction from u to $C[q]$, hence the pair (t, u) is accepted.

Proof. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the pair of automata defining the GTT which accepts R. We compute by induction the automata $\mathcal{A}_1^n, \mathcal{A}_2^n$. $\mathcal{A}_i^0 = \mathcal{A}_i$ and $\mathcal{A}_i^{n+1}$ is obtained by adding new transitions to $\mathcal{A}_i^n$: Let Q_i be the set of states of $\mathcal{A}_i$ (and also the set of states of $\mathcal{A}_i^n$).

- If $L_{\mathcal{A}_2^n}(q) \cap L_{\mathcal{A}_1^n}(q') \neq \emptyset$, $q \in Q_1 \cap Q_2$ and $q \not\xrightarrow[\mathcal{A}_1^n]{*} q'$, then $\mathcal{A}_1^{n+1}$ is obtained from $\mathcal{A}_1^n$ by adding the ϵ-transition $q \rightarrow q'$ and $\mathcal{A}_2^{n+1} = \mathcal{A}_2^n$.
- If $L_{\mathcal{A}_1^n}(q) \cap L_{\mathcal{A}_2^n}(q') \neq \emptyset$, $q \in Q_1 \cap Q_2$ and $q \not\xrightarrow[\mathcal{A}_2^n]{*} q'$, then $\mathcal{A}_2^{n+1}$ is obtained from $\mathcal{A}_2^n$ by adding the ϵ-transition $q \rightarrow q'$ and $\mathcal{A}_1^{n+1} = \mathcal{A}_1^n$.

If there are several ways of obtaining $\mathcal{A}_i^{n+1}$ from $\mathcal{A}_i^n$ using these rules, we don't care which of these ways is used.

First, these completion rules are decidable by the decision properties of chapter 1. Their application also terminates as at each application strictly decreases $k_1(n) + k_2(n)$ where $k_i(n)$ is the number of pairs of states $(q, q') \in (Q_1 \cup Q_2) \times (Q_1 \cup Q_2)$ such that there is no ϵ-transition in $\mathcal{A}_i^n$ from q to q'. We let $\mathcal{A}_i^*$ be a fixed point of this computation. We show that $(\mathcal{A}_1^*, \mathcal{A}_2^*)$ defines a GTT accepting R^*.

- Each pair of terms accepted by the GTT $(\mathcal{A}_1^*, \mathcal{A}_2^*)$ is in R^*: we show by induction on n that each pair of terms accepted by the GTT $(\mathcal{A}_1^n, \mathcal{A}_2^n)$ is in R^*. For $n = 0$, this follows from the hypothesis. Let us now assume that $\mathcal{A}_1^{n+1}$ is obtained by adding $q \rightarrow q'$ to the transitions of $\mathcal{A}_1^n$ (The other case is symmetric). Let (t, u) be accepted by the GTT $(\mathcal{A}_1^{n+1}, \mathcal{A}_2^{n+1})$. By definition, there is a context C and positions $p_1, \ldots, p_k$ such that $t = C[t_1, \ldots, t_k]_{p_1,\ldots,p_k}$, $u = C[u_1, \ldots, u_k]_{p_1,\ldots,p_k}$ and there are states $q_1, \ldots, q_k \in Q_1 \cap Q_2$ such that, for all i, $t_i \xrightarrow[\mathcal{A}_1^{n+1}]{*} q_i$ and $u_i \xrightarrow[\mathcal{A}_2^n]{*} q_i$.

  We prove the result by induction on the number m of times $q \rightarrow q'$ is applied in the reductions $t_i \xrightarrow[\mathcal{A}_1^{n+1}]{*} q_i$. If $m = 0$. Then this is the first induction hypothesis: (t, u) is accepted by $(\mathcal{A}_1^n, \mathcal{A}_2^n)$, hence $(t, u) \in R^*$. Now, assume that, for some i,

  $$t_i \xrightarrow[\mathcal{A}_1^{n+1}]{*} t_i'[q]_p \xrightarrow[q \rightarrow q']{*} t_i'[q']_p \xrightarrow[\mathcal{A}_1^n]{*} q_i$$

  By definition, there is a term v such that $v \xrightarrow[\mathcal{A}_2^n]{*} q$ and $v \xrightarrow[\mathcal{A}_1^n]{*} q'$. Hence

  $$t_i[v]_p \xrightarrow[\mathcal{A}_1^{n+1}]{*} q_i$$

  And the number of reduction steps using $q \rightarrow q'$ is strictly smaller here than in the reduction from t_i to q_i. Hence, by induction hypothesis, $(t[v]_{p_ip}, u) \in R^*$. On the other hand, $(t, t[v]_{p_ip})$ is accepted by the GTT $(\mathcal{A}_1^{n+1}, \mathcal{A}_2^n)$ since $t|_{p_ip} \xrightarrow[\mathcal{A}_1^{n+1}]{*} q$ and $v \xrightarrow[\mathcal{A}_2^n]{*} q$. Moreover, by construction, the first sequence of reductions uses strictly less than m times the transition $q \rightarrow q'$. Then, by induction hypothesis, $(t, t[v]_{p_ip}) \in R^*$. Now from $(t, t[v]_{p_ip}) \in R^*$ and $(t[v]_{p_ip}, u) \in R^*$, we conclude $(t, u) \in R^*$.
- If $(t, u) \in R^*$, then (t, u) is accepted by the GTT $(\mathcal{A}_1^*, \mathcal{A}_2^*)$. Let us prove the following intermediate result:

Lemma 3.2.15.

$$\text{If} \left\{ \begin{array}{l} t \xrightarrow[\mathcal{A}_1^*]{*} q \\ v \xrightarrow[\mathcal{A}_2^*]{*} q \\ v \xrightarrow[\mathcal{A}_1^*]{*} C[q_1, \dots, q_k]_{p_1,\dots,p_k} \\ u \xrightarrow[\mathcal{A}_2^*]{*} C[q_1, \dots, q_k]_{p_1,\dots,p_k} \end{array} \right\} \text{then } u \xrightarrow[\mathcal{A}_2^*]{*} q$$

and hence (t, u) *is accepted by the GTT.*

Let $v \xrightarrow[\mathcal{A}_2^*]{*} C[q_1', \dots, q_k']_{p_1,\dots,p_k} \xrightarrow[\mathcal{A}_2^*]{*} q$. For each i, $v|_{p_i} \in L_{\mathcal{A}_2^*}(q_i') \cap L_{\mathcal{A}_1^*}(q_i)$ and $q_i \in Q_1 \cap Q_2$. Hence, by construction, $q_i \xrightarrow[\mathcal{A}_2^*]{} q_i'$. It follows that

$$u \xrightarrow[\mathcal{A}_2^*]{*} C[q_1, \dots q_k]_{p_1,\dots,p_k} \xrightarrow[\mathcal{A}_2^*]{*} C[q_1', \dots, q_k']_{p_1,\dots,p_k} \xrightarrow[\mathcal{A}_2^*]{*} q$$

Which proves our lemma.

Symmetrically, if $t \xrightarrow[\mathcal{A}_1^*]{*} C[q_1, \dots, q_k]_{p_1,\dots,p_k}$, $v \xrightarrow[\mathcal{A}_2^*]{*} C[q_1, \dots, q_k]_{p_1,\dots,p_k}$, $v \xrightarrow[\mathcal{A}_1^*]{*} q$ and $u \xrightarrow[\mathcal{A}_2^*]{*} q$, then $t \xrightarrow[\mathcal{A}_1^*]{*} q$

Now, let $(t, u) \in R^n$: we prove that (t, u) is accepted by the GTT $(\mathcal{A}_1^*, \mathcal{A}_2^*)$ by induction on n. If $n = 1$, then the result follows from the inclusion of $L(\mathcal{A}_1, \mathcal{A}_2)$ in $L(\mathcal{A}_1^*, \mathcal{A}_2^*)$. Now, let v be such that $(t, v) \in R$ and $(v, u) \in R^{n-1}$. By induction hypothesis, both (t, v) and (v, u) are accepted by the GTT $(\mathcal{A}_1^*, \mathcal{A}_2^*)$: there are context C and C' and positions $p_1, \dots, p_k, p_1', \dots, p_m'$ such that

$$t = C[t_1, \dots, t_k]_{p_1,\dots,p_k},\ v = C[v_1, \dots, v_k]_{p_1,\dots,p_k}$$

$$v = C'[v_1', \dots, v_m']_{p_1',\dots,p_m'},\ u = C'[u_1, \dots, u_m]$$

and states $q_1, \dots, q_k, q_1', \dots, q_m' \in Q_1 \cap Q_2$ such that for all i, j, $t_i \xrightarrow[\mathcal{A}_1^*]{*} q_i$, $v_i \xrightarrow[\mathcal{A}_2^*]{*} q_i$, $v_j' \xrightarrow[\mathcal{A}_1^*]{*} q_j'$, $u_j \xrightarrow[\mathcal{A}_2^*]{*} q_j'$. Let C'' be the largest context more general than C and C'; the positions of C'' are the positions of both $C[q_1, \dots, q_n]_{p_1,\dots,p_n}$ and $C'[q_1', \dots, q_m']_{p_1',\dots,p_m'}$. C'', $p_1'', \dots, p_l''$ are such that:

- For each $1 \leq i \leq l$, there is a j such that either $p_j = p_i''$ or $p_j' = p_i''$
- For all $1 \leq i \leq n$ there is a j such that $p_i \geq p_j''$
- For all $1 \leq i \leq m$ there is a j such that $p_i' \geq p_j''$
- the positions p_j'' are pairwise incomparable w.r.t. the prefix ordering.

Let us fix a $j \in [1..l]$. Assume that $p_j'' = p_i$ (the other case is symmetric). We can apply our lemma to $t_j = t|_{p_j''}$ (in place of t), $v_j = v|_{p_j''}$ (in place of v) and $u|_{p_j''}$ (in place of u), showing that $u|_{p_j''} \xrightarrow[\mathcal{A}_2^*]{*} q_i$. If we let now $q_j'' = q_i$ when $p_j'' = p_i$ and $q_j'' = q_i'$ when $p_j'' = p_i'$, we get

$$t \xrightarrow[\mathcal{A}_1^*]{*} C''[q_1'', \dots, q_l'']_{p_1'',\dots,p_l''} \xleftarrow[\mathcal{A}_2^*]{*} u$$

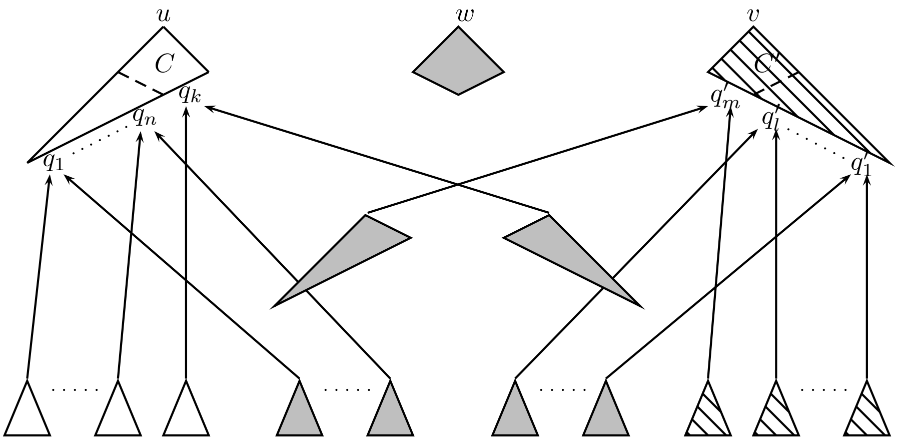


Figure 3.6: The proof of Proposition 3.2.16

which completes the proof.

□

Proposition 3.2.16. *If R and R' are in GTT then their composition $R \circ R'$ is also in GTT.*

Proof. Let $(\mathcal{A}_1, \mathcal{A}_2)$ and $(\mathcal{A}'_1, \mathcal{A}'_2)$ be the two pairs of automata which recognize R and R' respectively. We assume without loss of generality that the sets of states are disjoint:

$$(Q_1 \cup Q_2) \cap (Q'_1 \cup Q'_2) = \emptyset$$

We define the automaton $\mathcal{A}^*_1$ as follows: the set of states is $Q_1 \cup Q'_1$ and the transitions are the union of:

- the transitions of $\mathcal{A}_1$
- the transitions of $\mathcal{A}'_1$
- the ϵ-transitions $q \rightarrow q'$ if $q \in Q_1 \cap Q_2$, $q' \in Q'_1$ and $L_{\mathcal{A}_2}(q) \cap L_{\mathcal{A}'_1}(q') \neq \emptyset$

Symmetrically, the automaton $\mathcal{A}^*_2$ is defined by: its states are $Q_2 \cup Q'_2$ and the transitions are:

- the transitions of $\mathcal{A}_2$
- the transitions of $\mathcal{A}'_2$
- the ϵ-transitions $q' \rightarrow q$ if $q' \in Q'_1 \cap Q'_2$, $q \in Q_2$ and $L_{\mathcal{A}'_1}(q') \cap L_{\mathcal{A}_2}(q) \neq \emptyset$

We prove below that $(\mathcal{A}^*_1, \mathcal{A}^*_2)$ is a GTT recognizing $R \circ R'$. See also the figure 3.6.

- Assume first that $(u, v) \in R \circ R'$. Then there is a term w such that $(u, w) \in R$ and $(w, v) \in R'$:

$$u = C[u_1, \ldots, u_k]_{p_1,\ldots,p_k},\ w = C[w_1, \ldots, w_k]_{p_1,\ldots,p_k}$$

$$w = C'[w'_1, \dots, w'_m]_{p'_1,\dots,p'_m},\ v = C'[v_1, \dots, v_m]_{p'_1,\dots,p'_m}$$

and, for every $i \in \{1,\dots,k\}$, $u_i \xrightarrow[\mathcal{A}_1]{*} q_i$, $w_i \xrightarrow[\mathcal{A}_2]{*} q_i$, for every $i \in \{1,\dots,m\}$, $w'_i \xrightarrow[\mathcal{A}'_1]{*} q'_i$, $v_i \xrightarrow[\mathcal{A}'_2]{*} q'_i$. Let $p''_1,\dots,p''_l$ be the minimal elements (w.r.t. the prefix ordering) of the set $\{p_1,\dots,p_k\} \cup \{p'_1,\dots,p'_m\}$. Each p''_i is either some p_j or some p'_j. Assume first $p''_i = p_j$. Then p_j is a position in C' and

$$C'[q'_1,\dots,q'_m]_{p'_1,\dots,p'_m}|_{p_j} = C_j[q'_{m_j},\dots,q'_{m_j+k_j}]_{p'_{m_j},\dots,p'_{m_j+k_j}}$$

Now, $w_j \xrightarrow[\mathcal{A}_2]{*} q_j$ and

$$w_j = C_j[w'_{m_j},\dots,w'_{m_j+k_j}]_{p'_{m_j},\dots,p'_{m_j+k_j}}$$

with $w'_{m_j+i} \xrightarrow[\mathcal{A}'_1]{*} q'_{m_j+i}$ for every $i \in \{1,\dots,k_j\}$. For $i \in \{1,\dots,k_j\}$, let $q_{j,i}$ be such that:

$$\begin{cases} w'_{m_j+i} = w_j|_{p'_{m_j+i}} \xrightarrow[\mathcal{A}_2]{*} q_{j,i} \\ C_j[q_{j,1},\dots,q_{j,k_j}]_{p'_{m_j},\dots,p'_{m_j+k_j}} \xrightarrow[\mathcal{A}_2]{*} q_j \end{cases}$$

For every i, $w'_{m_j+i} \in L_{\mathcal{A}_2}(q_{j,i}) \cap L_{\mathcal{A}'_1}(q'_{m_j+i})$ and $q'_{m_j+i} \in Q'_1 \cap Q'_2$. Then, by definition, there is a transition $q'_{m_j+i} \xrightarrow[\mathcal{A}_2^*]{} q_{j,i}$. Therefore, $C_j[q'_{m_j},\dots,q'_{m_j+k_j}] \xrightarrow[\mathcal{A}_2^*]{*} q_j$ and then $v|_{p_j} \xrightarrow[\mathcal{A}_2^*]{*} q_j$.

Now, if $p''_i = p'_j$, we get, in a similar way, $u|_{p'_j} \xrightarrow[\mathcal{A}_1^*]{*} q'_j$. Altogether:

$$u \xrightarrow[\mathcal{A}_1^*]{*} C''[q''_1,\dots,q''_l]_{p''_1,\dots,p''_l} \xleftarrow[\mathcal{A}_2^*]{*} v$$

where $q''_i = q_j$ if $p''_i = p_j$ and $q''_j = q'_j$ if $p''_i = p'_j$.

- Conversely, assume that (u,v) is accepted by $(\mathcal{A}_1^*, \mathcal{A}_2^*)$. Then

$$u \xrightarrow[\mathcal{A}_1^*]{*} C[q''_1,\dots,q''_l]_{p''_1,\dots,p''_l} \xleftarrow[\mathcal{A}_2^*]{*} v$$

and, for every i, either $q''_i \in Q_1 \cap Q_2$ or $q''_i \in Q'_1 \cap Q'_2$ (by the disjointness hypothesis). Assume for instance that $q''_i \in Q'_1 \cap Q'_2$ and consider the computation of $\mathcal{A}_1^*$: $u|_{p''_i} \xrightarrow[\mathcal{A}_1^*]{*} q''_i$. By definition, $u|_{p''_i} = C_i[u_1,\dots,u_{k_i}]$ with

$$u_j \xrightarrow[\mathcal{A}_1]{*} q_j \xrightarrow[\mathcal{A}_1^*]{} q'_j$$

for every $j = 1,\dots,k_i$ and $C_i[q'_1,\dots,q'_{k_i}] \xrightarrow[\mathcal{A}'_1]{*} q''_i$. By construction, $q_j \in Q_1 \cap Q_2$ and $L_{\mathcal{A}_2}(q_j) \cap L_{\mathcal{A}'_1}(q'_j) \neq \emptyset$. Let $w_{i,j}$ be a term in this intersection and $w_i = C_i[w_{i,1}, \ldots, w_{i,k_i}]$. Then

$$\begin{cases} w_i \xrightarrow[\mathcal{A}_2]{*} C_i[q_1, \ldots, q_{k_i}] \\ u|_{p''_i} \xrightarrow[\mathcal{A}_1]{*} C_i[q_1, \ldots, q_{k_i}] \\ w_i \xrightarrow[\mathcal{A}'_1]{*} q''_i \\ v|_{p''_i} \xrightarrow[\mathcal{A}'_2]{*} q''_i \end{cases}$$

The last property comes from the fact that $v|_{p''_i} \xrightarrow[\mathcal{A}_2^*]{*} q''_i$ and, since $q''_i \in Q'_2$, there can be only transition steps from $\mathcal{A}'_2$ in this reduction.

Symmetrically, if $q''_i \in Q_1 \cap Q_2$, then we define w_i and the contexts C_i such that

$$\begin{cases} w_i \xrightarrow[\mathcal{A}_2]{*} q''_i \\ u|_{p''_i} \xrightarrow[\mathcal{A}_1]{*} q''_i \\ w_i \xrightarrow[\mathcal{A}'_1]{*} C_i[q'_1, \ldots, q'_{k_i}] \\ v|_{p''_i} \xrightarrow[\mathcal{A}'_2]{*} C_i[q'_1, \ldots, q'_{k_i}] \end{cases}$$

Finally, letting $w = C[w_1, \ldots, w_l]$, we have $(u, w) \in R$ and $(w, v) \in R'$.

□

GTTs do not have many other good closure properties (see the exercises).

3.3 The Logic WSkS

3.3.1 Syntax

Terms of WSkS are formed out of the constant ϵ, first-order variable symbols (typically written with lower-case letters $x, y, z, x', x_1, \ldots$) and unary symbols $1, \ldots, n$ written in postfix notation. For instance $x1123, \epsilon2111$ are terms. The latter will be often written omitting ϵ (e.g. 2111 instead of $\epsilon2111$).

Atomic formulas are either equalities $s = t$ between terms, inequalities $s \leq t$ or $s \geq t$ between terms, or membership constraints $t \in X$ where t is a term and X is a *second-order variable symbol.* Second-order variables will be typically denoted using upper-case letters.

Formulas are built from the atomic formulas using the logical connectives $\wedge, \vee, \neg, \Rightarrow, \Leftarrow, \Leftrightarrow$ and the quantifiers $\exists x, \forall x$(quantification on individuals)$\exists X, \forall X$ (quantification on sets); we may quantify both first-order and second-order variables.

As usual, we do not need all this artillery: we may stick to a subset of logical connectives (and even a subset of atomic formulas as will be discussed in Section 3.3.4). For instance $\phi \Leftrightarrow \psi$ is an abbreviation for $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$, $\phi \Rightarrow \psi$ is another way of writing $\psi \Leftarrow \phi$, $\phi \Rightarrow \psi$ is an abbreviation for $(\neg\phi) \vee \psi$, $\forall x.\phi$ stands for $\neg\exists x.\neg\phi$ etc. We will use the extended syntax for convenience, but we will restrict ourselves to the atomic formulas $s = t, s \leq t, t \in X$ and the logical connectives $\vee, \neg, \exists x, \exists X$ in the proofs.

The set of *free variables* of a formula ϕ is defined as usual.

3.3.2 Semantics

We consider the particular interpretation where terms are strings belonging to $\{1,\ldots,k\}^*$, $=$ is the equality of strings, and $\leq$ is interpreted as the prefix ordering. Second order variables are interpreted as *finite* subsets of $\{1,\ldots,k\}^*$, so $\in$ is then the membership predicate.

Let $t_1,\ldots,t_n \in \{1,\ldots,k\}^*$ and $S_1,\ldots,S_n$ be finite subsets of $\{1,\ldots,k\}^*$. Given a formula

$$\phi(x_1,\ldots,x_n,X_1,\ldots,X_m)$$

with free variables $x_1,\ldots,x_n,X_1,\ldots,X_m$, the assignment $\sigma = \{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n, X_1 \mapsto S_1,\ldots,X_m \mapsto S_m\}$ *satisfies* ϕ, which is written $\sigma \models \phi$ (or also $t_1,\ldots,t_n,S_1,\ldots,S_m \models \phi$) if replacing the variables with their corresponding value, the formula holds in the above model.

Remark: the logic SkS is defined as above, except that set variables may be interpreted as infinite sets.

3.3.3 Examples

We list below a number of formulas defining predicates on sets and singletons. After these examples, we may use the below-defined abbreviations as if they were primitives of the logic.

X is a subset of Y:

$$X \subseteq Y \stackrel{\text{def}}{=} \forall x.(x \in X \Rightarrow x \in Y)$$

Finite union:

$$X = \bigcup_{i=1}^{n} X_i \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} X_i \subseteq X \wedge \forall x.(x \in X \Rightarrow \bigvee_{i=1}^{n} x \in X_i)$$

Intersection:

$$X \cap Y = Z \stackrel{\text{def}}{=} \forall x.x \in Z \Leftrightarrow (x \in X \wedge x \in Y)$$

Partition:

$$\text{Partition}(X, X_1,\ldots,X_n) \stackrel{\text{def}}{=} X = \bigcup_{i=1}^{n} X_i \wedge \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} X_i \cap X_j = \emptyset$$

The prefix ordering:

$$x \leq y \stackrel{\text{def}}{=} \forall X.(y \in X \wedge (\forall z.(\bigvee_{i=1}^{k} zi \in X) \Rightarrow z \in X)) \Rightarrow x \in X$$

"every set containing y and closed by predecessor contains x"

This shows that $\leq$ can be removed from the syntax of WSkS formulas without decreasing the expressive power of the logic.

X is closed under prefix:

$$\text{PrefixClosed}(X) \stackrel{\text{def}}{=} \forall z.\forall y.(z \in X \wedge y \leq z) \Rightarrow y \in X$$

Set equality:

$$Y = X \overset{\text{def}}{=} Y \subseteq X \wedge X \subseteq Y$$

Emptiness:

$$X = \emptyset \overset{\text{def}}{=} \forall Y.(Y \subseteq X \Rightarrow Y = X)$$

X **is a Singleton:**

$$\text{Sing}(X) \overset{\text{def}}{=} X \neq \emptyset \wedge \forall Y(Y \subseteq X \Rightarrow (Y = X \vee Y = \emptyset)$$

Coding of trees: assume that k is the maximal arity of a function symbol in $\mathcal{F}$. For $t \in T(\mathcal{F})$, $C(t)$ is the tuple of sets $(S, S_{f_1}, \ldots, S_{f_n})$, where $\mathcal{F} = \{f_1, \ldots, f_n\}$, $S = \bigcup_{i=1}^{n} S_{f_i}$ and S_{f_i} is the set of positions in t which are labeled with f_i.

For instance $C(f(g(a), f(a, b)))$ is the tuple $S = \{\varepsilon, 1, 11, 2, 21, 22\}$, $S_f = \{\varepsilon, 2\}$, $S_g = \{1\}$, $S_a = \{11, 21\}$, $S_b = \{22\}$.

$(S, S_{f_1}, \ldots, S_{f_n})$ is the coding of some $t \in T(\mathcal{F})$ defined by:

$$\begin{aligned}
\text{Term}(X, X_1, \ldots, X_n) \overset{\text{def}}{=} {} & X \neq \emptyset \\
& \wedge \text{Partition}(X, X_1, \ldots, X_n) \wedge \text{PrefixClosed}(X) \\
& \wedge \bigwedge_{i=1}^{k} \bigwedge_{a(f_j)=i} \left(\bigwedge_{l=1}^{i} \forall x.(x \in X_{f_j} \Rightarrow xl \in X\right) \\
& \wedge \bigwedge_{l=i+1}^{k} \forall y.(y \in X_{f_j} \Rightarrow yl \notin X))
\end{aligned}$$

3.3.4 Restricting the Syntax

If we consider that a first-order variable is a singleton set, it is possible to transform any formula into an equivalent one which does not contain any first-order variable.

More precisely, we consider now that formulas are built upon the atomic formulas:

$$X \subseteq Y, \text{Sing}(X), X = Yi, X = \epsilon$$

using the logical connectives and second-order quantification only. Let us call this new syntax the *restricted syntax.*

These formulas are interpreted as expected. In particular $\text{Sing}(X)$ holds true when X is a singleton set and $X = Yi$ holds true when X and Y are singleton sets $\{s\}$ and $\{t\}$ respectively and $s = ti$. Let us write $\models_2$ the satisfaction relation for this new logic.

Proposition 3.3.1. *There is a translation T from WSkS formulas to the restricted syntax such that*

$$s_1, \ldots, s_n, S_1, \ldots, S_m \models \phi(x_1, \ldots, x_n, X_1, \ldots, X_m)$$

if and only if

$$\{s_1\}, \ldots, \{s_n\}, S_1, \ldots, S_m \models_2 T(\phi)(X_{x_1}, \ldots, X_{x_n}, X_1, \ldots, X_m)$$

Conversely, there is a translation T' from the restricted syntax to WSkS such that

$$S_1, \ldots, S_m \models T'(\phi)(X_1, \ldots, X_m)$$

if and only if

$$S_1, \ldots, S_m \models_2 \phi(X_1, \ldots, X_m))$$

Proof. First, according to the previous section, we can restrict our attention to formulas built upon the only atomic formulas $t \in X$ and $s = t$. Then, each atomic formula is flattened according to the rules:

$$\begin{array}{rcl} ti \in X & \rightarrow & \exists y.y = ti \wedge y \in X \\ xi = yj & \rightarrow & \exists z.z = xi \wedge z = yj \\ ti = s & \rightarrow & \exists z.z = t \wedge zi = s \end{array}$$

The last rule assumes that t is not a variable.

Next, we associate a second-order variable X_y to each first-order variable y and transform the flat atomic formulas:

$$\begin{array}{rcl} T(y \in X) & \stackrel{\text{def}}{=} & X_y \subseteq X \\ T(y = xi) & \stackrel{\text{def}}{=} & X_y = X_x i \\ T(x = \epsilon) & \stackrel{\text{def}}{=} & X_x = \epsilon \\ T(x = y) & \stackrel{\text{def}}{=} & X_x = X_y \end{array}$$

The translation of other flat atomic formulas can be derived from these ones, in particular when exchanging the arguments of $=$.

Now, $T(\phi \vee \psi) \stackrel{\text{def}}{=} T(\phi) \vee T(\psi)$, $T(\neg(\phi)) \stackrel{\text{def}}{=} \neg T(\phi)$, $T(\exists X.\phi) \stackrel{\text{def}}{=} \exists X.T(\phi)$, $T(\exists y.\phi) \stackrel{\text{def}}{=} \exists X_y.\mathrm{Sing}(X_y) \wedge T(\phi)$. Finally, we add $\mathrm{Sing}(X_x)$ for each free variable x.

For the converse, the translation T' has been given in the previous section, except for the atomic formulas $X = Yi$ (which becomes $\mathrm{Sing}(X) \wedge \mathrm{Sing}(Y) \wedge \exists x \exists y.x \in X \wedge y \in Y \wedge x = yi$) and $X = \epsilon$ (which becomes $\mathrm{Sing}(X) \wedge \forall x.x \in X \Rightarrow x = \epsilon$).

□

3.3.5 Definable Sets are Recognizable Sets

Definition 3.3.2. *A set L of tuples of finite sets of words is* definable in WSkS *if there is a formula ϕ of WSkS with free variables $X_1, \ldots, X_n$ such that*

$$(S_1, \ldots, S_n) \in L \textit{ if and only if } S_1, \ldots, S_n \models \phi.$$

Each tuple of finite sets of words $S_1, \ldots, S_n \subseteq \{1, \ldots, k\}^*$ is identified to a finite tree $(S_1, \ldots, S_n)^{\sim}$ over the alphabet $\{0, 1, \bot\}^n$ where any string containing a 0 or a 1 is k-ary and $\bot^n$ is a constant symbol, in the following way[3]:

$$\mathcal{P}os((S_1, \ldots, S_n)^{\sim}) \stackrel{\text{def}}{=} \{\epsilon\} \cup \{pi \mid \exists p' \in \bigcup_{j=1}^{n} S_j, p \leq p', i \in \{1, \ldots, k\}\}$$

---

[3] This is very similar to the coding of Section 3.2.1

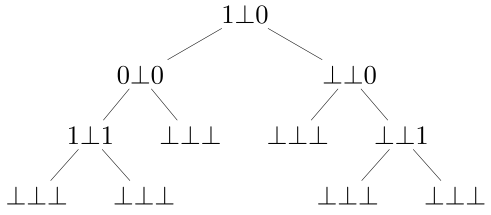


Figure 3.7: An example of a tree coding a triple of finite sets of strings

is the set of prefixes of words in some S_i. The symbol at position p:

$$(S_1, \ldots, S_n)^\sim(p) = \alpha_1 \ldots \alpha_n$$

is defined as follows:

- $\alpha_i = 1$ if and only if $p \in S_i$
- $\alpha_i = 0$ if and only if $p \notin S_i$ and $\exists p' \in S_i$ and $\exists p''.p \cdot p'' = p'$
- $\alpha_i = \perp$ otherwise.

Example 3.3.3. Consider for instance $S_1 = \{\epsilon, 11\}$, $S_2 = \emptyset$, $S_3 = \{11, 22\}$ three subsets of $\{1, 2\}^*$. Then the coding $(S_1, S_2, S_3)^\sim$ is depicted on Figure 3.7.

Lemma 3.3.4. *If a set L of tuples of finite subsets of $\{1, \ldots, k\}^*$ is definable in WSkS, then $\widetilde{L} \overset{def}{=} \{(S_1, \ldots, S_n)^\sim \mid (S_1, \ldots, S_n) \in L\}$ is in Rec.*

Proof. By Proposition 3.3.1, if L is definable in WSkS, it is also definable with the restricted syntax. We are now going to prove the lemma by induction on the structure of the formula ϕ which defines L. We assume that all variables in ϕ are bound at most once in the formula and we also assume a fixed total ordering $\leq$ on the variables. If ψ is a subformula of ϕ with free variables $Y_1 < \cdots < Y_n$, we construct an automaton $\mathcal{A}_\psi$ working on the alphabet $\{0, 1, \perp\}^n$ such that $(S_1, \ldots, S_n) \models_2 \psi$ if and only if $(S_1, \ldots, S_n)^\sim \in L(\mathcal{A}_\psi)$.

The base case consists in constructing an automaton for each atomic formula. (We assume here that $k = 2$ for simplicity, but this works of course for arbitrary k).

The automaton $\mathcal{A}_{\text{Sing}(X)}$ is depicted on Figure 3.8. The only final state is q'.

The automaton $\mathcal{A}_{X \subseteq Y}$ (with $X < Y$) is depicted on Figure 3.9. The only state (which is also final) is q.

The automaton $\mathcal{A}_{X=Y1}$ is depicted on Figure 3.10. The only final state is q''. An automaton for $X = Y2$ is obtained in a similar way.

The automaton for $X = \epsilon$ is depicted on Figure 3.11 (the final state is q').

Now, for the induction step, we have several cases to investigate:

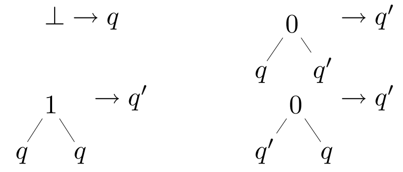

⊥ → q
0 → q'
q q'
1 → q'
q q
0 → q'
q' q

Figure 3.8: The automaton for Sing(X)

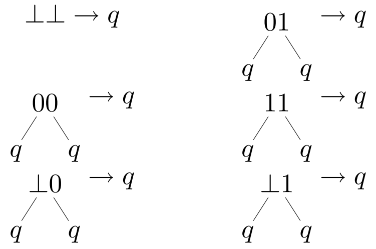

⊥⊥ → q
01 → q
q q
00 → q
q q
11 → q
q q
⊥0 → q
q q
⊥1 → q
q q

Figure 3.9: The automaton for $X \subseteq Y$

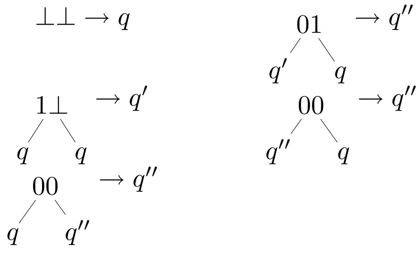

⊥⊥ → q
01 → q''
q' q
1⊥ → q'
q q
00 → q''
q'' q
00 → q''
q q''

Figure 3.10: The automaton for $X = Y1$

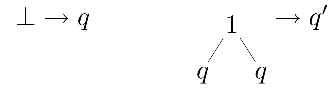


Figure 3.11: The automaton for $X = \epsilon$

- If ϕ is a disjunction $\phi_1 \vee \phi_2$, where $\vec{X_i}$ is the set of free variables of ϕ_i respectively. Then we first cylindrify the automata for ϕ_1 and ϕ_2 in such a way that they recognize the solutions of ϕ_1 and ϕ_2, with free variables $\vec{X_1} \cup \vec{X_2}$. (See Proposition 3.2.12.) More precisely, let $\vec{X_1} \cup \vec{X_2} = \{Y_1, \ldots, Y_n\}$ with $Y_1 < \ldots < Y_n$. Then we successively apply the ith cylindrification to the automaton of ϕ_1 (resp. ϕ_2) for the variables Y_i which are not free in ϕ_1 (resp. ϕ_2). Then the automaton $\mathcal{A}_\phi$ is obtained as the union of these automata. (*Rec* is closed under union by Proposition 3.2.9).
- If ϕ is a formula $\neg\phi_1$ then $\mathcal{A}_\phi$ is the automaton accepting the complement of $\mathcal{A}_{\phi_1}$. (See Theorem 1.3.1)
- If ϕ is a formula $\exists X.\phi_1$, assume that X corresponds to the ith component. Then $\mathcal{A}_\phi$ is the i^{th} projection of $\mathcal{A}_{\phi_1}$ (see Proposition 3.2.12).

□

Example 3.3.5. Consider the following formula, with free variables X, Y:

$$\forall x, y.(x \in X \wedge y \in Y) \Rightarrow \neg(x \geq y)$$

We want to compute an automaton which accepts the assignments to X, Y satisfying the formula. First, write the formula as

$$\neg\exists X_1, Y_1.X_1 \subseteq X \wedge Y_1 \subseteq Y \wedge G(X_1, Y_1)$$

where $G(X_1, Y_1)$ expresses that X_1 is a singleton x, Y_1 is a singleton y and $x \geq y$. We can use the definition of $\geq$ as a WS2S formula, or compute directly the automaton, yielding

$$\begin{array}{rcl@{\qquad}rcl}
\bot\bot & \rightarrow & q & 11(q,q) & \rightarrow & q_2 \\
1\bot(q,q) & \rightarrow & q_1 & 0\bot(q,q_1) & \rightarrow & q_1 \\
0\bot(q_1,q) & \rightarrow & q_1 & 01(q_1,q) & \rightarrow & q_2 \\
01(q,q_2) & \rightarrow & q_2 & 00(q_2,q) & \rightarrow & q_2 \\
00(q,q_2) & \rightarrow & q_2 & & &
\end{array}$$

where q_2 is the only final state. Now, using cylindrification, intersection, projection and negation we get the following automaton (intermediate steps yield large automata which would require a full page to be displayed):

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\bot\bot & \rightarrow & q_0 & \bot 1(q_0,q_0) & \rightarrow & q_1 & 1\bot(q_0,q_0) & \rightarrow & q_2 \\
\bot 0(q_0,q_1) & \rightarrow & q_1 & \bot 0(q_1,q_0) & \rightarrow & q_1 & \bot 0(q_1,q_1) & \rightarrow & q_1 \\
0\bot(q_0,q_2) & \rightarrow & q_2 & 0\bot(q_2,q_0) & \rightarrow & q_2 & 0\bot(q_2,q_2) & \rightarrow & q_2 \\
\bot 1(q_0,q_1) & \rightarrow & q_1 & \bot 1(q_1,q_0) & \rightarrow & q_1 & \bot 1(q_1,q_1) & \rightarrow & q_1 \\
1\bot(q_0,q_2) & \rightarrow & q_2 & 1\bot(q_2,q_0) & \rightarrow & q_2 & 1\bot(q_2,q_2) & \rightarrow & q_2 \\
10(q_1,q_0) & \rightarrow & q_3 & 10(q_0,q_1) & \rightarrow & q_3 & 10(q_1,q_1) & \rightarrow & q_3 \\
10(q_1,q_2) & \rightarrow & q_3 & 10(q_2,q_1) & \rightarrow & q_3 & 10(q_i,q_3) & \rightarrow & q_3 \\
10(q_3,q_i) & \rightarrow & q_3 & 00(q_i,q_3) & \rightarrow & q_3 & 00(q_3,q_i) & \rightarrow & q_3
\end{array}$$

where i ranges over $\{0, 1, 2, 3\}$ and q_3 is the only final state.

3.3.6 Recognizable Sets are Definable

We have seen in section 3.3.3 how to represent a term using a tuple of set variables. Now, we use this formula Term on the coding of tuples of terms; for $(t_1,\ldots,t_n) \in T(\mathcal{F})^n$, we write $\overline{[t_1,\ldots,t_n]}$ the $(|\mathcal{F}|+1)^n + 1$-tuple of finite sets which represents it: one set for the positions of $[t_1,\ldots,t_n]$ and one set for each element of the alphabet $(\mathcal{F} \cup \{\perp\})^n$. As it has been seen in section 3.3.3, there is a WSkS formula $\mathrm{Term}([t_1,\ldots,t_n])$ which expresses the image of the coding.

Lemma 3.3.6. *Every relation in Rec is definable. More precisely, if $R \in$ Rec there is a formula ϕ such that, for all terms $t_1,\ldots,t_n$, if $(S_1,\ldots,S_m) = \overline{[t_1,\ldots,t_n]}$, then*

$$(S_1,\ldots,S_m) \models_2 \phi \text{ if and only if } (t_1,\ldots,t_n) \in R$$

Proof. Let $\mathcal{A}$ be the automaton which accepts the set of terms $[t_1,\ldots,t_n]$ for $(t_1,\ldots,t_n) \in R$. The terminal alphabet of $\mathcal{A}$ is $\mathcal{F}' = (\mathcal{F} \cup \{\perp\})^n$, the set of states Q, the final states Q_f and the set of transition rules T. Let $\mathcal{F}' = \{f_1,\ldots,f_m\}$ and $Q = \{q_1,\ldots,q_l\}$. The following formula $\phi_{\mathcal{A}}$ (with $m+1$ free variables) defines the set $\{\overline{[t_1,\ldots,t_n]} \mid (t_1,\ldots,t_n) \in R\}$.

$$\begin{array}{ll}
\multicolumn{2}{l}{\exists Y_{q_1},\ldots,\exists Y_{q_l}.} \\
 & \mathrm{Term}(X, X_{f_1},\ldots,X_{f_m}) \\
\wedge & \mathrm{Partition}(X, Y_{q_1},\ldots,Y_{q_l}) \\
\wedge & \bigvee_{q \in Q_f} \epsilon \in Y_q \\
\wedge & \forall x. \displaystyle\bigwedge_{f \in \mathcal{F}'} \bigwedge_{q \in Q} \Big((x \in X_f \wedge x \in Y_q) \Rightarrow (\bigvee_{f(q_1,\ldots,q_s) \to q \in T} \bigwedge_{i=1}^{s} xi \in Y_{q_i}) \Big)
\end{array}$$

This formula basically expresses that there is a successful run of the automaton on $[t_1,\ldots,t_n]$: the variables Y_{q_i} correspond to sets of positions which are labeled with q_i by the run. They should be a partition of the set of positions. The root has to be labeled with a final state (the run is successful). Finally, the last line expresses local properties that have to be satisfied by the run: if the sons xi of a position x are labeled with $q_1,\ldots,q_n$ respectively and x is labeled with symbol f and state q, then there should be a transition $f(q_1,\ldots,q_n) \to q$ in the set of transitions.

We have to prove two inclusions. First assume that $(S, S_1,\ldots,S_m) \models_2 \phi$. Then $(S, S_1,\ldots,S_m) \models \mathrm{Term}(X, X_{f_1},\ldots,X_{f_m})$, hence there is a term $u \in T(\mathcal{F}')$ whose set of position is S and such that for all i, S_i is the set of positions labeled with f_i. Now, there is a partition $E_{q_1},\ldots,E_{q_l}$ of S which satisfies

$$\begin{array}{l}
S, S_1,\ldots,S_m, E_{q_1},\ldots,E_{q_l} \models \\
\qquad \forall x. \displaystyle\bigwedge_{f \in \mathcal{F}'} \bigwedge_{q \in Q} \Big((x \in X_f \wedge x \in Y_q) \Rightarrow (\bigvee_{f(q_1,\ldots,q_s) \to q \in T} \bigwedge_{i=1}^{s} xi \in Y_{q_i}) \Big)
\end{array}$$

This implies that the labeling $E_{q_1},\ldots,E_{q_l}$ is compatible with the transition rules: it defines a run of the automaton. Finally, the condition that the root ε belongs to E_{q_f} for some final state q_f implies that the run is successful, hence that u is accepted by the automaton.

Conversely, if u is accepted by the automaton, then there is a successful run of $\mathcal{A}$ on u and we can label its positions with states in such a way that this labeling is compatible with the transition rules in $\mathcal{A}$. □

Putting together Lemmas 3.3.4 and 3.3.6, we can state the following slogan (which is not very precise; the precise statements are given by the lemmas):

Theorem 3.3.7. *L is definable if and only if L is in Rec.*

And, as a consequence:

Theorem 3.3.8 ([TW68])**.** *WSkS is decidable.*

Proof. Given a formula ϕ of WSkS, by Lemma 3.3.4, we can compute a finite tree automaton which has the same solutions as ϕ. Now, assume that ϕ has no free variable. Then the alphabet of the automaton is empty (or, more precisely, it contains the only constant $\top$ according to what we explained in Section 3.2.4). Finally, the formula is valid iff the constant $\top$ is in the language, i.e. iff there is a rule $\top \rightarrow q_f$ for some $q_f \in Q_f$. □

3.3.7 Complexity Issues

We have seen in chapter 1 that, for finite tree automata, emptiness can be decided in linear time (and is PTIME-complete) and that inclusion is EXPTIME-complete. Considering WSkS formulas with a fixed number of quantifier alternations N, the decision method sketched in the previous section will work in time which is a tower of exponentials, the height of the tower being $O(N)$. This is so because each time we encounter a sequence $\forall X, \exists Y$, the existential quantification corresponds to a projection, which may yield a non-deterministic automaton, even if the input automaton was deterministic. Then the elimination of $\forall X$ requires a determinization (because we have to compute a complement automaton) which requires in general exponential time and exponential space.

Actually, it is not really possible to do much better since, even when $k = 1$, deciding a formula of WSkS requires non-elementary time, as shown in [SM73].

3.3.8 Extensions

There are several extensions of the logic, which we already mentioned: though quantification is restricted to finite sets, we may consider infinite sets as models (this is what is often called *weak second-order monadic logic with k successors* and also written WSkS), or consider quantifications on arbitrary sets (this is the full SkS).

These logics require more sophisticated automata which recognize sets of *infinite* terms. The proof of Theorem 3.3.8 carries over these extensions, with the provision that the devices enjoy the required closure and decidability properties. But this becomes much more intricate in the case of infinite terms. Indeed, for infinite terms, it is not possible to design bottom-up tree automata. We have to use a top-down device. But then, as mentioned in chapter 1, we cannot expect to reduce the non-determinism. Now, the closure by complement becomes problematic because the usual way of computing the complement uses reduction of non-determinism as a first step.

It is out of the scope of this book to define and study automata on infinite objects (see [Tho90] instead). Let us simply mention that the closure under complement for *Rabin automata* which work on infinite trees (this result is known as *Rabin's Theorem*) is one of the most difficult results in the field.

3.4 Examples of Applications

3.4.1 Terms and Sorts

The most basic example is what is known in the algebraic specification community as *order-sorted signatures.* These signatures are exactly what we called bottom-up tree automata. There are only differences in the syntax. For instance, the following signature:

SORTS:$\mathsf{Nat}, \mathsf{int}$
SUBSORTS : $\mathsf{Nat} \leq \mathsf{int}$
FUNCTION DECLARATIONS:

$$\begin{array}{lrcl}
0: & & \rightarrow & \mathsf{Nat} \\
+: & \mathsf{Nat} \times \mathsf{Nat} & \rightarrow & \mathsf{Nat} \\
s: & \mathsf{Nat} & \rightarrow & \mathsf{Nat} \\
p: & \mathsf{Nat} & \rightarrow & \mathsf{int} \\
+: & \mathsf{int} \times \mathsf{int} & \rightarrow & \mathsf{int} \\
abs: & \mathsf{int} & \rightarrow & \mathsf{Nat} \\
fact: & \mathsf{Nat} & \rightarrow & \mathsf{Nat} \\
\ldots & & &
\end{array}$$

is an automaton whose states are $\mathsf{Nat}, \mathsf{int}$ with an ϵ-transition from Nat to int and each function declaration corresponds to a transition of the automaton. For example $+(\mathsf{Nat}, \mathsf{Nat}) \rightarrow \mathsf{Nat}$. The set of *well-formed terms* (as in the algebraic specification terminology) is the set of terms recognized by the automaton in any state.

More general typing systems also correspond to more general automata (as will be seen e.g. in the next chapter).

This correspondence is not surprising; types and sorts are introduced in order to prevent run-time errors by some "abstract interpretation" of the inputs. Tree automata and tree grammars also provide such a symbolic evaluation mechanism. For other applications of tree automata in this direction, see e.g. chapter 5.

From what we have seen in this chapter, we can go beyond simply recognizing the set of well-formed terms. Consider the following *sort constraints* (the alphabet $\mathcal{F}$ of function symbols is given):

The set of *sort expressions* $\mathcal{SE}$ is the least set such that

- $\mathcal{SE}$ contains a finite set of *sort symbols* S, including the two particular symbols $\top_S$ and $\bot_S$.
- If $s_1, s_2 \in \mathcal{SE}$, then $s_1 \vee s_2$, $s_1 \wedge s_2$, $\neg s_1$ are in $\mathcal{SE}$
- If $s_1, \ldots, s_n$ are in $\mathcal{SE}$ and f is a function symbol of arity n, then $f(s_1, \ldots, s_n) \in \mathcal{SE}$.

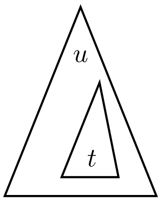


Figure 3.12: u encompasses t

The atomic formulas are expressions $t \in s$ where $t \in T(\mathcal{F}, \mathcal{X})$ and $s \in \mathcal{SE}$. The formulas are arbitrary first-order formulas built on these atomic formulas.

These formulas are interpreted as follows: we are giving an order-sorted signature (or a tree automaton) whose set of sorts is S. We define the interpretation $[\![\cdot]\!]_S$ of sort expressions as follows:

- if $s \in S$, $[\![s]\!]_S$ is the set of terms in $T(\mathcal{F})$ that are accepted in state s.
- $[\![\top_S]\!]_S = T(\mathcal{F})$ and $[\![\bot_S]\!]_S = \emptyset$
- $[\![s_1 \vee s_2]\!]_S = [\![s_1]\!]_S \cup [\![s_2]\!]_S$, $[\![s_1 \wedge s_2]\!]_S = [\![s_1]\!]_S \cap [\![s_2]\!]_S$, $[\![\neg s]\!]_S = T(\mathcal{F}) \setminus [\![s]\!]_S$
- $[\![f(s_1, \ldots, s_n)]\!]_S = \{f(t_1, \ldots, t_n) \mid t_1 \in [\![s_1]\!]_S, \ldots t_n \in [\![s_n]\!]_S\}$

An assignment σ, mapping variables to terms in $T(\mathcal{F})$, *satisfies* $t \in s$ (we also say that σ *is a solution* of $t \in s$) if $t\sigma \in [\![s]\!]_S$. Solutions of arbitrary formulas are defined as expected. Then

Theorem 3.4.1. *Sort constraints are decidable.*

The decision technique is based on automata computations, following the closure properties of $Rec_\times$ and a decomposition lemma for constraints of the form $f(t_1, \ldots, t_n) \in s$.

More results and applications of sort constraints are discussed in the bibliographic notes.

3.4.2 The Encompassment Theory for Linear Terms

Definition 3.4.2. *If $t \in T(\mathcal{F}, \mathcal{X})$ and $u \in T(\mathcal{F})$, u* encompasses *t if there is a substitution σ such that $t\sigma$ is a subterm of u. (See Figure 3.12.) This binary relation is denoted $t \trianglelefteq u$ or, seen as a unary relation on ground terms parametrized by t: $\trianglelefteq_t(u)$.*

Encompassment plays an important role in rewriting: a term t is reducible by a term rewriting system R if and only if t encompasses at least one left hand side of a rule.

The relationship with tree automata is given by the proposition:

Proposition 3.4.3. *If t is linear, then the set of terms that encompass t is recognized by a NFTA of size $O(|t|)$.*

Proof. To each non-variable subterm v of t we associate a state q_v. In addition we have a state $q_\top$. The only final state is q_t. The transition rules are:

- $f(q_\top, \dots, q_\top) \rightarrow q_\top$ for all function symbols.
- $f(q_{t_1}, \dots, q_{t_n}) \rightarrow q_{f(t_1,\dots,t_n)}$ if $f(t_1, \dots, t_n)$ is a subterm of t and q_{t_i} is actually $q_\top$ is t_i is a variable.
- $f(q_\top \dots, q_\top, q_t, q_\top, \dots, q_\top) \rightarrow q_t$ for all function symbols f whose arity is at least 1.

The proof that this automaton indeed recognizes the set of terms that encompass t is left to the reader. ☐

Note that the automaton may be non deterministic. With a slight modification, if u is a linear term, we can construct in linear time an automaton which accepts the set of instances of u (this is also left as an exercise in chapter 1, exercise 1.9).

Corollary 3.4.4. *If $\mathcal{R}$ is a term rewriting system whose all left members are linear, then the set of reducible terms in $T(\mathcal{F})$, as well as the set NF of irreducible terms in $T(\mathcal{F})$ are recognized by a finite tree automaton.*

Proof. This is a consequence of Theorem 1.3.1. ☐

The *theory of reducibility* associated with a set of term $S \subseteq T(\mathcal{F}, \mathcal{X})$ is the set of first-order formulas built on the unary predicate symbols E_t, $t \in S$ and interpreted as the set of terms encompassing t.

Theorem 3.4.5. *The reducibility theory associated with a set of linear terms is decidable.*

Proof. By proposition 3.4.3, the set of solutions of an atomic formula is recognizable, hence definable in WSkS by Lemma 3.3.6. Hence, any first-order formula built on these atomic predicate symbols can be translated into a (second-order) formula of WSkS which has the same models (up to the coding of terms into tuples of sets). Then, by Theorem 3.3.8, the reducibility theory associated with a set of linear terms is decidable. ☐

Note however that we do not use here the full power of WSkS. Actually, the solutions of a Boolean combination of atomic formulas are in $Rec_\times$. So, we cannot apply the complexity results for WSkS here. (In fact, the complexity of the reducibility theory is unknown so far).

Let us simply show an example of an interesting property of rewrite systems which can be expressed in this theory.

Definition 3.4.6. *Given a term rewriting system R, a term t is* ground reducible *if, for every ground substitution σ, $t\sigma$ is reducible by R.*

Note that a term might be irreducible and still ground reducible. For instance consider the alphabet $\mathcal{F} = \{0, s\}$ and the rewrite system $R = \{s(s(0)) \rightarrow 0\}$. Then the term $s(s(x))$ is irreducible by R, but all its ground instances are reducible.

It turns out that ground reducibility of t is expressible in the encompassment theory by the formula:

$$\forall x.(\trianglelefteq_t(x) \Rightarrow \bigvee_{i=1}^{n} \trianglelefteq_{l_i}(x))$$

Where $l_1, \ldots, l_n$ are the left hand sides of the rewrite system. By Theorem 3.4.5, if $t, l_1, \ldots, l_n$ are linear, then ground reducibility is decidable. Actually, it has been shown that this problem is EXPTIME-complete, but is beyond the scope of this book to give the proof.

3.4.3 The First-order Theory of a Reduction Relation: the Case Where no Variables are Shared

We consider again an application of tree automata to a decision problem in logic and term rewriting.

Consider the following logical theory. Let $\mathcal{L}$ be the set of all first-order formulas using no function symbols and a single binary predicate symbol $\rightarrow$.

Given a rewrite system $\mathcal{R}$, interpreting $\rightarrow$ as $\xrightarrow[\mathcal{R}]{}$, yields the *theory of one step rewriting*; interpreting $\rightarrow$ as $\xrightarrow[\mathcal{R}]{*}$ yields the *theory of rewriting.*

Both theories are undecidable for arbitrary $\mathcal{R}$. They become however decidable if we restrict our attention to term rewriting systems in which each variable occurs at most once. Basically, the reason is given by the following:

Proposition 3.4.7. *If $\mathcal{R}$ is a linear rewrite system such that left and right members of the rules do not share variables, then $\xrightarrow[\mathcal{R}]{*}$ is recognized by a GTT.*

Proof. As in the proof of Proposition 3.4.3, we can construct in linear time a (non-deterministic) automaton which accepts the set of instances of a linear term. For each rule $l_i \rightarrow r_i$ we can construct a pair $(\mathcal{A}_i, \mathcal{A}'_i)$ of automata which respectively recognize the set of instances of l_i and the set of instances of r_i. Assume that the sets of states of the $\mathcal{A}_i$s are pairwise disjoint and that each $\mathcal{A}_i$ has a single final state q^i_f. We may assume a similar property for the $\mathcal{A}'_i$s: they do not share states and for each i, the only common state between $\mathcal{A}_i$ and $\mathcal{A}'_i$ is q^i_f (the final state for both of them). Then $\mathcal{A}$ (resp. $\mathcal{A}'$) is the union of the $\mathcal{A}_i$s: the states are the union of all sets of states of the $\mathcal{A}_i$s (resp. $\mathcal{A}'_i$s), transitions and final states are also unions of the transitions and final states of each individual automaton.

We claim that $(\mathcal{A}, \mathcal{A}')$ defines a GTT whose closure by iteration $(\mathcal{A}_*, \mathcal{A}'_*)$ (which is again a GTT according to Theorem 3.2.14) accepting $\xrightarrow[\mathcal{R}]{*}$. For, assume first that $u \xrightarrow[l_i \rightarrow r_i]{p} v$. Then $u|_p$ is an instance $l_i\sigma$ of l_i, hence is accepted in state q^i_f. $v|_p$ is an instance $r_i\theta$ of r_i, hence accepted in state q^i_f. Now, $v = u[r_i\theta]_p$, hence (u, v) is accepted by the GTT $(\mathcal{A}, \mathcal{A}')$. It follows that if $u \xrightarrow[\mathcal{R}]{*} v$, (u, v) is accepted by $(\mathcal{A}_*, \mathcal{A}'_*)$.

Conversely, assume that (u, v) is accepted by $(\mathcal{A}, \mathcal{A}')$, then

$$u \xrightarrow[\mathcal{A}]{*} C[q_1, \ldots, q_n]_{p_1,\ldots,p_n} \xleftarrow[\mathcal{A}']{*} v$$

Moreover, each q_i is some state q_f^j, which, by definition, implies that $u|_{p_i}$ is an instance of l_j and $v|_{p_i}$ is an instance of r_j. Now, *since l_j and r_j do not share variables*, for each i, $u|_{p_i} \xrightarrow[\mathcal{R}]{} v|_{p_i}$. Which implies that $u \xrightarrow[\mathcal{R}]{*} v$. Now, if (u, v) is accepted by $(\mathcal{A}_*, \mathcal{A}'_*)$, u can be rewritten in v by the transitive closure of $\xrightarrow[\mathcal{R}]{*}$, which is $\xrightarrow[\mathcal{R}]{*}$ itself.

□

Theorem 3.4.8. *If $\mathcal{R}$ is a linear term rewriting system such that left and right members of the rules do not share variables, then the first-order theory of rewriting is decidable.*

Proof. By Proposition 3.4.7, $\xrightarrow[\mathcal{R}]{*}$ is recognized by a GTT. From Proposition 3.2.7, $\xrightarrow[\mathcal{R}]{*}$ is in *Rec*. By Lemma 3.3.6, there is a WSkS formula whose solutions are exactly the pairs (s, t) such that $s \xrightarrow[\mathcal{R}]{*} t$. Finally, by Theorem 3.3.8, the first-order theory of $\xrightarrow[\mathcal{R}]{*}$ is decidable. □

3.4.4 Reduction Strategies

So far, we gave examples of first-order theories (or *constraint systems*) which can be decided using tree automata techniques. Other examples will be given in the next two chapters. We give here another example of application in a different spirit: we are going to show how to decide the existence (and compute) "optimal reduction strategies" in term rewriting systems. Informally, a reduction sequence is optimal when every redex which is contracted along this sequence has to be contracted in any reduction sequence yielding a normal form. For example, if we consider the rewrite system $\{x \vee \top \to \top; \top \vee x \to \top\}$, there is no optimal sequential reduction strategy in the above sense since, given an expression $e_1 \vee e_2$, where e_1 and e_2 are unevaluated, the strategy should specify which of e_1 or e_2 has to be evaluated first. However, if we start with e_1, then maybe e_2 will reduce to $\top$ and the evaluation step on e_1 was unnecessary. Symmetrically, evaluating e_2 first may lead to unnecessary computations. An interesting question is to give sufficient criteria for a rewrite system to admit optimal strategies and, in case there is such a strategy, give it explicitly.

The formalization of these notions was given by Huet and Lévy in [HL91] who introduce the notion of *sequentiality*. We give briefly a summary of (part of) their definitions.

$\mathcal{F}$ is a fixed alphabet of function symbols and $\mathcal{F}_\Omega = \mathcal{F} \cup \{\Omega\}$ is the alphabet $\mathcal{F}$ enriched with a new constant Ω (whose intended meaning is "unevaluated term").

We define on $T(\mathcal{F}_\Omega)$ the relation "less evaluated than" as:

> $u \sqsubseteq v$ if and only if either $u = \Omega$ or else $u = f(u_1, \ldots, u_n)$, $v = f(v_1, \ldots, v_n)$ and for all i, $u_i \sqsubseteq v_i$

Definition 3.4.9. *A predicate P on $T(\mathcal{F}_\Omega)$ is* monotonic *if $u \in P$ and $u \sqsubseteq v$ implies $v \in P$.*

For example, a monotonic predicate of interest for rewriting is the predicate $N_{\mathcal{R}}$: $t \in N_{\mathcal{R}}$ if and only if there is a term $u \in T(\mathcal{F})$ such that u is irreducible by $\mathcal{R}$ and $t \xrightarrow[\mathcal{R}]{*} u$.

Definition 3.4.10. *Let P be a monotonic predicate on $T(\mathcal{F}_\Omega)$. If $\mathcal{R}$ is a term rewriting system and $t \in T(\mathcal{F}_\Omega)$, a position p of Ω in t is an* index *for P if for all terms $u \in T(\mathcal{F}_\Omega)$ such that $t \sqsubseteq u$ and $u \in P$, then $u|_p \neq \Omega$*

In other words: it is necessary to evaluate t at position p in order to have the predicate P satisfied.

Example 3.4.11. Let $\mathcal{R} = \{f(g(x), y) \to g(f(x, y));\ f(a, x) \to a; b \to g(b)\}$. Then 1 is an index of $f(\Omega, \Omega)$ for $N_{\mathcal{R}}$: any reduction yielding a normal form without Ω will have to evaluate the term at position 1. More formally, every term $f(t_1, t_2)$ which can be reduced to a term in $T(\mathcal{F})$ in normal form satisfies $t_1 \neq \Omega$. On the contrary, 2 is not an index of $f(\Omega, \Omega)$ since $f(a, \Omega) \xrightarrow[R]{*} a$.

Definition 3.4.12. *A monotonic predicate P is* sequential *if every term t such that:*

- *$t \notin P$*
- *there is $u \in T(\mathcal{F})$, $t \sqsubseteq u$ and $u \in P$*

has an index for P.

If $N_{\mathcal{R}}$ is sequential, the reduction strategy consisting of reducing an index is optimal for non-overlapping and left linear rewrite systems.

Now, the relationship with tree automata is given by the following result:

Theorem 3.4.13. *If P is definable in WSkS, then the sequentiality of P is expressible as a WSkS formula.*

The proof of this result is quite easy: it suffices to translate directly the definitions.

For instance, if $\mathcal{R}$ is a rewrite system whose left and right members do not share variables, then $N_{\mathcal{R}}$ is recognizable (by Propositions 3.4.7 and 3.2.7), hence definable in WSkS by Lemma 3.3.6 and the sequentiality of $N_{\mathcal{R}}$ is decidable by Theorem 3.4.13.

In general, the sequentiality of $N_{\mathcal{R}}$ is undecidable. However, one can notice that, if $\mathcal{R}$ and $\mathcal{R}'$ are two rewrite systems such that $\xrightarrow[\mathcal{R}]{} \subseteq \xrightarrow[\mathcal{R}']{}$, then a position p which is an index for $\mathcal{R}'$ is also an index for $\mathcal{R}$. (And thus, $\mathcal{R}$ is sequential whenever $\mathcal{R}'$ is sequential).

For instance, we may approximate the term rewriting system, replacing all right hand sides by a new variable which does not occur in the corresponding left member. Let $\mathcal{R}?$ be this approximation and $N?$ be the predicate $N_{\mathcal{R}?}$. (This is the approximation considered by Huet and Lévy).

Another, refined, approximation consists in renaming all variables of the right hand sides of the rules in such a way that all right hand sides become linear and do not share variables with their left hand sides. Let $\mathcal{R}'$ be such an approximation of $\mathcal{R}$. The predicate $N_{\mathcal{R}'}$ is written NV.

Proposition 3.4.14. *If $\mathcal{R}$ is left linear, then the predicates $N?$ and NV are definable in WSkS and their sequentiality is decidable.*

Proof. The approximations $\mathcal{R}?$ and $\mathcal{R}'$ satisfy the hypotheses of Proposition 3.4.7 and hence $\xrightarrow[\mathcal{R}?]{*}$ and $\xrightarrow[\mathcal{R}']{*}$ are recognized by GTTs. On the other hand, the set of terms in normal form for a left linear rewrite system is recognized by a finite tree automaton (see Corollary 3.4.4). By Proposition 3.2.7 and Lemma 3.3.6, all these predicates are definable in WSkS. Then $N?$ and NV are also definable in WSkS. For instance for NV:

$$NV(t) \stackrel{\text{def}}{=} \exists u.t \xrightarrow[\mathcal{R}']{*} u \wedge u \in NF$$

Then, by Theorem 3.4.13, the sequentiality of $N?$ and NV are definable in WSkS and by Theorem 3.3.8 they are decidable. □

3.4.5 Application to Rigid E-unification

Given a finite (universally quantified) set of equations E, the classical problem of *E-unification* is, given an equation $s = t$, find substitutions σ such that $E \models s\sigma = t\sigma$. The associated decision problem is to decide whether such a substitution exists. This problem is in general unsolvable: there are decision procedures for restricted classes of axioms E.

The *simultaneous rigid E-unification problem* is slightly different: we are still giving E and a finite set of equations $s_i = t_i$ and the question is to find a substitution σ such that

$$\models (\bigwedge_{e \in E} e\sigma) \Rightarrow (\bigwedge_{i=1}^{n} s_i\sigma = t_i\sigma)$$

The associated decision problem is to decide the existence of such a substitution.

The relevance of such questions to automated deduction is very briefly described in the bibliographic notes. We want here to show how tree automata help in this decision problem.

Simultaneous rigid E-unification is undecidable in general. However, for the one variable case, we have:

Theorem 3.4.15. *The simultaneous rigid E-unification problem with one variable is EXPTIME-complete.*

The EXPTIME membership is a direct consequence of the following lemma, together with closure and decision properties for recognizable tree languages. The EXPTIME-hardness is obtained by reduction the intersection non-emptiness problem, see Theorem 1.7.5).

Lemma 3.4.16. *The solutions of a rigid E-unification problem with one variable are recognizable by a finite tree automaton.*

Proof. (sketch) Assume that we have a single equation $s = t$. Let x be the only variable occurring in $E, s = t$ and $\hat{E}$ be the set E in which x is considered as a constant. Let R be a canonical ground rewrite system (see e.g. [DJ90]) associated with $\hat{E}$ (and for which x is minimal). We define v as x if s and t have the same normal form w.r.t. R and as the normal form of $x\sigma$ w.r.t. R otherwise.

Assume $E\sigma \models s\sigma = t\sigma$. If $v \not\equiv x$, we have $\hat{E} \cup \{x = v\} \models x = x\sigma$. Hence $\hat{E} \cup \{x = v\} \models s = t$ in any case. Conversely, assume that $\hat{E} \cup \{x = v\} \models s = t$. Then $\hat{E} \cup \{x = x\sigma\} \models s = t$, hence $E\sigma \models s\sigma = t\sigma$.

Now, assume $v \not\equiv x$. Then either there is a subterm u of an equation in $\hat{E}$ such that $\hat{E} \models u = v$ or else $R_1 = R \cup \{v \to x\}$ is canonical. In this case, from $\hat{E} \cup \{v = x\} \models s = t$, we deduce that either $\hat{E} \models s = t$ (and $v \equiv x$) or there is a subterm u of s, t such that $\hat{E} \models v = u$. we can conclude that, in all cases, there is a subterm u of $E \cup \{s = t\}$ such that $\hat{E} \models u = v$.

To summarize, σ is such that $E\sigma \models s\sigma = t\sigma$ iff there is a subterm u of $E \cup \{s = t\}$ such that $\hat{E} \models u = x\sigma$ and $\hat{E} \cup \{u = x\} \models s = t$.

If we let T be the set of subterms u of $E \cup \{s = t\}$ such that $\hat{E} \cup \{u = x\} \models s = t$, then T is finite (and computable in polynomial time). The set of solutions is then $\xrightarrow[R^{-1}]{*}(T)$, which is a recognizable set of trees, thanks to Proposition 3.4.7.

□

Further comments and references are given in the bibliographic notes.

3.4.6 Application to Higher-order Matching

We give here a last application (but the list is not closed!), in the typed lambda-calculus.

To be self-contained, let us first recall some basic definitions in typed lambda calculus.

The set of *types* of the simply typed lambda calculus is the least set containing the constant o (basic type) and such that $\tau \to \tau'$ is a type whenever τ and τ' are types.

Using the so-called Curryfication, any type $\tau \to (\tau' \to \tau'')$ is written $\tau, \tau' \to \tau''$. In this way all non-basic types are of the form $\tau_1, \ldots, \tau_n \to o$ with intuitive meaning that this is the type of functions taking n arguments of respective types $\tau_1, \ldots, \tau_n$ and whose result is a basic type o.

The ***order*** of a type τ is defined by:

- $O(o) = 1$
- $O(\tau_1, \ldots, \tau_n \to o) = 1 + \max\{O(\tau_1), \ldots, O(\tau_n)\}$

Given, for each type τ a set of variables $\mathcal{X}_\tau$ of type τ and a set C_τ of constants of type τ, the set of ***terms*** (of the simply typed lambda calculus) is the least set Λ such that:

- $x \in \mathcal{X}_\tau$ is a term of type τ
- $c \in C_\tau$ is a term of type τ
- If $x_1 \in \mathcal{X}_{\tau_1}, \ldots, x_n \in \mathcal{X}_{\tau_n}$ and t is a term of type τ, then $\lambda x_1, \ldots x_n : t$ is a term of type $\tau_1, \ldots, \tau_n \to \tau$
- If t is a term of type $\tau_1, \ldots, \tau_n \to \tau$ and $t_1, \ldots, t_n$ are terms of respective types $\tau_1, \ldots, \tau_n$, then $t(t_1, \ldots, t_n)$ is a term of type τ.

The ***order*** of a term t is the order of its type $\tau(t)$.

The set of ***free variables*** $\mathcal{V}ar(t)$ of a term t is defined by:

- $\mathcal{V}ar(x) = \{x\}$ if x is a variable
- $\mathcal{V}ar(c) = \emptyset$ if c is a constant
- $\mathcal{V}ar(\lambda x_1, \ldots, x_n : t) = \mathcal{V}ar(t) \setminus \{x_1, \ldots, x_n\}$
- $\mathcal{V}ar(t(u_1, \ldots, u_n)) = \mathcal{V}ar(t) \cup \mathcal{V}ar(u_1) \cup \ldots \cup \mathcal{V}ar(u_n)$

Terms are always assumed to be in ***η-long form***, i.e. they are assumed to be in normal form with respect to the rule:

$$(\eta) \quad t \rightarrow \lambda x_1, \ldots, x_n.t(x_1, \ldots, x_n) \quad \begin{array}{l} \text{if } \tau(t) = \tau_1, \ldots \tau_n \rightarrow \tau \\ \text{and } x_i \in \mathcal{X}_{\tau_i} \setminus \mathcal{V}ar(t) \text{ for all } i \end{array}$$

We define the ***α-equivalence*** $=_\alpha$ on Λ as the least congruence relation such that: $\lambda x_1, \ldots, x_n : t =_\alpha \lambda x'_1 \ldots, x'_n : t'$ when

- t' is the term obtained from t by substituting for every index i, every free occurrence of x_i with x'_i.
- There is no subterm of t in which, for some index i, both x_i and x'_i occur free.

In the following, we consider only lambda terms modulo α-equivalence. Then we may assume that, in any term, any variable is bounded at most once and free variables do not have bounded occurrences.

The ***β-reduction*** is defined on Λ as the least binary relation $\underset{\beta}{\rightarrow}$ such that

- $\lambda x_1, \ldots, x_n : t(t_1, \ldots, t_n) \underset{\beta}{\rightarrow} t\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$
- for every context C, $C[t] \underset{\beta}{\rightarrow} C[u]$ whenever $t \underset{\beta}{\rightarrow} u$

It is well-known that $\beta\eta$-reduction is terminating and confluent on Λ and, for every term $t \in \Lambda$, we let $t\downarrow$ be the unique normal form of t.

A ***matching problem*** is an equation $s = t$ where $s, t \in \Lambda$ and $\mathcal{V}ar(t) = \emptyset$. A ***solution*** of a matching problem is a substitution σ of the free variables of t such that $s\sigma\downarrow = t\downarrow$.

Whether or not the matching problem is decidable is an open question at the time we write this book. However, it can be decided when every free variable occurring in s is of order less or equal to 4. We sketch here how tree automata may help in this matter. We will consider only two special cases here, leaving the general case as well as details of the proofs as exercises (see also bibliographic notes).

First consider a problem

$$(1) \qquad\qquad x(s_1, \ldots, s_n) = t$$

where x is a third order variable and $s_1, \ldots, s_n, t$ are terms without free variables.

The first result states that the set of solutions is recognizable by a $\Box$-automaton. $\Box$-automata are a slight extension of finite tree automata: we assume here that the alphabet contains a special symbol $\Box$. Then a term u is accepted by a $\Box$-automaton $\mathcal{A}$ if and only if there is a term v which is accepted (in the usual sense) by $\mathcal{A}$ and such that u is obtained from v by replacing each occurrence of the symbol $\Box$ with a term (of appropriate type). Note that two distinct occurrences of $\Box$ need not to be replaced with the same term.

We consider the automaton $\mathcal{A}_{s_1,\ldots,s_n,t}$ defined by: $\mathcal{F}$ consists of all symbols occurring in t plus the variable symbols $x_1,\ldots,x_n$ whose types are respectively the types of $s_1,\ldots,s_n$ and the constant $\Box$.

The set of states Q consists of all subterms of t, which we write q_u (instead of u) and a state $q_\Box$. In addition, we have the final state q_f.

The transition rules Δ consist in

- The rules
$$f(q_{t_1},\ldots,q_{t_n}) \to q_{f(t_1,\ldots,t_n)}$$
each time $q_{f(t_1,\ldots,t_n)} \in Q$
- For $i = 1,\ldots,n$, the rules
$$x_i(q_{t_1},\ldots,q_{t_n}) \to q_u$$
where u is a subterm of t such that $s_i(t_1,\ldots,t_n)\downarrow = u$ and $t_j = \Box$ whenever $s_i(t_1,\ldots,t_{j-1},\Box,t_{j+1},\ldots,t_n)\downarrow = u$.
- the rule $\lambda x_1,\ldots,\lambda x_n.q_t \to q_f$

Theorem 3.4.17. *The set of solutions of (1) (up to α-conversion) is accepted by the $\Box$-automaton $\mathcal{A}_{s_1,\ldots,s_n,t}$.*

More about this result, its proof and its extension to fourth order will be given in the exercises (see also bibliographic notes). Let us simply give an example.

Example 3.4.18. Let us consider the interpolation equation

$$x(\lambda y_1\lambda y_2.y_1, \lambda y_3.f(y_3,y_3)) = f(a,a)$$

where y_1, y_2 are assumed to be of base type o. Then $\mathcal{F} = \{a, f, x_1, x_2, \Box_o\}$. $Q = \{q_a, q_{f(a,a)}, q_{\Box_o}, q_f\}$ and the rules of the automaton are:

$$\begin{array}{rcl@{\qquad}rcl}
a & \to & q_a & f(q_a,q_a) & \to & q_{f(a,a)} \\
\Box_o & \to & q_{\Box_o} & x_1(q_a,q_{\Box_o}) & \to & q_a \\
x_1(q_{f(a,a)},q_{\Box_o}) & \to & q_{f(a,a)} & x_2(q_a) & \to & q_{f(a,a)} \\
\lambda x_1\lambda x_2.q_{f(a,a)} & \to & q_f & & &
\end{array}$$

For instance the term $\lambda x_1\lambda x_2.x_1(x_2(x_1(x_1(a,\Box_o),\Box_o)),\Box_o)$ is accepted by the automaton:

$$\begin{array}{rcl}
\lambda x_1\lambda x_2.x_1(x_2(x_1(x_1(a,\Box_o),\Box_o)),\Box_o) & \xrightarrow[\mathcal{A}]{*} & \lambda x_1\lambda x_2.x_1(x_2(x_1(x_1(q_a,q_{\Box_o}),q_{\Box_o})),q_{\Box_o}) \\
& \xrightarrow[\mathcal{A}]{} & \lambda x_1\lambda x_2.x_1(x_2(x_1(q_a,q_{\Box_o})),q_{\Box_o}) \\
& \xrightarrow[\mathcal{A}]{} & \lambda x_1\lambda x_2.x_1(x_2(q_a),q_{\Box_o}) \\
& \xrightarrow[\mathcal{A}]{} & \lambda x_1\lambda x_2.x_1(q_{f(a,a)},q_{\Box_o}) \\
& \xrightarrow[\mathcal{A}]{} & \lambda x_1\lambda x_2.q_{f(a,a)} \\
& \xrightarrow[\mathcal{A}]{} & q_f
\end{array}$$

And indeed, for every terms t_1, t_2, t_3, $\lambda x_1\lambda x_2.x_1(x_2(x_1(x_1(a,t_1),t_2)),t_3)$ is a solution of the interpolation problem.

3.5 Exercises

Exercise 3.1. Let $\mathcal{F}$ be the alphabet consisting of finitely many unary function symbols $a_1, \ldots, a_n$ and a constant 0.

1. Show that the set S of pairs (of words) $\{(a_1^n(a_1(a_2(a_2^p(0)))), a_1^n(a_2^p(0))) \mid n, p \in \mathbb{N}\}$ is in *Rec*. Show that S^* is not in *Rec*, hence that *Rec* is not closed under transitive closure.
2. More generally, show that, for any finite rewrite system $\mathcal{R}$ (on the alphabet $\mathcal{F}$!), the reduction relation $\xrightarrow[\mathcal{R}]{}$ is in *Rec*.
3. Is there any hope to design a class of tree languages which contains *Rec*, which is closed by all Boolean operations and by transitive closure and for which emptiness is decidable? Why?

Exercise 3.2. Show that the set of pairs $\{(t, f(t,t')) \mid t, t' \in T(\mathcal{F})\}$ is not in *Rec*.

Exercise 3.3. Show that if a binary relation is recognized by a GTT, then its inverse is also recognized by a GTT.

Exercise 3.4. Give an example of two relations that are recognized by GTTs and whose union is not recognized by any GTT.

Similarly, show that the class of relations recognized by a GTT is not closed by complement. Is the class closed by intersection?

Exercise 3.5. Give an example of a n-ary relation such that its ith projection followed by its ith cylindrification does not give back the original relation. On the contrary, show that ith cylindrification followed by ith projection gives back the original relation.

Exercise 3.6. About *Rec* and bounded delay relations. We assume that $\mathcal{F}$ only contains unary function symbols and a constant, i.e. we consider words rather than trees and we write $u = a_1 \ldots a_n$ instead of $u = a_1(\ldots(a_n(0))\ldots)$. Similarly, $u \cdot v$ corresponds to the usual concatenation of words.

A binary relation R on $T(\mathcal{F})$ is called a *bounded delay relation* if and only if

$$\exists k / \forall (u,v) \in R, \ |\ |u| - |v|\ | \leq k$$

R preserves the length if and only if

$$\forall (u,v) \in R, \ \ |u| = |v|$$

If A, B are two binary relations, we write $A \cdot B$ (or simply AB) the relation

$$A \cdot B \stackrel{\text{def}}{=} \{(u,v) / \exists (u_1,v_1) \in A, (u_2,v_2) \in B u = u_1.u_2, v = v_1.v_2\}$$

Similarly, we write (in this exercise!)

$$A^* = \{(u,v) / \exists (u_1,v_1) \in A, \ldots, (u_n,v_n) \in A, u = u_1 \ldots u_n, v = v_1 \ldots v_n\}$$

1. Given $A, B \in$ *Rec*, is $A \cdot B$ necessary in *Rec*? is A^* necessary in *Rec*? Why?
2. Show that if $A \in$ *Rec* preserves the length, then $A^* \in$ *Rec*.
3. Show that if $A, B \in$ *Rec* and A is of bounded delay, then $A \cdot B \in$ *Rec*.
4. The family of *rational relations* is the smallest set of subsets of $T(\mathcal{F})^2$ which contains the finite subsets of $T(\mathcal{F})^2$ and which is closed under union, concatenation $(\cdot)$ and $*$.

   Show that, if A is a bounded delay rational relation, then $A \in$ *Rec*. Is the converse true?

Exercise 3.7. Let R_0 be the rewrite system $\{x \times 0 \to 0; 0 \times x \to 0\}$ and $\mathcal{F} = \{0, 1, s, \times\}$

1. Construct explicitly the GTT accepting $\xrightarrow[R_0]{*}$.
2. Let $R_1 = R_0 \cup \{x \times 1 \to x\}$. Show that $\xrightarrow[R_1]{*}$ is is not recognized by a GTT.
3. Let $R_2 = R_1 \cup \{1 \times x \to x \times 1\}$. Using a construction similar to the transitive closure of GTTs, show that the set $\{t \in T(\mathcal{F}) \mid \exists u \in T(\mathcal{F}), t \xrightarrow[R_2]{*} u, u \in NF\}$ where NF is the set of terms in normal form for R_2 is recognized by a finite tree automaton.

Exercise 3.8. (*) More generally, prove that given any rewrite system $\{l_i \to r_i \mid 1 \leq i \leq n\}$ such that

1. for all i, l_i and r_i are linear
2. for all i, if $x \in \mathcal{V}ar(l_i) \cap \mathcal{V}ar(r_i)$, then x occurs at depth at most one in l_i.

the set $\{t \in T(\mathcal{F}) \mid \exists u \in NF, t \xrightarrow[R]{*} u\}$ is recognized by finite tree automaton.

What are the consequences of this result?

(See [Jac96] for details about this results and its applications. Also compare with Exercise 1.17, question 4.)

Exercise 3.9. Show that the set of pairs $\{(f(t, t'), t) \mid t, t' \in T(\mathcal{F})\}$ is not definable in WSkS. (See also Exercise 3.2)

Exercise 3.10. Show that the set of pairs of words $\{(w, w') \mid l(w) = l(w')\}$, where $l(x)$ is the length of x, is not definable in WSkS.

Exercise 3.11. Let $\mathcal{F} = \{a_1, \ldots, a_n, 0\}$ where each a_i is unary and 0 is a constant. Consider the following constraint system: terms are built on $\mathcal{F}$, the binary symbols $\cap, \cup$, the unary symbol $\neg$ and set variables. Formulas are conjunctions of inclusion constraints $t \subseteq t'$. The formulas are interpreted by assigning to variables finite subsets of $T(\mathcal{F})$, with the expected meaning for other symbols.

Show that the set of solutions of such constraints is in Rec_2. What can we conclude?

(*) What happens if we remove the condition on the a_i's to be unary?

Exercise 3.12. Complete the proof of Proposition 3.2.16.

Exercise 3.13. Show that the subterm relation is not definable in WSkS.

Given a term t Write a WSkS formula ϕ_t such that a term $u \models \phi_t$ if and only if t is a subterm of u.

Exercise 3.14. Define in SkS "X is finite". (Hint: express that every totally ordered subset of X has an upper bound and use König's lemma)

Exercise 3.15. A tuple $(t_1, \ldots, t_n) \in T(\mathcal{F})^n$ can be represented in several ways as a finite sequence of finite sets. The first one is the encoding given in Section 3.3.6, overlapping the terms and considering one set for each tuple of symbols. A second one consists in having a tuple of sets for each component: one for each function symbol.

Compare the number of free variables which result from both codings when defining an n-ary relation on terms in WSkS. Compare also the definitions of the diagonal Δ using both encodings. How is it possible to translate an encoding into the other one?

Exercise 3.16. (*) Let $\mathcal{R}$ be a finite rewrite system whose all left and right members are ground.

1. Let Termination(x) be the predicate on $T(\mathcal{F})$ which holds true on t when there is no infinite sequence of reductions starting from t. Show that adding this predicate as an atomic formula in the first-order theory of rewriting, this theory remains decidable for ground rewrite systems.

2. Generalize these results to the case where the left members of $\mathcal{R}$ are linear and the right members are ground.

Exercise 3.17. The complexity of automata recognizing the set of irreducible ground terms.

1. For each $n \in \mathbb{N}$, give a linear rewrite system $\mathcal{R}_n$ whose size is $O(n)$ and such that the minimal automaton accepting the set of irreducible ground terms has a size $O(2^n)$.
2. Assume that for any two strict subterms s, t of left hand side(s) of $\mathcal{R}$, if s and t are unifiable, then s is an instance of t or t is an instance of s. Show that there is a NFTA $\mathcal{A}$ whose size is linear in $\mathcal{R}$ and which accepts the set of irreducible ground terms.

Exercise 3.18. Prove Theorem 3.4.13.

Exercise 3.19. The Propositional Linear-time Temporal Logic. The logic **PTL** is defined as follows:

Syntax P is a finite set of *propositional variables*. Each symbol of P is a formula (an atomic formula). If ϕ and ψ are formulas, then the following are formulas:

$$\phi \wedge \psi,\ \phi \vee \psi,\ \phi \rightarrow \psi,\ \neg\phi,\ \phi\mathbf{U}\psi,\ \mathbf{N}\phi,\ \mathbf{L}\phi$$

Semantics Let P^* be the set of words over the alphabet P. A word $w \in P^*$ is identified with the sequence of letters $w(0)w(1)\ldots w(|w|-1)$. $w(i..j)$ is the word $w(i)\ldots w(j)$. The satisfaction relation is defined by:

- if $p \in P$, $w \models p$ if and only if $w(0) = p$
- The interpretation of logical connectives is the usual one
- $w \models \mathbf{N}\phi$ if and only if $|w| \geq 2$ and $w(1..|w|-1) \models \phi$
- $w \models \mathbf{L}\phi$ if and only if $|w| = 1$
- $w \models \phi\mathbf{U}\psi$ if and only if there is an index $i \in [0..|w|]$ such that for all $j \in [0..i]$, $w(j..|w|-1) \models \phi$ and $w(i..|w|-1) \models \psi$.

Let us recall that the language defined by a formula ϕ is the set of words w such that $w \models \phi$.

1. What it is the language defined by $\mathbf{N}(p_1\mathbf{U}p_2)$ (with $p_1, p_2 \in P$)?
2. Give **PTL** formulas defining respectively $P^*p_1P^*$, p_1^*, $(p_1p_2)^*$.
3. Give a first-order WS1S formula (i.e. without second-order quantification and containing only one free second-order variable) which defines the same language as $\mathbf{N}(p_1\mathbf{U}p_2)$
4. For any **PTL** formula, give a first-order WS1S formula which defines the same language.
5. Conversely, show that any language defined by a first-order WS1S formula is definable by a **PTL** formula.

Exercise 3.20. About 3rd-order interpolation problems

1. Prove Theorem 3.4.17.
2. Show that the size of the automaton $\mathcal{A}_{s_1,\ldots,s_n,t}$ is $O(n \times |t|)$
3. Deduce from Exercise 1.20 that the existence of a solution to a system of interpolation equations of the form $x(s_1, \ldots, s_n) = t$ (where x is a third order variable in each equation) is in NP.

Exercise 3.21. About general third order matching.

1. How is it possible to modify the construction of $\mathcal{A}_{s_1,\dots,s_n,t}$ so as to forbid some symbols of t to occur in the solutions?
2. Consider a third order matching problem $u = t$ where t is in normal form and does not contain any free variable. Let $x_1, \dots, x_n$ be the free variables of u and $x_i(s_1, \dots, s_m)$ be the subterm of u at position p. Show that, for every solution σ, either $u[\Box]_p\sigma \downarrow =_\alpha t$ or else that $x_i\sigma(s_1\sigma, \dots, s_m\sigma)\downarrow$ is in the set S_p defined as follows: $v \in S_p$ if and only if there is a subterm t' of t and there are positions $p_1, \dots, p_k$ of t' and variables $z_1, \dots, z_k$ which are bound above p in u such that $v = t'[z_1, \dots, z_k]_{p_1,\dots,p_k}$.
3. By guessing the results of $x_i\sigma(s_1\sigma, \dots, s_m\sigma)$ and using the previous exercise, show that general third order matching is in NP.

3.6 Bibliographic Notes

The following bibliographic notes only concern the applications of the usual finite tree automata on finite trees (as defined at this stage of the book). We are pretty sure that there are many missing references and we are pleased to receive more pointers to the litterature.

3.6.1 GTT

GTT were introduced in [DTHL87] where they were used for the decidability of confluence of ground rewrite systems.

3.6.2 Automata and Logic

The development of automata in relation with logic and verification (in the sixties) is reported in [Tra95]. This research program was explained by A. Church himself in 1962 [Chu62].

Milestones of the decidability of monadic second-order logic are the papers [Büc60] [Rab69]. Theorem 3.3.8 is proved in [TW68].

3.6.3 Surveys

There are numerous surveys on automata and logic. Let us mention some of them: M.O. Rabin [Rab77] surveys the decidable theories; W. Thomas [Tho90, Tho97] provides an excellent survey of relationships between automata and logic.

3.6.4 Applications of tree automata to constraint solving

Concerning applications of tree automata, the reader is also referred to [Dau94] which reviews a number of applications of Tree automata to rewriting and constraints.

The relation between sorts and tree automata is pointed out in [Com89]. The decidability of arbitrary first-order sort constraints (and actually the first order theory of finite trees with equality and sort constraints) is proved in [CD94].

More general sort constraints involving some second-order terms are studied in [Com98b] with applications to a sort constrained equational logic [Com98a].

Sort constraints are also applied to specifications and automated inductive proofs in [BJ97] where tree automata are used to represent some normal forms sets. They are used in logic programming and automated reasoning [FSVY91, GMW97], in order to get more efficient procedures for fragments which fall into the scope of tree automata techniques. They are also used in automated deduction in order to increase the expressivity of (counter-)model constructions [Pel97].

Concerning encompassment, M. Dauchet et al gave a more general result (dropping the linearity requirement) in [DCC95]. We will come back to this result in the next chapter.

3.6.5 Application of tree automata to semantic unification

Rigid unification was originally considered by J. Gallier et al. [GRS87] who showed that this is a key notion in extending the matings method to a logic with equality. Several authors worked on this problem and it is out of the scope of this book to give a list of references. Let us simply mention that the result of Section 3.4.5 can be found in [Vea97b]. Further results on application of tree automata to rigid unification can be found in [DGN$^+$98], [GJV98].

Tree automata are also used in solving classical semantic unification problems. See e.g. [LM93] [KFK97] [Uri92]. For instance, in [KFK97], the idea is to capture some loops in the narrowing procedure using tree automata.

3.6.6 Application of tree automata to decision problems in term rewriting

Some of the applications of tree automata to term rewriting follow from the results on encompassment theory. Early works in this area are also mentioned in the bibliographic notes of Chapter 1. The reader is also referred to the survey [GT95].

The first-order theory of the binary (many-steps) reduction relation w.r.t. a ground rewrite system has been shown decidable by. M. Dauchet and S. Tison [DT90]. Extensions of the theory, including some function symbols, or other predicate symbols like the parallel rewriting or the termination predicate (Terminate(t) holds if there is no infinite reduction sequence starting from t), or fair termination etc... remain decidable [Tis89]. See also the exercises.

Both the theory of one step and the theory of many steps rewriting are undecidable for arbitrary $\mathcal{R}$ [Tre96].

Reduction strategies for term rewriting have been first studied by Huet and Lévy in 1978 [HL91]. They show here the decidability of *strong sequentiality* for orthogonal rewrite systems. This is based on an approximation of the rewrite system which, roughly, only considers the left sides of the rules. Better approximation, yielding refined criteria were further proposed in [Oya93], [Com95], [Jac96]. The orthogonality requirement has also been replaced with the weaker condition of left linearity. The first relation between tree automata, WSkS and reduction strategies is pointed out in [Com95]. Further studies of call-by-need strategies, which are still based on tree automata, but do not use a detour through monadic second-order logic can be found in [DM97]. For all these works, a key property is the preservation of regularity by (many-steps) rewriting, which was shown for ground systems in [Bra69], for linear systems which do not share variables in [DT90], for shallow systems in [Com95], for right linear monadic rewrite systems [Sal88], for linear semi-monadic rewrite systems [CG90], also called (with slight differences) growing systems in [Jac96]. Growing systems are the currently most general class for which the preservation of recognizability is known.

As already pointed out, the decidability of the encompassment theory implies the decidability of ground reducibility. There are several papers written along these lines which will be explained in the next chapter.

Finally, approximations of the reachable terms are computed in [Gen97] using tree automata techniques, which implies the decision of some safety properties.

3.6.7 Other applications

The relationship between finite tree automata and higher-order matching is studied in [CJ97b].

Finite tree automata are also used in logic programming [FSVY91], type reconstruction [Tiu92] and automated deduction [GMW97].

For further applications of tree automata in the direction of program verification, see e.g. chapter 5 of this book or e.g. [Jon87].

Chapter 4

Automata with Constraints

4.1 Introduction

A typical example of a language which is not recognized by a finite tree automaton is the set of terms $\{f(t,t) \mid t \in T(\mathcal{F})\}$, i.e. the set of ground instances of the term $f(x,x)$. The reason is that the two sons of the root are recognized independently and only a fixed finite amount of information can be carried up to the root position, whereas t may be arbitrarily large. Therefore, as seen in the application section of the previous chapter, this imposes some linearity conditions, typically when automata techniques are applied to rewrite systems or to sort constraints. The shift from linear to non linear situations can also be seen as a generalization from tree automata to DAG (directed acyclic graphs) automata. This is the purpose of the present chapter: how is it possible to extend the definitions of tree automata in order to carry over the applications of the previous chapter to (some) non-linear situations?

Such an extension has been studied in the early 80's by M. Dauchet and J. Mongy. They define a class of automata which (when working in a top-down fashion) allow duplications. Considering bottom-up automata, this amounts to check equalities between subtrees. This yields the *RATEG class*. This class is not closed under complement. If we consider its closure, we get the class of automata with equality and disequality constraints. This class is studied in Section 4.2.

Unfortunately, the emptiness problem is undecidable for the class RATEG (and hence for automata with equality and disequality constraints).
Several decidable subclasses have been studied in the literature. The most remarkable ones are:

- The class of *automata with constraints between brothers* which, roughly, allows equality (or disequality) tests only between positions with the same ancestors. For instance, the above set of terms $f(t,t)$ is recognized by such an automaton. This class is interesting because most important properties of tree automata (Boolean closure and decision results) carry over this extension and hence most of the applications of tree automata can be extended, replacing linearity conditions with corresponding restrictions on non-linearities.

  We study this class in Section 4.3.

- The class of *reduction automata* which, roughly, allows arbitrary disequality constraints but only a fixed finite amount of equality constraints on each run of the automaton. For instance the above set of terms $f(t, t)$ also belongs to this class, since recognizing the terms of this set requires only one equality test, performed at the top position. Though closure properties have to be handled with care (with the definition sketched above, the closure by complement is unknown for this class), reduction automata are interesting because for example the set of irreducible terms (w.r.t. an arbitrary, possibly non-linear rewrite system) is recognized by a reduction automaton. Then the decidability of ground reducibility is a direct consequence of emptiness decidability for deterministic reduction automata. There is also a logical counterpart: the *reducibility theory* which is presented in the linear case in the previous chapter and which can be shown decidable in the general case using a similar technique.

  Reduction automata are studied in Section 4.4.

4.2 Automata with Equality and Disequality Constraints

4.2.1 The Most General Class

An ***equality constraint*** (resp. a ***disequality constraint***) is a predicate on $T(\mathcal{F})$ written $\pi = \pi'$ (resp. $\pi \neq \pi'$) where $\pi, \pi' \in \{1, \dots, k\}^*$. Such a predicate is satisfied on a term t, which we write $t \models \pi = \pi'$, if $\pi, \pi' \in \mathcal{P}os(t)$ and $t|_\pi = t|_{\pi'}$ (resp. $\pi \neq \pi'$ is satisfied on t if $\pi = \pi'$ is not satisfied on t).

The satisfaction relation $\models$ is extended as usual to any Boolean combination of equality and disequality constraints. The empty conjunction and disjunction are respectively written $\top$ (true) and $\bot$ (false).

Example 4.2.1. With the above definition, we have: $f(a, f(a, b)) \models 1 = 21$, and $f(a, b) \models 1 \neq 2$.

An ***automaton with equality and disequality constraints*** [is a tuple $(Q, \mathcal{F}, Q_f, \Delta)$ where $\mathcal{F}$ is a finite ranked alphabet, Q is a finite set of states, Q_f is a subset of Q of ***final states*** and Δ is a set of transition rules of of the form:

$$f(q_1, \dots, q_n) \xrightarrow{c} q$$

where $f \in \mathcal{F}$, $q_1, \dots, q_n, q \in Q$, and c is a Boolean combination of equality (and disequality) constraints. The state q is called ***target state*** in the above transition rule. We write for short AWEDC the class of automata with equality and disequality constraints.

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta) \in$ AWEDC. The move relation $\rightarrow_{\mathcal{A}}$ is defined using the satisfaction of equality and disequality constraints: let $t, t' \in T(\mathcal{F} \cup Q)$, then $t \rightarrow_{\mathcal{A}} t'$ if and only there is a context $C \in \mathcal{C}(\mathcal{F} \cup Q)$, some terms $u_1, \dots, u_n \in T(\mathcal{F})$, and a transition rule $f(q_1, \dots, q_n) \xrightarrow{c} q \in \Delta$, such that $t = C[f(q_1(u_1), \dots, q_n(u_n))]$, $t' = C[q(f(u_1, \dots, u_n))]$ and $f(u_1, \dots, u_n) \models c$.

We denote $\xrightarrow{*}_{\mathcal{A}}$ the reflexive and transitive closure of $\rightarrow_{\mathcal{A}}$. As in Chapter 1, we usually write $t \xrightarrow{*}_{\mathcal{A}} q$ instead of $t \xrightarrow{*}_{\mathcal{A}} q(t)$.

An automaton $\mathcal{A} \in$ AWEDC ***accepts*** (or ***recognizes***) a ground term $t \in T(\mathcal{F})$ if $t \xrightarrow{*}_{\mathcal{A}} q$ for some state $q \in Q_f$. More generally, we also say that $\mathcal{A}$ accepts t in state q iff $t \xrightarrow{*}_{\mathcal{A}} q$ (acceptance by $\mathcal{A}$ is the particular case of acceptance by $\mathcal{A}$ in a final state).

The ***language accepted***, or ***recognized***, by an automaton $\mathcal{A} \in$ AWEDC is the set $L(\mathcal{A})$ of terms $t \in T(\mathcal{F})$ that are accepted by $\mathcal{A}$. The set of terms $t \in T(\mathcal{F})$ that are accepted by $\mathcal{A}$ in state q is denoted $L(\mathcal{A}, q)$.

A ***run*** is a mapping ρ from $\mathcal{P}os(t)$ into Δ such that if $t(p) = f$ and the target state of $\rho(p)$ is q, then there is a transition rule $f(q_1, \ldots, q_n) \xrightarrow{c} q$ in Δ such that for all $1 \leq i \leq n$, the target state of $\rho(pi)$ is q_i and $t|_p \models c$.

Note that we do not have here exactly the same definition of a run as in Chapter 1: instead of the state, we keep also the rule which yielded this state. This will be useful in the design of an emptiness decision algorithm for tree automata with equality and disequality constraints.

Example 4.2.2. Balanced complete binary trees over the alphabet f (binary) and a (constant) are recognized by the AWEDC $(\{q\}, \{f, a\}, \{q\}, \Delta)$ where Δ consists of the following rules:

$$\begin{array}{rc}
r_1 : & a \rightarrow q \\
r_2 : & f(q, q) \xrightarrow{1=2} q
\end{array}$$

For example, $t = f(f(a, a), f(a, a))$ is accepted. The mapping which associates r_1 to every position p of t such that $t(p) = a$ and which associates r_2 to every position p of t such that $t(p) = f$ is indeed a successful run: for every position p of t such that $t(p) = f$, $t|_{p \cdot 1} = t_{p \cdot 2}$, hence $t|_p \models 1 = 2$.

Example 4.2.3. Consider the following AWEDC: $(Q, \mathcal{F}, Q_f, \Delta)$ with $\mathcal{F} = \{0, s, f\}$ where 0 is a constant, s is unary and f has arity 4, $Q = \{q_n, q_0, q_f\}$, $Q_f = \{q_f\}$, and Δ consists of the following rules:

$$\begin{array}{rcl rcl}
0 & \rightarrow & q_0 & s(q_0) & \rightarrow & q_n \\
s(q_n) & \rightarrow & q_n & f(q_0, q_0, q_n, q_n) & \xrightarrow{3=4} & q_f \\
f(q_0, q_0, q_0, q_0) & \rightarrow & q_f & f(q_0, q_n, q_0, q_n) & \xrightarrow{2=4} & q_f \\
f(q_f, q_n, q_n, q_n) & \xrightarrow{14=4 \wedge 21=12 \wedge 131=3} & q_f & & &
\end{array}$$

This automaton computes the sum of two natural numbers written in base one in the following sense: if t is accepted by $\mathcal{A}$ then[1] $t = f(t_1, s^n(0), s^m(0), s^{n+m}(0))$ for some t_1 and $n, m \geq 0$. Conversely, for each $n, m \geq 0$, there is a term $f(t_1, s^n(0), s^m(0), s^{n+m}(0))$ which is accepted by the automaton.

For instance the term depicted on Figure 4.1 is accepted by the automaton. Similarly, it is possible to design an automaton of the class AWEDC which

---

[1] $s^n(0)$ denotes $\underbrace{s(\ldots s}_{n}(0)$

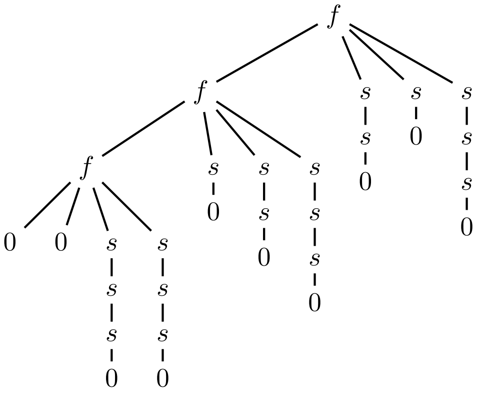


Figure 4.1: A computation of the sum of two natural numbers

"computes the multiplication" (see exercises)

Complexity Measures

In order to evaluate the complexity of operations on automata of the class AWEDC, we need to precise a representation of the automata and estimate the space which is necessary for this representation.

The ***size*** of a Boolean combination of equality and disequality constraints is defined by induction:

- $\|\pi = \pi'\| \stackrel{\text{def}}{=} \|\pi \neq \pi'\| \stackrel{\text{def}}{=} |\pi| + |\pi'|$ ($|\pi|$ is the length of π)
- $\|c \wedge c'\| \stackrel{\text{def}}{=} \|c \vee c'\| \stackrel{\text{def}}{=} \|c\| + \|c'\| + 1$
- $\|\neg c\| \stackrel{\text{def}}{=} \|c\|$

Now, the complexity of deciding whether $t \models c$ depends on the representation of t. If t is represented as a directed acyclic graph (a DAG) with maximal sharing, then this can be decided in $O(\|c\|)$ on a RAM. Otherwise, it requires to compute first this representation of t, and hence can be computed in time at most $O(\|t\| \log \|t\| + \|c\|)$.

From now on, we assume, for complexity analysis, that the terms are represented with maximal sharing in such a way that checking an equality or a disequality constraint on t can be completed in a time which is independent of $\|t\|$.

The ***size*** of an automaton $\mathcal{A} \in$ AWEDC is

$$\|\mathcal{A}\| \stackrel{\text{def}}{=} |Q| + \sum_{f(q_1,\ldots,q_n) \xrightarrow{c} q \in \Delta} n + 2 + \|c\|$$

Determinism

An automaton $\mathcal{A}$ in AWEDC is ***deterministic*** if for every $t \in T(\mathcal{F})$, there is at most one state q such that $t \xrightarrow[\mathcal{A}]{*} q$. It is ***complete*** if for every $t \in T(\mathcal{F})$ there is at least one state q such that $t \xrightarrow[\mathcal{A}]{*} q$.

When every constraint is a tautology, then our definition of automata reduces to the definition of Chapter 1. However, in such a case, the notions of determinacy do not fully coincide, as noticed in Chapter 1, page 23.

4.2.2 Reducing Non-determinism and Closure Properties

Proposition 4.2.4. *For every automaton $\mathcal{A} \in$ AWEDC, there is a complete automaton $\mathcal{A}'$ which accepts the same language as $\mathcal{A}$. The size $\|\mathcal{A}'\|$ is polynomial in $\|\mathcal{A}\|$ and the computation of $\mathcal{A}'$ can be performed in polynomial time (for a fixed alphabet). If $\mathcal{A}$ is deterministic, then $\mathcal{A}'$ is deterministic.*

Proof. The proof is similar as for Theorem 1.1.7: we add a trash state $q_\perp$ and every appropriate transition to the trash state. However, we need to be careful concerning the constraints of the computed transitions in order to preserve the determinism.

The additional transitions are computed as follows: for each function symbol $f \in \mathcal{F}$ and each tuple of states (including the trash state) $q_1, \ldots, q_n$, if there is no transition $f(q_1, \ldots, q_n) \xrightarrow{c} q \in \Delta$, then we simply add the rule $f(q_1, \ldots, q_n) \rightarrow q_\perp$ to Δ. Otherwise, let $f(q_1, \ldots, q_n) \xrightarrow{c_i} s_i$ $(i = 1, \ldots, m)$ be all rules in Δ whose left member is $f(q_1, \ldots, q_n)$. We add the rule $f(q_1, \ldots, q_n) \xrightarrow{c'} q_\perp$ to Δ, where $c' \overset{\text{def}}{=} \neg \bigvee_{i=1}^{m} c_i$. □

Example 4.2.5. In order to complete the AWEDC of Example 4.2.2, we add the following transitions:

$$\begin{array}{rclrcl} f(q,q) & \xrightarrow{1\neq 2} & q_\perp & f(q_\perp,q) & \rightarrow & q_\perp \\ f(q,q_\perp) & \rightarrow & q_\perp & f(q_\perp,q_\perp) & \rightarrow & q_\perp \end{array}$$

Note that the AWEDC obtained is deterministic.

Proposition 4.2.6. *For every automaton $\mathcal{A} \in$ AWEDC, there is a deterministic automaton $\mathcal{A}'$ which accepts the same language as $\mathcal{A}$. The size $\|\mathcal{A}'\|$ is exponential in $\|\mathcal{A}\|$ and $\mathcal{A}'$ can be computed in exponential time. If $\mathcal{A}$ is complete, then $\mathcal{A}'$ is complete.*

Proof. The construction is the same as in Theorem 1.1.9: states of $\mathcal{A}'$ are sets of states of $\mathcal{A}$. Final states of $\mathcal{A}'$ are those sets which contain at least one final state of $\mathcal{A}$. The construction time complexity as well as the size $\mathcal{A}'$ are also of the same magnitude as in Theorem 1.1.9. The only difference is the computation of the constraints in transition rules: if $S_1, \ldots, S_n, S$ are sets of states, in the deterministic automaton, the rule $f(S_1, \ldots, S_n) \xrightarrow{c} S$ is labeled by a constraint c defined by:

$$c \stackrel{\text{def}}{=} \Big(\bigwedge_{q \in S} \bigvee_{\substack{f(q_1,\ldots,q_n) \xrightarrow{c_r} q \in \Delta \\ q_i \in S_i,\ i \leq n}} c_r \Big) \wedge \Big(\bigwedge_{q \notin S} \bigwedge_{\substack{f(q_1,\ldots,q_n) \xrightarrow{c_r} q \in \Delta \\ q_i \in S_i,\ i \leq n}} \neg c_r \Big)$$

Let us prove that a term t is accepted by $\mathcal{A}$ in states $q_1, \ldots, q_k$ (and no other states) if and only if t is accepted by $\mathcal{A}'$ in the state $\{q_1, \ldots, q_k\}$:

Direction $\Rightarrow$. Assume that $t \xrightarrow[\mathcal{A}]{n_i} q_i$ (i.e. $t \xrightarrow[\mathcal{A}]{*} q_i$ in n_i steps), for $i = 1, \ldots, k$. We prove, by induction on $n = \max(n_1, \ldots, n_k)$, that

$$t \xrightarrow[\mathcal{A}']{n} \{q_1, \ldots, q_k\}.$$

If $n = 1$, then t is a constant and $t \to S$ is a rule of $\mathcal{A}'$ where $S = \{q \mid a \xrightarrow[\mathcal{A}]{} q\}$.

Assume now that $n > 1$. Let, for each $i = 1, \ldots, k$,

$$t = f(t_1, \ldots, t_p) \xrightarrow[\mathcal{A}]{m_i} f(q_1^i, \ldots, q_p^i) \xrightarrow[\mathcal{A}]{} q_i$$

and let $f(q_1^i, \ldots, q_p^i) \xrightarrow{c_i} q_i$ be a rule of $\mathcal{A}$ such that $t \models c_i$. By induction hypothesis, each term t_j is accepted by $\mathcal{A}'$ in the states of a set $S_j \supseteq \{q_j^1, \ldots, q_j^k\}$. Moreover, by definition of $S = \{q_1, \ldots, q_k\}$, if $t \xrightarrow[\mathcal{A}]{*} q'$ then $q' \in S$. Therefore, for every transition rule of $\mathcal{A}$ $f(q_1', \ldots, q_p') \xrightarrow{c'} q'$ such that $q' \notin S$ and $q_j \in S_j$ for every $j \leq p$, we have $t \not\models c'$. Then t satisfies the above defined constraint c.

Direction $\Leftarrow$. Assume that $t \xrightarrow[\mathcal{A}']{n} S$. We prove by induction on n that, for every $q \in S$, and for no other state of $\mathcal{A}$, $t \xrightarrow[\mathcal{A}]{n} q$.

If $n = 1$, then S is the set of states q such that $t \xrightarrow[\mathcal{A}]{} q$, hence the property.

Assume now that:

$$t = f(t_1, \ldots, t_p) \xrightarrow[\mathcal{A}']{n} f(S_1, \ldots, S_p) \xrightarrow[\mathcal{A}']{} S.$$

Let $f(S_1, \ldots, S_p) \xrightarrow{c} S$ be the last rule used in this reduction. Then $t \models c$ and, by definition of c, for every state $q \in S$, there is a rule $f(q_1, \ldots, q_n) \xrightarrow{c_r} q$ of $\mathcal{A}$ such that $q_i \in S_i$ for every $i \leq n$ and $t \models c_r$. By induction hypothesis, for each i, $t_i \xrightarrow[\mathcal{A}']{m_i} S_i$ implies $t_i \xrightarrow[\mathcal{A}]{m_i} q_i$ (for some $m_i < n$) and hence $t \xrightarrow[\mathcal{A}]{n} f(q_1, \ldots, q_p) \xrightarrow[\mathcal{A}]{} q$.

Let $q \notin S$ and assume that $t \xrightarrow[\mathcal{A}]{*} q$ and that the last step in this reduction uses the transition rule $f(q_1, \ldots, q_n) \xrightarrow{c_r} q$. By induction hypothesis, every $q_i \in S_i$ (for $i \leq n$). Hence, by definition of the constraint c, $t \not\models c$, and it contradicts $t \xrightarrow[\mathcal{A}']{*} S$.

Thus, by construction of the final states set, a ground term t is accepted by $\mathcal{A}'$ iff t is accepted by $\mathcal{A}$.

Now, we have to prove that $\mathcal{A}'$ is deterministic indeed. Assume that $t \xrightarrow[\mathcal{A}']{*} S$ and $t \xrightarrow[\mathcal{A}']{*} S'$. Assume moreover that $S \neq S'$ and that t is the smallest term (in size) recognized in two different states. Then there exists $S_1, \ldots, S_n$ such that $t \xrightarrow[\mathcal{A}']{*} f(S_1, \ldots, S_n)$ and such that $f(S_1, \ldots, S_n) \xrightarrow{c} S$ and $f(S_1, \ldots, S_n) \xrightarrow{c'} S'$ are transition rules of $\mathcal{A}'$, with $t \models c$ and $t \models c'$. By symmetry, we may assume that there is a state $q \in S$ such that $q \notin S'$. Then, by construction of $\mathcal{A}'$, there are some states $q_i \in S_i$, for every $i \leq n$, and a rule $f(q_1, \ldots, q_n) \xrightarrow{c_r} q$ of $\mathcal{A}$ where c_r occurs positively in c, and is therefore satisfied by t, $t \models c_r$. By construction of the constraint of $\mathcal{A}'$, c_r must occur negatively in the second part of (the conjunction) c'. Therefore, $t \models c'$ contradicts $t \models c_r$. □

Example 4.2.7. Consider the following automaton on the alphabet $\mathcal{F} = \{a, f\}$ where a is a constant and f is a binary symbol: $Q = \{q, q_\bot\}$, $Q_f = \{q\}$ and Δ contains the following rules:

$$\begin{array}{ccc}
a \rightarrow q & f(q,q) \xrightarrow{1=2} q & f(q,q) \rightarrow q_\bot \\
f(q_\bot,q) \rightarrow q_\bot & f(q,q_\bot) \rightarrow q_\bot & f(q_\bot,q_\bot) \rightarrow q_\bot
\end{array}$$

This complete (and non-deterministic) automaton accepts the same languages as the automaton of Example 4.2.2. Then the deterministic automaton computed as in the previous proposition is given by:

$$\begin{array}{cc}
a \rightarrow \{q\} & f(\{q\},\{q\}) \xrightarrow{1=2\wedge\bot} \{q\} \\
f(\{q\},\{q\}) \xrightarrow{1=2} \{q, q_\bot\} & f(\{q\},\{q\}) \xrightarrow{1\neq 2} \{q_\bot\} \\
f(\{q\},\{q_\bot\}) \rightarrow \{q_\bot\} & f(\{q_\bot\},\{q\}) \rightarrow \{q_\bot\} \\
f(\{q_\bot\},\{q_\bot\}) \rightarrow \{q_\bot\} & f(\{q,q_\bot\},\{q\}) \xrightarrow{1=2\wedge\bot} \{q\} \\
f(\{q,q_\bot\},\{q\}) \xrightarrow{1=2} \{q,q_\bot\} & f(\{q,q_\bot\},\{q_\bot\}) \rightarrow \{q_\bot\} \\
f(\{q,q_\bot\},\{q,q_\bot\}) \xrightarrow{1=2} \{q,q_\bot\} & f(\{q,q_\bot\},\{q\}) \xrightarrow{1\neq 2} \{q_\bot\} \\
f(\{q\},\{q,q_\bot\}) \xrightarrow{1=2} \{q,q_\bot\} & f(\{q\},\{q,q_\bot\}) \xrightarrow{1\neq 2} \{q_\bot\} \\
f(\{q,q_\bot\},\{q\}) \xrightarrow{1\neq 2} \{q_\bot\} & f(\{q\},\{q,q_\bot\}) \xrightarrow{1=2\wedge\bot} \{q\} \\
f(\{q,q_\bot\},\{q,q_\bot\}) \xrightarrow{1=2\wedge\bot} \{q\} & f(\{q_\bot\},\{q,q_\bot\}) \rightarrow \{q_\bot\}
\end{array}$$

For instance, the constraint $1{=}2\wedge\bot$ is obtained by the conjunction of the label of $f(q,q) \xrightarrow{1=2} q$ and the negation of the constraint labelling $f(q,q) \rightarrow q_\bot$, (which is $\top$).

Some of the constraints, such as $1{=}2\wedge\bot$ are unsatisfiable, hence the corresponding rules can be removed. If we finally rename the two accepting states $\{q\}$ and $\{q, q_\bot\}$ into a single state q_f (this is possible since by replacing one of these states by the other in any left hand side of a transition rule, we get another transition rule), then we get a simplified version of the deterministic automaton:

$$\begin{array}{rcl@{\qquad}rcl} a & \rightarrow & q_f & f(q_\perp, q_f) & \rightarrow & q_\perp \\ f(q_f, q_f) & \xrightarrow{1=2} & q_f & f(q_f, q_\perp) & \rightarrow & q_\perp \\ f(q_f, q_f) & \xrightarrow{1\neq 2} & q_\perp & f(q_\perp, q_\perp) & \rightarrow & q_\perp \end{array}$$

Proposition 4.2.8. *The class AWEDC is effectively closed by all Boolean operations. Union requires linear time, intersection requires quadratic time and complement requires exponential time. The respective sizes of the AWEDC obtained by these construction are of the same magnitude as the time complexity.*

Proof. The proof of this proposition can be obtained from the proof of Theorem 1.3.1 (Chapter 1, pages 30–31) with straightforward modifications. The only difference lies in the product automaton for the intersection: we have to consider conjunctions of constraints. More precisely, if we have two AWEDC $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_1 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$, we construct an AWEDC $\mathcal{A} = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta)$ such that if $f(q_1, \ldots, q_n) \xrightarrow{c} q \in \Delta_1$ and $f(q'_1, \ldots, q'_n) \xrightarrow{c'} q' \in \Delta_2$, then $f((q_1, q'_1), \ldots, (q_n, q'_n)) \xrightarrow{c \wedge c'} (q, q') \in \Delta$. The AWEDC $\mathcal{A}$ recognizes $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. ☐

4.2.3 Decision Problems

Membership

Proposition 4.2.9. *Given $t \in T(\mathcal{F})$ and $\mathcal{A} \in$AWEDC, deciding whether t is accepted by $\mathcal{A}$ can be completed in polynomial time (linear time for a deterministic automaton).*

Proof. Because of the DAG representation of t, the satisfaction of a constraint $\pi = \pi'$ on t can be completed in time $O(|\pi| + |\pi'|)$. Thus, if $\mathcal{A}$ is deterministic, the membership test can be performed in time $O(\|t\| + \|\mathcal{A}\| + MC)$ where $MC = \max(\|c\| \mid c$ is a constraint of a rule of $\mathcal{A})$. If $\mathcal{A}$ is nondeterministic, the complexity of the algorithm will be $O(\|t\| \times \|\mathcal{A}\| \times MC)$. ☐

Undecidability of Emptiness

Theorem 4.2.10. *The emptiness problem for AWEDC is undecidable.*

Proof. We reduce the Post Correspondence Problem (PCP). If $w_1, \ldots, w_n$ and $w'_1, \ldots, w'_n$ are the word sequences of the PCP problem over the alphabet $\{a, b\}$, we let $\mathcal{F}$ contain h (ternary), a, b (unary) and 0 (constant). Lets recall that the answer for the above instance of the PCP is a sequence $i_1, \ldots, i_p$ (which may contain some repetitions) such that $w_{i_1} \ldots w_{i_p} = w'_{i_1} \ldots w'_{i_p}$.

If $w \in \{a, b\}^*$, $w = a_1 \ldots a_k$ and $t \in T(\mathcal{F})$, we write $w(t)$ the term $a_1(\ldots(a_k(t))\ldots) \in T(\mathcal{F})$.

Now, we construct $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta) \in$ AWEDC as follows:

- Q contains a state q_v for each prefix v of one of the words w_i, w'_i (including q_{w_i} and $q_{w'_i}$ as well as 3 extra states: q_0, q and q_f. We assume that a and b are both prefix of at least one of the words w_i, w'_i. $Q_f = \{q_f\}$.

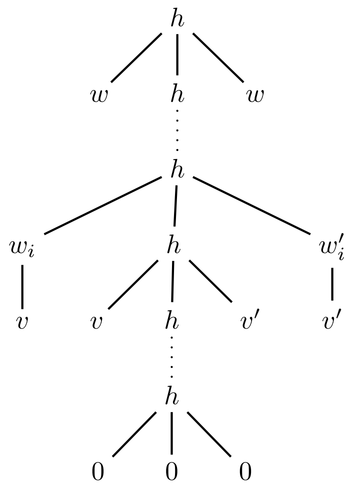


Figure 4.2: An automaton in AWEDC accepting the solutions of PCP

- Δ contains the following rules:

$$\begin{array}{rcllcl}
a(q_0) & \rightarrow & q_a & b(q_0) & \rightarrow & q_b \\
a(q_v) & \rightarrow & q_{a\cdot v} & \multicolumn{3}{l}{\text{if } q_v, q_{a\cdot v} \in Q} \\
b(q_v) & \rightarrow & q_{b\cdot v} & \multicolumn{3}{l}{\text{if } q_v, q_{b\cdot v} \in Q} \\
a(q_{w_i}) & \rightarrow & q_a & b(q_{w_i}) & \rightarrow & q_b \\
a(q_{w'_i}) & \rightarrow & q_a & b(q_{w'_i}) & \rightarrow & q_b
\end{array}$$

Δ also contains the rules:

$$\begin{array}{rcl}
0 & \rightarrow & q_0 \\
h(q_0, q_0, q_0) & \rightarrow & q \\
h(q_{w_i}, q, q_{w'_i}) & \xrightarrow{1\cdot 1^{|w_i|}=2\cdot 1 \wedge 3\cdot 1^{|w'_i|}=2\cdot 3} & q \\
h(q_{w_i}, q, q_{w'_i}) & \xrightarrow{1\cdot 1^{|w_i|}=2\cdot 1 \wedge 3\cdot 1^{|w'_i|}=2\cdot 3 \wedge 1=3} & q_f
\end{array}$$

The rule with left member $h(q_0, q_0, q_0)$ recognizes the beginning of a Post sequence. The rules with left members $h(q_{w_i}, q, q_{w'_i})$ ensure that we are really in presence of a successor in the PCP sequences: the constraint expresses that the subterm at position 1 is obtained by concatenating some w_i with the term at position $2 \cdot 1$ and that the subterm at position 3 is obtained by concatenating w'_i (with the same index i) with the subterm at position $2 \cdot 3$. Finally, entering the final state is subject to the additional constraint $1 = 3$. This last constraint expresses that we went through two identical words with the w_i sequences and the w'_i sequences respectively. (See Figure 4.2). The details that this automaton indeed accepts the solutions of the PCP are left to the reader.

Then the language accepted by $\mathcal{A}$ is empty if and only if the PCP has a solution. Since PCP is undecidable, emptiness of $\mathcal{A}$ is also undecidable. $\Box$

4.3 Automata with Constraints Between Brothers

The undecidability result of the previous section led to look for subclasses which have the desired closure properties, contain (properly) the classical tree automata and still keep the decidability of emptiness. This is the purpose of the class AWCBB.

4.3.1 Definition

An automaton $\mathcal{A} \in$ AWEDC is an ***automaton with constraints between brothers*** if every equality (resp. disequality) constraint has the form $i = j$ (resp. $i \neq j$) where $i, j \in \mathbb{N}_+$. AWCBB is the set of automata with constraints between brothers.

Example 4.3.1. The set of terms $\{f(t,t) \mid t \in T(\mathcal{F})\}$ is accepted by an automaton of the class AWCBB, because the automaton of Example 4.2.2 is in AWCBB indeed.

4.3.2 Closure Properties

Proposition 4.3.2. *AWCBB is a stable subclass of AWEDC w.r.t. Boolean operations (union, intersection, complementation).*

Proof. It is sufficient to check that the constructions of Propositions 4.2.4, 4.2.6 and 4.2.8 preserve the property of being a member of AWCBB. ☐

Recall that the time complexity of each such construction is the same in AWEDC and in the unconstrained case: union and intersection are polynomial, complementation requires determinization and is exponential.

4.3.3 Emptiness Decision

To decide emptiness we would like to design a marking algorithm like the reduction algorithm of Section 1.1. Here the constraints in the transition rules cause additional difficulties. For instance consider the following example.

Example 4.3.3. Let us consider the AWCBB $\mathcal{A}$ which contains two states q and q_1 and the rules:

$$\begin{array}{lllcccc} a & \rightarrow & q & \qquad & f(q,q) & \xrightarrow{1\neq 2} & q_1 \\ b & \rightarrow & q & & & & \end{array}$$

With the reduction algorithm of Section 1.1, we mark the state q, meaning that this state is inhabited by at least one term (for instance a). However, this information is not sufficient to ensure that one term is accepted in q_1. Indeed, in order to use the rule $f(q,q) \xrightarrow{1\neq 2} q_1$, and hence pass the constraint $1 \neq 2$, we need at least two distinct terms in q (for instance a and b). Note that knowing that two terms are accepted in q, we can deduce that at least two terms will be accepted in q_1 (for instance $f(a, b)$ and $f(b, a)$).

The basic idea for the marking algorithm below is that one can bound the number of "marks" (terms) needed for each state. More precisely, if we have enough distinct terms accepted in states $q_1, \ldots, q_n$, and the the transition $f(q_1, \ldots, q_n) \xrightarrow{c} q$ is possible using some of these terms, then we know that we have enough terms accepted in state q. In the above example, we see that the bound is 2. We show in the next lemma that it is actually he maximal arity of a function symbol of the signature. Let us denote $maxar(\mathcal{F}) \overset{\text{def}}{=} \max\{ar(f) \mid f \in \mathcal{F}\}$.

Lemma 4.3.4. *Let $\mathcal{A}$ be a deterministic AWCBB on $\mathcal{F}$, let $f(q_1, \ldots, q_n) \xrightarrow{c} q$ be a transition rule of $\mathcal{A}$ and assume a subset $L_i \subseteq L(\mathcal{A}, q_i)$ for each $i \leq n$. If there exists $i \leq n$ such that $|L_i| \geq maxar(\mathcal{F})$ and there exists at least one term $t = f(t_1, \ldots, t_n)$ with $t_i \in L_i$ for each $i \leq n$ and $t \models c$, then then $|L(\mathcal{A}, q)| \geq maxar(\mathcal{F})$.*

Proof. We assume without loss of generality that c is in disjunctive normal form, and let c' be a conjunction in c of constraints $j = j'$ or $j \neq j'$ such that $t \models c'$ for one t like in the lemma. Let $L_i = \{s_1, \ldots, s_a, \ldots\}$ (with $a = maxar(\mathcal{F})$) and let k be the number of occurrences of q_i in $q_1, \ldots, q_n$. For the sake of simplicity, we assume below that the k occurrence of q_i are $q_1, \ldots, q_k$.

If $k = 1$, we have $f(s_j, t_2, \ldots, t_n) \models c'$ for each $j \leq a$, hence $f(s_j, t_2, \ldots, t_n) \in L(\mathcal{A}, q)$ for each $j \leq a$. Indeed, since $\mathcal{A}$ is deterministic, terms in different states are distinct. Hence, c' cannot contain an equality $1 = j$ with $j > 1$ (otherwise it would contradict $t \models c'$) and every disequality $1 \neq j$ in c' with $j > 1$ is still satisfied by every $f(s_j, t_2, \ldots, t_n)$. Moreover, it is clear that by hypothesis every constraint of the form $j = j'$ or $j \neq j'$ with $j' > j > 1$ is still satisfied by every $f(s_j, t_2, \ldots, t_n)$.

If $k > 1$, as above, there are no equalities $j = j'$ with $1 \leq j \leq k < j' \leq n$ in c' and every disequality $j \neq j'$ $(1 \leq j \leq k < j' \leq n)$ is still satisfied by any term of the form $f(s_{i_1}, \ldots, s_{i_k}, t_{k+1}, \ldots t_n)$ with $i_1, \ldots, i_k \leq a$. It remains to study the constraints of the form $j = j'$ or $j \neq j'$ with $1 \leq j < j' \leq k$. Assume that there are d such disequality constraints. Then there are $\frac{a!}{(a-d+1)!}$ combinations of the form $f(s_{i_1}, \ldots, s_{i_k}, t_{k+1}, \ldots t_n)$ which satisfy c'. Since $d \leq k \leq a$, it follows that $L(\mathcal{A}, q)$ contains at least a terms of this form. ☐

Now, the marking algorithm for emptiness decision is the following:

```
input: AWCBB $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
begin
    – Marked is a mapping which associates each state with a set of
    terms accepted in that state.
        Set Marked to the function which maps each state to the $\emptyset$
        repeat
            Set Marked($q$) to Marked($q$) $\cup \{t\}$
            where
                $f \in \mathcal{F}_n$, $t_1 \in$ Marked($q_1$), ..., $t_n \in$ Marked($q_n$),
            $f(q_1,\dots,q_n) \xrightarrow{c} q \in \Delta$,
            $t = f(t_1,\dots,t_n)$ and $t \models c$,
            $|Marked(q)| \leq maxar(\mathcal{F}) - 1$,
        until no term can be added to any $Marked(q)$
        output: true if, for every state $q_f \in Q_f$, $Marked(q_f) = \emptyset$.
    end
```

Theorem 4.3.5. *Emptiness can be decided in polynomial time for deterministic automata in AWCBB.*

Proof. We use the above algorithm. Lemma 4.3.4 and the determinacy hypothesis ensures that it is sufficient to keep at most $maxar(\mathcal{F})$ terms in each state. □

Theorem 4.3.6. *The emptiness problem is EXPTIME-complete for (non-deterministic) automata in AWCBB.*

Proof. For non-deterministic automata, an exponential time algorithm is derived from Proposition 4.2.6 (determinization) and Theorem 4.3.5.

For the EXPTIME-hardness, we may reduce to non-emptiness decision for non-deterministic AWCBB the problem of intersection non-emptiness (for standard tree automata), which is known to be EXPTIME-complete, see Section 1.7.

Let $\mathcal{A}_1$,…,$\mathcal{A}_n$ be tree automata over $\mathcal{F}$. We want to decide whether $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ is empty or not.

We may assume without loss of generality that the states sets of $\mathcal{A}_1$,…,$\mathcal{A}_n$ (called respectively Q_1,…,Q_n) are pairwise disjoint, and that every $\mathcal{A}_i$ has a single final state called q_i^f. We also assume that $n = 2^k$ for some integer k. If this is not the case, let k be the smallest integer i such that $n < 2^i$ and let $n' = 2^k$. We consider a second instance of the above problem: $\mathcal{A}'_1$,…,$\mathcal{A}'_{n'}$ where

$\mathcal{A}'_i = \mathcal{A}_i$ for each $i \leq n$.

$\mathcal{A}'_i = (\{q\}, \mathcal{F}, \{q\}, \{f(q,\dots,q) \to q \mid f \in \mathcal{F}\})$ for each $n < i \leq n'$.

Note that the tree automaton in the second case is universal, i.e. it accepts every term of $T(\mathcal{F})$. Hence, the answer is "yes" for $\mathcal{A}'_1$,…,$\mathcal{A}'_{n'}$ iff it is "yes" for $\mathcal{A}_1$,…,$\mathcal{A}_n$.

Now, we add a single new binary symbol g to $\mathcal{F}$, getting $\mathcal{F}'$, and consider an AWCBB $\mathcal{A}$ whose language is empty iff $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n) = \emptyset$. The idea is that, while reading a term t bottom-up, $\mathcal{A}$ will simulate the computation of each $\mathcal{A}_i$ in several subterms $t_1,\dots,t_n$ of t and will test later on (with equality constraints) that these terms t_i are all equal. For instance, for $n = 4$, $\mathcal{A}$ accepts $g(g(t_1,t_2), g(t_3,t_4))$ iff for all $i = 1..4$, $\mathcal{A}_i$ accepts t_i and moreover $t_1 = t_2$ and $g(t_1,t_2) = g(t_3,t_4)$ (hence $t_1 = t_3 = t_4$).

$$\mathcal{A} = (\bigcup_{i=1}^{n} Q_i \uplus \{q_1,\dots,q_{\frac{n}{2}}\}, \mathcal{F}', \{q_1\}, \Delta)$$

where q_1,…,$q_{\frac{n}{2}}$ are new states, and the transition of Δ are:

- every transition rule of $\mathcal{A}_1,\dots,\mathcal{A}_n$

- for each i, $1 \leq i \leq \frac{n}{2}$, $g(q_{2i}, q_{2i+1}) \xrightarrow{1=2} q_i$,
- for each i, $1 \leq i \leq \frac{n}{2}$, $g(q^f_{2i}, q^f_{2i+1}) \xrightarrow{1=2} q_i$.

Note that $\mathcal{A}$ is non-deterministic, even if every $\mathcal{A}_i$ is deterministic.

We can show by induction on k ($n = 2^k$) that the answer to the intersection non-emptiness problem is "yes" iff the language recognized by $\mathcal{A}$ is not empty. Moreover, the size of $\mathcal{A}$ is linear in the size of the initial problem and $\mathcal{A}$ is constructed in a time which is linear in its size. This proves the EXPTIME-hardness of emptiness decision for AWCBB. □

4.3.4 Applications

The main difference between AWCBB and NFTA is the non-closure of AWCBB under projection and cylindrification. Actually, the shift from automata on trees to automata on tuples of trees cannot be extended to the class AWCBB. For instance, choosing $\{0, 1\}^2$ for a signature, we have $00(00, 01) \models 1 \neq 2$, but this constraint is no more valid after projection of this term on the first component.

As long as we are interested in automata recognizing sets of trees, all results on NFTA (and all applications) can be extended to the class AWCBB (with a bigger complexity). For instance, Theorem 3.4.1 (sort constraints) can be extended to interpretations of sorts as languages accepted by AWCBB. Proposition 3.4.3 (encompassment) can be easily generalized to the case of non-linear terms in which non-linearities only occur between brother positions, provided that we replace NFTA with AWCBB. Theorem 3.4.5 can also be generalized to the reducibility theory with predicates $\trianglelefteq_t$ where t is non-linear terms, provided that non-linearities in t only occur between brother positions. However, we can no longer invoke an embedding into WSkS. The important point is that the reductibility theory only requires the weak notion of recognizability on tuples ($Rec_\times$). Hence we do not need automata on tuples, but only tuples of automata. As an example of application, we get a decision algorithm for ground reducibility of a term t w.r.t. left hand sides $l_1, \ldots, l_n$, provided that all non-linearities in $t, l_1, \ldots, l_n$ occur at brother positions: simply compute the automata $\mathcal{A}_i$ accepting the terms that encompass l_i and check that $L(\mathcal{A}) \subseteq L(\mathcal{A}_1) \cup \ldots \cup L(\mathcal{A}_n)$.

Finally, the application on reduction strategies does not carry over to the case of non-linear terms because it really needs automata on tuples.

4.4 Reduction Automata

As we have seen above, the first-order theory of finitely many unary encompassment predicates $\trianglelefteq_{t_1}, \ldots, \trianglelefteq_{t_n}$ (reducibility theory) is decidable when non-linearities in the terms t_i are restricted to brother positions. What happens when we drop the restrictions and consider arbitrary terms $t_1, \ldots, t_n$? It turns out that the theory remains decidable, as we will see. Intuitively, we make impossible counter examples like the one in the proof of Theorem 4.2.10 (stating undecidability of the emptiness problem for AWEDC) with an additional condition that using the automaton which accepts the set of terms encompassing t, we may only check for a bounded number of equalities along each branch. That is the idea of the next definitions of reduction automata.

4.4.1 Definition

A ***reduction automaton*** $\mathcal{A}$ is a member of AWEDC such that there is an ordering on the states of $\mathcal{A}$ such that, for each rule $f(q_1, \ldots, q_n) \xrightarrow{c} q$, q is an upper bound of $q_1, \ldots, q_n$ and it is moreover a strict upper bound if c contains an equality constraint.

In case of an automaton with ϵ-transitions $q \to q'$ we also require q' to be not larger than q.

Example 4.4.1. Consider the set of terms on the alphabet $\mathcal{F} = \{a, g\}$ encompassing $g(g(x, y), x)$. It is accepted by the following reduction automaton, the final state of which is q_f and q_f is minimal in the ordering on states.

$$\begin{array}{rclrcl}
a & \to & q_\top & g(q_\top, q_\top) & \to & q_{g(x,y)} \\
g(q_\top, q_{g(x,y)}) & \to & q_{g(x,y)} & & & \\
g(q_{g(x,y)}, q_\top) & \xrightarrow{11=2} & q_f & g(q_{g(x,y)}, q_\top) & \xrightarrow{11\neq 2} & q_{g(x,y)} \\
g(q_{g(x,y)}, q_{g(x,y)}) & \xrightarrow{11=2} & q_f & g(q_{g(x,y)}, q_{g(x,y)}) & \xrightarrow{11\neq 2} & q_{g(x,y)} \\
g(q, q_f) & \to & q_f & g(q_f, q) & \to & q_f \\
\multicolumn{6}{l}{\text{where } q \in \{q_\top, q_{g(x,y)}, q_f\}}
\end{array}$$

This construction can be generalized, along the lines of the proof of Proposition 3.4.3 (page 97):

Proposition 4.4.2. *The set of terms encompassing a term t is accepted by a deterministic and complete reduction automaton. The size of this automaton is polynomial in $\|t\|$ as well as the time complexity for its construction.*

4.4.2 Closure Properties

As usual, we are now interested in closure properties:

Proposition 4.4.3. *The class of reduction automata is closed under union and intersection.*

Proof. The constructions for union and intersection are the same as in the proof of Proposition 4.2.8, and therefore, the respective time complexity and sizes are the same. The proof that the class of reduction automata is closed under these constructions is left as an exercise. ☐

It is still unknown whether the class of reduction automata is closed under complementation or not. As we shall see below, it is not possible to reduce the non-determinism for reduction automata.

However, we have a weak version of stability:

Proposition 4.4.4.

- *With each reduction automaton, we can associate a complete reduction automaton which accepts the same language. Moreover, this construction preserves the determinism.*

- *The class of complete deterministic reduction automata is closed under complement.*

4.4.3 Emptiness Decision

Theorem 4.4.5. *Emptiness is decidable for the class of complete and deterministic reduction automata.*

The proof of this result is quite complicated and gives quite high upper bounds on the complexity (a tower of several exponentials). Hence, we are not going to reproduce it here. Let us only sketch how it works.

The idea is to use a pumping lemma in order to show that if there exists a term recognized by a given reduction automaton $\mathcal{A}$, there exists one recognized term smaller in size than a certain bound (which depends on $\mathcal{A}$). Hence, we need to show that every term bigger than the bound can be reduced by pumping. The equality and disequality constraints cause additional difficulties in the design of a pumping lemma. For instance consider the following example.

Example 4.4.6. Let us consider a reduction automaton $\mathcal{A}$ with the rules:

$$a \rightarrow q, \quad b \rightarrow q, \quad f(q,q) \xrightarrow{1 \neq 2} q$$

Now consider the term $t = f(f(a,b),b)$ which is accepted by the automaton. The terms $f(a,b)$ and b yield the same state q. Hence, if there was no disequality constraints in $\mathcal{A}$ (i.e. if $\mathcal{A}$ were a classical finite tree automaton), we may replace $f(a,b)$ with b in t and get a smaller term $f(b,b)$ which is still accepted by $\mathcal{A}$. However, this is not the case here since $f(b,b)$ which is not accepted because $f(b,b) \not\models 1 \neq 2$. The reason of this phenomenon is easy to understand: some constraint which was satisfied before the pumping is no longer valid after the pumping.

Consider some term t bigger than the bound, and accepted by $\mathcal{A}$ with a run r. We want to perform a pumping in r giving a successful run r' on a term t' strictly smaller than t. Here the problem is to preserve the satisfaction of equality and disequality constraints along term replacements.

Preserving the equality constraints. Concerning equality constraints, we define an equivalence relation between positions (of equal subtrees). It depends on the run r considered. We have the following properties for the equivalence classes:

first, their cardinality can be bounded by a number which only depends on the automaton, because of the condition with the ordering on states (this is actually not true for the class AWCBB).

second, we can compute a bound b_2 (which only depends on the automaton) such that the difference of the lengths of two equivalent positions is smaller than b_2.

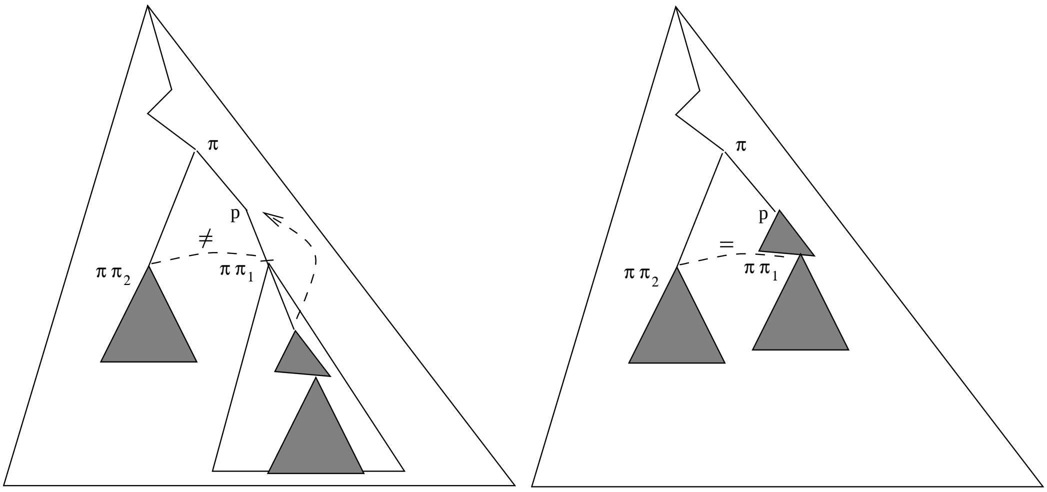


Figure 4.3: A close equality is created

Nevertheless, equalities are not a real problem, as soon as the automaton is deterministic. Indeed, pumping can then be defined on equivalence classes of positions.

If the automaton is not deterministic, we cannot anymore guarantee that we reach the same state at two equivalent positions. Actually, defining a pumping for non deterministic reduction automata is not possible, according to Theorem 4.4.7 below.

Preserving the disequality constraints. Handling disequalities requires more care; the number of distinct subterms in a minimal accepted term cannot be bounded (in contrast, this number can be bounded by $|Q| \times N$, where N is the maximal arity of a function symbol, for AWCBB).

The problem is the possible "overlap" of disequalities checked by the automaton. As in Example 4.4.6, a pumping may yield a term which is no longer accepted, since a disequality checked somewhere in the term is no longer satisfied. In such a case, we say that the pumping *creates an equality.* Then, we distinguish two kinds of equalities created by a pumping: the ***close equalities*** and the ***remote equalities***. Roughly, an equality created by a pumping $(t[v(u)]_p, t[u]_p)$ is a pair of positions $(\pi \cdot \pi_1, \pi \cdot \pi_2)$ of $t[v(u)]_p$ which was checked for disequality by the run ρ at position π on $t[v(u)]_p$ and such that $t[u]_p|_{\pi\cdot\pi_1} = t[u]_p|_{\pi\cdot\pi_2}$ (π is the longest common prefix to both members of the pair). This equality $(\pi \cdot \pi_1, \pi \cdot \pi_2)$ is a close equality if $\pi \leq p < \pi \cdot \pi_1$ or $\pi \leq p < \pi \cdot \pi_2$. Otherwise ($p \geq \pi \cdot \pi_1$ or $p \geq \pi \cdot \pi_2$), it is a remote equality. The different situations are depicted on Figures 4.3 and 4.4.

One possible proof sketch is

- First show that it is sufficient to consider equalities that are created at positions around which the states are incomparable w.r.t. >
- Next, show that, for a deep enough path, there is at least one pumping

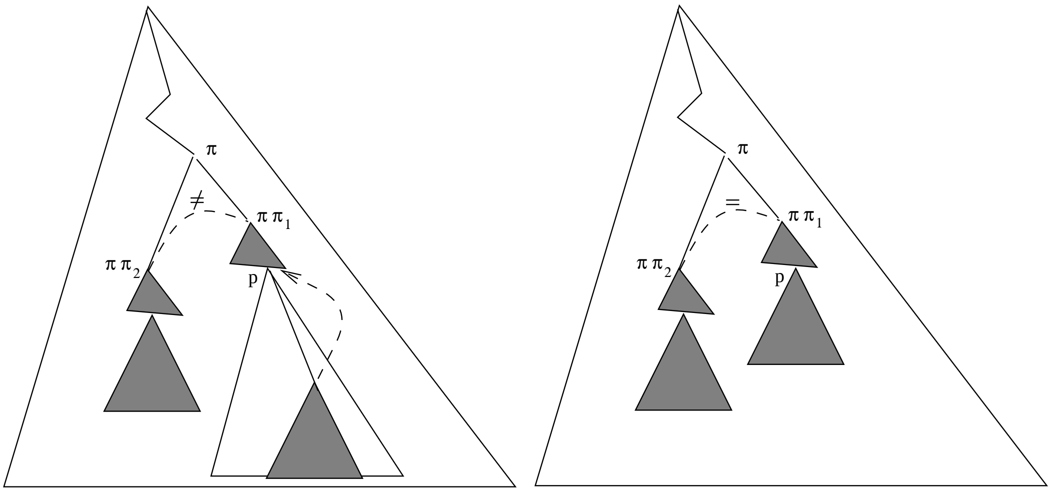


Figure 4.4: A remote equality is created

which does not yield a close equality (this makes use of a combinatorial argument; the bound is an exponential in the maximal size of a constraint).

- For remote equalities, pumping is not sufficient. However, if some pumping creates a remote equality anyway, this means that there are "big"equal terms in t. Then we switch to another branch of the tree, combining pumping in both subtrees to find one (again using a combinatorial argument) such that no equality is created.

Of course, this is a very sketchy proof. The reader is referred to the bibliography for more information about the proof.

Theorem 4.4.7. *Emptiness is not decidable for (non deterministic) reduction automata.*

Proof. We reduce the halting problem for 2-counters machines. Let us consider a 2-counters machine with states p_0 (initial), p_1, …, p_k (p_k is the unique final state, and we assume *wlog* that it cannot be reentered by $\mathcal{M}$). The transitions of $\mathcal{M}$ have the form: $p(x_1, x_2) \xrightarrow{c} p'(x'_1, x'_2)$ where p, p' are states of $\mathcal{M}$ and x_1, x_2, x'_1, x'_2 are the respective contents of the 2 counters, where x'_i is $x_i + 1$ or $x_i - 1$ or x_i. The condition c is a combination of $x_i = 0$ and $x_i \neq 0$ $(i = 1, 2)$. The problem is to decide whether $p_0(0, 0) \xrightarrow[\mathcal{M}]{*} p_k(0, 0)$. We reduce it to the emptiness problem for a reduction automaton which recognizes the term representations of successful computations of $\mathcal{M}$.

We consider a term representation of computations of $\mathcal{M}$ with some redundancy. For this purpose, we use the following signature:

$$\mathcal{F} = \{0 : 0, \# : 0, s : 1, p_0, \dots, p_k : 2, g : 2, h : 2, h' : 2\}$$

A configuration $p(n, m)$ of $\mathcal{M}$ is represented by the term $p(s^n(0), s^m(0))$.

A transition of $\mathcal{M}$ between a configuration c and a configuration c' is represented by a term $g(c, h(g(c', y), y))$, where y represents the rest of a computation of $\mathcal{M}$. Some examples of transitions are given in Figure 4.5.

The term in Figure 4.6 represents a computation sequence of configurations $c_0, \ldots, c_n$, with $c_0 = p_0(0, 0)$ and $c_n = p_k(0, 0)$ and the transition of $\mathcal{M}$ between c_i and c_{i+1} is denoted T_i. In Figure 4.6, the nodes are decorated with the states of $\mathcal{A}$ in which they are accepted. We describe below these states and the transitions of $\mathcal{A}$ which permit the recognition of such a term.

The states of $\mathcal{A}$ are $q_{\#}$ (for #), q_0 (for 0) and q_1 (for strictly positive integers), a universal state $q_\forall$ (in which any term is accepted), and, for every transition $T : \ell \to r$ of $\mathcal{M}$, some states q_ℓ, q_r, q_{gr}, q_{hr}, and q_T. Finally, $\mathcal{A}$ contains also a state q for chaining the transitions of $\mathcal{M}$ and a final state $q_\mathfrak{f}$.

The transitions for recognizing integers are simply:

$$0 \to q_0, s(q_0) \to q_1, s(q_1) \to q_1$$

We have also some transitions ensuring that every term is accepted in the universal state $q_\forall$: $0 \to q_\forall$, $\# \to q_\forall$, $s(q_\forall) \to q_\forall$, $p_0(q_\forall, q_\forall) \to q_\forall, \ldots$

We associate five transitions of $\mathcal{A}$ to every transition T of $\mathcal{M}$. Let us describe below three examples of this construction (the other cases are similar).

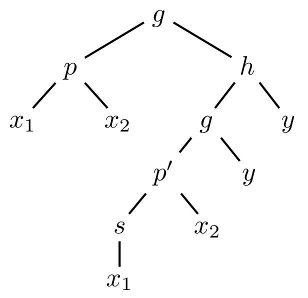
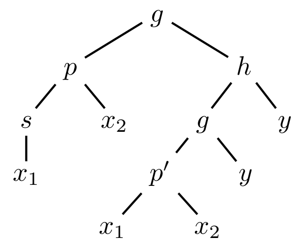
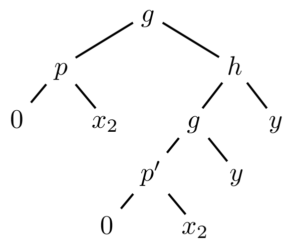
$p(x_1, x_2) \to p'(x_1 + 1, x_2)$	$p(x_1, x_2) \xrightarrow{x_1 \neq 0} p'(x_1 - 1, x_2)$	$p(x_1, x_2) \xrightarrow{x_1 = 0} p'(x_1, x_2)$
		

Figure 4.5: Transitions of the 2-counter machine

For the first transition $T_1 = \ell_1 \to r_1$ in Figure 4.5 (with $\ell_1 = p(x_1, x_2)$ and $r_1 = p'(x_1 + 1, x_2)$), we have the following transitions of the reduction automata $\mathcal{A}$ (q_{01} is an abbreviation for q_0 or q_1):

$$\begin{array}{rclcrcl} p(q_{01}, q_{01}) & \to & q_{\ell_1} & & p'(q_1, q_{01}) & \to & q_{r_1} \\ g(q_{r_1}, q_\forall) & \to & q_{gr_1} & & h(q_{gr_1}, q_\forall) & \to & q_{hr_1} \\ \multicolumn{7}{c}{g(q_{\ell_1}, q_{hr_1}) \xrightarrow{11=21111 \wedge 12=2112 \wedge 22=212} q_{T_1}} \end{array}$$

These transitions permit the recognition by $\mathcal{A}$ of a term of the form described in Figure 4.5. Note that both occurrences of y in this figure are recognized in the universal state $q_\forall$. It means that nothing is checked in these subterms. The constraints in the last transition above guaranty the equality between the two occurrences of x_1, x_2 and y, respectively.

The transition $T_2 = \ell_2 \rightarrow r_2$ in Figure 4.5 (with $\ell_2 = p(x_1, x_2)$ and $r_2 = p'(x_1 - 1, x_2))$ is characterized by the following transitions of $\mathcal{A}$:

$$\begin{array}{rclrcl} p(q_1, q_{01}) & \rightarrow & q_{\ell_2} & p'(q_{01}, q_{01}) & \rightarrow & q_{r_2} \\ g(q_{r_2}, q_\forall) & \rightarrow & q_{gr_2} & h(q_{gr_2}, q_\forall) & \rightarrow & q_{hr_2} \end{array}$$

$$g(q_{\ell_2}, q_{hr_2}) \xrightarrow{111=2111 \wedge 12=2112 \wedge 22=212} q_{T_2}$$

For the last the transition $T_3 = \ell_3 \rightarrow r_3$ in Figure 4.5 (with $\ell_3 = p(0, x_2)$ and $r_3 = p'(0, x_2))$ we have the following transitions of $\mathcal{A}$:

$$\begin{array}{rclrcl} p(q_0, q_{01}) & \rightarrow & q_{\ell_3} & p'(q_0, q_{01}) & \rightarrow & q_{r_3} \\ g(q_{r_3}, q_\forall) & \rightarrow & q_{gr_3} & h(q_{gr_3}, q_\forall) & \rightarrow & q_{hr_3} \end{array}$$

$$g(q_{\ell_3}, q_{hr_3}) \xrightarrow{12=2112 \wedge 22=212} q_{T_3}$$

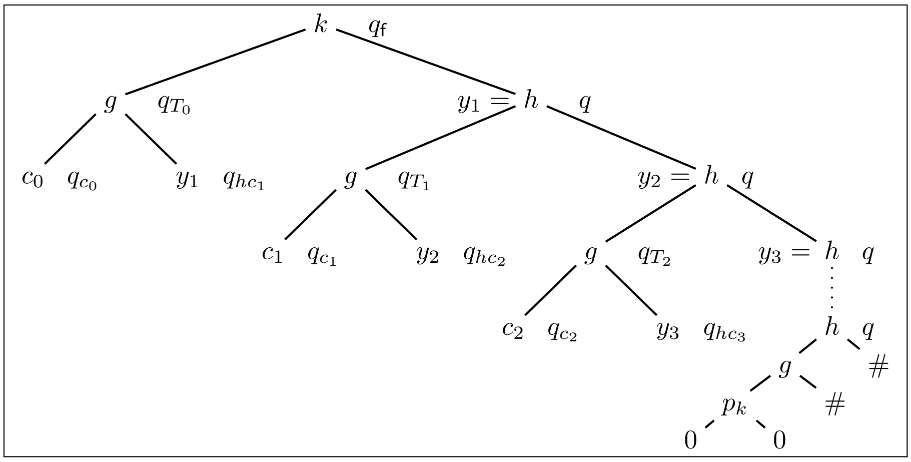


Figure 4.6: A computation of $\mathcal{A}$

In order to recognize with $\mathcal{A}$ the chaining of transitions depicted in Figure 4.6, we use the state q and, for each transition T of $\mathcal{M}$, following unconstrained transition:

$$h(q_T, q) \rightarrow q$$

We have in $\mathcal{A}$ a special constrained transition starting from q_{T_0}, where T_0 is the unique transition of $\mathcal{M}$ starting from the initial configuration $c_0 = p_0(0, 0)$, ($\mathcal{M}$ is assumed deterministic). Note that in this transition we have a symbol h' in the head, instead of a h:

$$h'(q_{T_0}, q) \xrightarrow{12=2} q_f$$

Note that the equality constraint $12 = 2$, together with the above constraints $22 = 212$ in the rules leading to states q_T (corresponding to the equality of both y in Figure 4.5) ensure, by transitivity, the equality of the two subterms denoted y_1 (and y_2 *etc*) in Figure 4.6.

Finally, we have four unconstrained transitions of $\mathcal{A}$ whose purpose is to initiate the bottom-up computation of the automaton with the final configuration

$c_k = p_k(0,0)$ of $\mathcal{M}$.

$$\begin{array}{rclcrcl} \# & \rightarrow & q_{\#} & \quad & p_k(q_0,q_0) & \rightarrow & q_{c_k} \\ g(q_{c_k},q_{\#}) & \rightarrow & q_{gc_k} & & h(q_{gc_k},q_{\#}) & \rightarrow & q \end{array}$$

Note that the equality tests are performed only at nodes labelled with the symbol g or h', hence the number of such tests is at most 2 on any path of the tree in Figure 4.6. We let $q_{\mathsf{f}} > q_{T_0}$, $q_{\mathsf{f}} > q$ and $q_T > q_\ell$, $q_T > q_{hr}$ for every transition $T = \ell \rightarrow r$ of $\mathcal{M}$ and every associated states q_T, q_ℓ, q_{hr}. This way, we have defined above a reduction automata $\mathcal{A}$. It is non-deterministic: the term y_1 in Figure 4.6 is recognized in both states q_{hc_1} and q.

To conclude, it can be shown that $\mathcal{M}$ halts on $p_{\mathsf{f}}(0,0)$ starting from $p_0(0,0)$ iff $L(\mathcal{A}, q_{\mathsf{f}}) \neq \emptyset$. ☐

Corollary 4.4.8. *Reduction automata cannot be determinized.*

Proof. By Theorems 4.4.7 and 4.4.5. ☐

4.4.4 Finiteness Decision

The following result is quite difficult to establish. We only mention it for sake of completeness.

Theorem 4.4.9. *Finiteness of the language is decidable for the class of reduction automata.*

4.4.5 Term Rewriting Systems

There is a strong relationship between reduction automata and term rewriting. We mention them for readers interested in that topic.

Proposition 4.4.10. *Given a term rewriting system $\mathcal{R}$, the set of ground $\mathcal{R}$-normal forms is recognizable by a reduction automaton, the size of which is exponential in the size of $\mathcal{R}$. The time complexity of the construction is exponential.*

Proof. The set of $\mathcal{R}$-reducible ground terms can be defined as the union of sets of ground terms encompassing the left members of rules of $\mathcal{R}$. Thus, by Propositions 4.4.2 and 4.4.3 the set of $\mathcal{R}$-reducible ground terms is accepted by a deterministic and complete reduction automaton. For the union, we use the product construction, preserving determinism (see the proof of Theorem 1.3.1, Chapter 1) with the price of an exponential blowup. The set of ground $\mathcal{R}$-normal forms is the complement of the set of ground $\mathcal{R}$-reducible terms, and it is therefore accepted by a reduction automaton, according to Proposition 4.4.4. ☐

Thus, we have the following consequence of Theorems 4.4.9 and 4.4.5.

Corollary 4.4.11. *Emptiness and finiteness of the language of ground $\mathcal{R}$-normal forms is decidable for every term rewriting system $\mathcal{R}$.*

Let us cite another important result concerning recognizability of sets normal forms.

Theorem 4.4.12. *Given a term rewriting system, it is decidable whether its set of ground normal forms is a recognizable tree language.*

4.4.6 Application to the Reducibility Theory

Consider the reducibility theory of Section 3.4.2: there are unary predicate symbols $\trianglelefteq_t$ which are interpreted as the set of terms which encompass t. Let us accept now non-linear terms t as indices.

Propositions 4.4.2, and 4.4.3, and 4.4.4 and Theorem 4.4.5 yield the following result:

Theorem 4.4.13. *The reducibility theory associated with any sets of terms is decidable.*

And, as in the previous chapter, we have, as an immediate corollary:

Corollary 4.4.14. *Ground reducibility is decidable.*

4.5 Other Decidable Subclasses

Complexity issues and restricted classes. There are two classes of automata with equality and disequality constraints for which tighter complexity results are known:

- For the class of automata containing only disequality constraints, emptiness can be decided in deterministic exponential time. For any term rewriting system $\mathcal{R}$, the set of ground $\mathcal{R}$-normal forms is still recognizable by an automaton of this subclass of reduction automata.
- For the class of deterministic reduction automata for which the constraints “cannot overlap”, emptiness can be decided in polynomial time.

Combination of AWCBB and reduction automata. If you relax the condition on equality constraints in the transition rules of reduction automata so as to allow constraints between brothers, you obtain the biggest known subclass of AWEDC with a decidable emptiness problem.

Formally, these automata, called ***generalized reduction automata***, are members of AWEDC such that there is an ordering on the states set such that for each rule $f(q_1, \ldots, q_n) \xrightarrow{c} q$, q is an upper bound of $q_1, \ldots, q_n$ and it is moreover a strict upper bound if c contains an equality constraint $\pi_1 = \pi_2$ with $|\pi_1| > 1$ or $|\pi_2| > 1$.

The closure and decidability results for reduction automata may be transposed to generalized reduction automata, with though a longer proof for the emptiness decision. Generalized reduction automata can thus be used for the decision of reducibility theory extended by some restricted sort declarations. In this extension, additionally to encompassment predicates $\trianglelefteq_t$, we allow a family of unary sort predicates $. \in S$, where S is a sort symbol. But, sort declarations are limited to atoms of the form $t \in S$ where where non linear variables in t only occur at brother positions. This fragment is decidable by an analog of Theorem 4.4.13 for generalized reduction automata.

4.6 Exercises

Exercise 4.1. 1. Show that the automaton $\mathcal{A}_+$ of Example 4.2.3 accepts only terms of the form $f(t_1, s^n(0), s^m(0), s^{n+m}(0))$

2. Conversely, show that, for every pair of natural numbers (n, m), there exists a term t_1 such that $f(t_1, s^n(0), s^m(0), s^{n+m}(0))$ is accepted by $\mathcal{A}_+$.
3. Construct an automaton $\mathcal{A}_\times$ of the class AWEDC which has the same properties as above, replacing $+$ with $\times$
4. Give a proof that emptiness is undecidable for the class AWEDC, reducing Hilbert's tenth problem.

Exercise 4.2. Give an automaton of the class AWCBB which accepts the set of terms t (over the alphabet $\{a(0), b(0), f(2)\}$) having a subterm of the form $f(u, u)$. (i.e. the set of terms that are reducible by a rule $f(x, x) \to v$).

Exercise 4.3. Show that the class AWCBB is not closed under linear tree homomorphisms. Is it closed under inverse image of such morphisms?

Exercise 4.4. Give an example of two automata in AWCBB such that the set of pairs of terms recognized respectively by the automata is not itself a member of AWCBB.

Exercise 4.5. (Proposition 4.4.3) Show that the class of (languages recognized by) reduction automata is closed under intersection and union.

Exercise 4.6. Show that the set of balanced term on alphabet $\{a, f\}$ as well as its complement are both recognizable by reduction automata.

Exercise 4.7. Show that the class of languages recognized by reduction automata is preserved under linear tree homomorphisms. Show however that this is no longer true for arbitrary tree homomorphisms.

Exercise 4.8. Let $\mathcal{A}$ be a reduction automaton. We define a ternary relation $q \xrightarrow{w} q'$ contained in $Q \times \mathbb{N}^* \times Q$ as follows:

- for $i \in \mathbb{N}$, $q \xrightarrow{i} q'$ if and only if there is a rule $f(q_1, \dots, q_n) \xrightarrow[\mathcal{A}]{c} q'$ with $q_i = q$
- $q \xrightarrow{i \cdot w} q'$ if and only if there is a state q'' such that $q \xrightarrow{i} q''$ and $q'' \xrightarrow{w} q'$.

Moreover, we say that a state $q \in Q$ is a *constrained state* if there is a rule $f(q_1, \dots, q_n) \xrightarrow[\mathcal{A}]{c} q$ in $\mathcal{A}$ such that c is not a valid constraint.

We say that the *the constraints of $\mathcal{A}$ cannot overlap* if, for each rule $f(q_1, \dots, q_n) \xrightarrow{c} q$ and for each equality (resp. disequality) $\pi = \pi'$ of c, there is no strict prefix p of π and no constrained state q' such that $q' \xrightarrow{p} q$.

1. Consider the rewrite system on the alphabet $\{f(2), g(1), a(0)\}$ whose left members are $f(x, g(x)), g(g(x)), f(a, a)$. Compute a reduction automaton, whose constraints do not overlap and which accepts the set of irreducible ground terms.
2. Show that emptiness can be decided in polynomial time for reduction automata whose constraints do not overlap. (Hint: it is similar to the proof of Theorem 4.3.5.)
3. Show that any language recognized by a reduction automaton whose constraints do not overlap is an homomorphic image of a language in the class AWCBB. Give an example showing that the converse is false.

Exercise 4.9. Prove the Proposition 4.4.2 along the lines of Proposition 3.4.3.

Exercise 4.10. The purpose of this exercise is to give a construction of an automaton with disequality constraints (no equality constraints) whose emptiness is equivalent to the ground reducibility of a given term t with respect to a given term rewriting system $\mathcal{R}$.

1. Give a direct construction of an automaton with disequality constraints $\mathcal{A}_{\mathrm{NF}(\mathcal{R})}$ which accepts the set of irreducible ground terms
2. Show that the class of languages recognized by automata with disequality constraints is closed under intersection. Hence the set of irreducible ground instances of a linear term is recognized by an automaton with disequality constraints.
3. Let $\mathcal{A}_{\mathrm{NF}(\mathcal{R})} = (Q_{\mathrm{NF}}, \mathcal{F}, Q^f_{\mathrm{NF}}, \Delta_{\mathrm{NF}})$. We compute $\mathcal{A}_{\mathrm{NF},t} \stackrel{\mathrm{def}}{=} (Q_{\mathrm{NF},t}, \mathcal{F}, Q^f_{\mathrm{NF},t}, \Delta_{\mathrm{NF},t})$ as follows:
    - $Q_{\mathrm{NF},t} \stackrel{\mathrm{def}}{=} \{t\sigma|_p \mid p \in Pos(t)\} \times Q_{\mathrm{NF}}$ where σ ranges over substitutions from $NLV(t)$ (the set of variables occurring at least twice in t) into Q^f_{NF}.
    - For all $f(q_1, \ldots, q_n) \xrightarrow{c} q \in \Delta_{\mathrm{NF}}$, and all $u_1, \ldots, u_n \in \{t\sigma|_p \mid p \in Pos(t)\}$, $\Delta_{\mathrm{NF},t}$ contains the following rules:
        - $f([q_{u_1}, q_1], \ldots, [q_{u_n}, q_n]) \xrightarrow{c \wedge c'} [q_{f(u_1,\ldots,u_n)}, q]$ if $f(u_1, \ldots, u_n) = t\sigma_0$ and c' is constructed as sketched below.
        - $f([q_{u_1}, q_1], \ldots, [q_{u_n}, q_n]) \xrightarrow{c} [q_{f(u_1,\ldots,u_n)}, q]$ if $[q_{f(u_1,\ldots,u_n)}, q] \in Q_{\mathrm{NF},t}$ and we are not in the first case.
        - $f([q_{u_1}, q_1], \ldots, [q_{u_n}, q_n]) \xrightarrow{c} [q_q, q]$ in all other cases

    c' is constructed as follows. From $f(u_1, \ldots, u_n)$ we can retrieve the rules applied at position p in t. Assume that the rule at p checks $\pi_1 \neq \pi_2$. This amounts to check $p\pi_1 \neq p\pi_2$ at the root position of t. Let $\mathcal{D}$ be all disequalities $p\pi_1 \neq p\pi_2$ obtained in this way. The non linearity of t implies some equalities: let $\mathcal{E}$ be the set of equalities $p_1 = p_2$, for all positions p_1, p_2 such that $t|_{p_1} = t|_{p_2}$ is a variable. Now, c' is the set of disequalities $\pi \neq \pi'$ which are not in $\mathcal{D}$ and that can be inferred from $\mathcal{D}, \mathcal{E}$ using the rules

    $$\begin{array}{l} pp_1 \neq p_2,\ p = p' \ \vdash\ p'p_1 \neq p_2 \\ p \neq p',\ pp_1 = p_2 \ \vdash\ p'p_1 \neq p_2 \end{array}$$

    For instance, let $t = f(x, f(x, y))$ and assume that the automaton $\mathcal{A}_{\mathrm{NF}}$ contains a rule $f(q, q) \xrightarrow{1 \neq 2} q$. Then the automaton $\mathcal{A}_{\mathrm{NF},t}$ will contain the rule $f([q_q, q], [q_{f(q,q)}, q]) \xrightarrow{1 \neq 2 \wedge 1 \neq 22} q$.

    The final states are $[q_u, q_f]$ where $q_f \in Q^f_{\mathrm{NF}}$ and u is an instance of t.

    Prove that $\mathcal{A}_{\mathrm{NF},t}$ accepts at least one term if and only if t is not ground reducible by $\mathcal{R}$.

Exercise 4.11. Prove Theorem 4.4.13 along the lines of the proof of Theorem 3.4.5.

Exercise 4.12. Show that the algorithm for deciding emptiness of deterministic complete flat tree automaton works for non-deterministic flat tree automata such that for each state q the number of non-equivalent terms reaching q is 0 or greater than or equal to 2.

4.7 Bibliographic notes

RATEG appeared in Mongy's thesis [Mon81]. Unfortunately, as shown in [Mon81] the emptiness problem is undecidable for the class RATEG (and hence for AWEDC). The undecidability can be even shown for a more restricted class of *automata with equality tests between cousins* (see [Tom92]).

The remarkable subclass AWCBB is defined in [BT92]. This paper presents the results cited in Section 4.3, especially Theorem 4.3.5.
Concerning complexity, the result used in Section 4.3.3 (EXPTIME-completeness of the emptiness of the intersection of n recognizable tree languages) may be found in [FSVY91, Sei94b].
[DCC95] is concerned with reduction automata and their use as a tool for the decision of the encompassment theory in the general case.
The first decidability proof for ground reducibility is due to [Pla85]. In [CJ97a], ground reducibility decision is shown EXPTIME-complete. In this work, an EXPTIME algorithm for emptiness decision for AWEDC with only disequality constrained The result mentioned in Section 4.5.
The class of generalized reduction automata is introduced in [CCC$^+$94]. In this paper, a efficient cleaning algorithm is given for emptiness decision.

Chapter 5

Tree Set Automata

This chapter introduces a class of automata for sets of terms called *Generalized Tree Set Automata*. Languages associated with such automata are sets of sets of terms. The class of languages recognized by *Generalized Tree Set Automata* fulfills properties that suffices to build automata-based procedures for solving problems involving sets of terms, for instance, for solving systems of set constraints.

5.1 Introduction

"The notion of type expresses the fact that one just cannot apply any operator to any value. Inferring and checking a program's type is then a proof of partial correction" quoting Marie-Claude Gaudel. *"The main problem in this field is to be flexible while remaining rigorous, that is to allow polymorphism (a value can have more than one type) in order to avoid repetitions and write very general programs while preserving decidability of their correction with respect to types."*

On that score, the set constraints formalism is a compromise between power of expression and decidability. This has been the object of active research for a few years.

Set constraints are relations between sets of terms. For instance, let us define the natural numbers with 0 and the successor relation denoted by s. Thus, the constraint

$$\mathsf{Nat} = 0 \cup s(\mathsf{Nat}) \tag{5.1}$$

corresponds to this definition. Let us consider the following system:

$$\begin{array}{rcl} \mathsf{Nat} & = & 0 \cup s(\mathsf{Nat}) \\ \mathsf{List} & = & cons(\mathsf{Nat}, \mathsf{List}) \cup \mathsf{nil} \\ \mathsf{List}_+ & \subseteq & \mathsf{List} \\ \mathsf{car}(\mathsf{List}_+) & \subseteq & s(\mathsf{Nat}) \end{array} \tag{5.2}$$

The first constraint defines natural numbers. The second constraint codes the set of LISP-like lists of natural numbers. The empty list is nil and other lists are obtained using the constructor symbol cons. The last two constraints represent the set of lists with a non zero first element. Symbol car has the usual interpretation: the head of a list. Here $\mathsf{car}(\mathsf{List}_+)$ can be interpreted as the set of all terms at first position in List_+, that is all terms t such that there exists u with $\mathsf{cons}(t, u) \in \mathsf{List}_+$. In the set constraint framework such an operator car is often written cons_1^{-1}.

Set constraints are the essence of Set Based Analysis. The basic idea is to reason about program variables as sets of possible values. Set Based Analysis involves first writing set constraints expressing relationships between sets of program values, and then solving the system of set constraints. A single approximation is: all dependencies between the values of program variables are ignored. Techniques developed for Set Based Analysis have been successfully applied in program analysis and type inference and the technique can be combined with others [HJ92].

Set constraints have also been used to define a constraint logic programming language over sets of ground terms that generalizes ordinary logic programming over an Herbrand domain [Koz98].

In a more general way, a *system of set constraints* is a conjunction of *positive* constraints of the form $exp \subseteq exp'$[1] and *negative* constraints of the form $exp \not\subseteq exp'$. Right hand side and left hand side of these inequalities are *set expressions*, which are built with

- function symbols: in our example 0, s, cons, nil are function symbols.
- operators: union $\cup$, intersection $\cap$, complement $\sim$
- projection symbols: for instance, in the last equation of system (5.2) car denotes the first component of cons. In the set constraints syntax, this is written $\mathsf{cons}_{(1)}^{-1}$.
- set variables like Nat or List.

An interpretation assigns to each set variable a set of terms only built with function symbols. A solution is an interpretation which satisfies the system. For example, $\{0, s(0), s(s(0)), \dots\}$ is a solution of Equation (5.1).

In the set constraint formalism, set inclusion and set union express in a natural way parametric polymorphism: $\mathsf{List} \subseteq \mathsf{nil} \cup \mathsf{cons}(X, \mathsf{List})$.

In logic or functional programming, one often use dynamic procedures to deal with type. In other words, a run-time procedure checks whether or not an expression is well-typed. This permits maximum programming flexibility at the potential cost of efficiency and security. Static analysis partially avoids these drawbacks with the help of type inference and type checking procedures. The information extracted at compile time is also used for optimization.

Basically, program sources are analyzed at compile time and an ad hoc formalism is used to represent the result of the analysis. For types considered as sets of values, the set constraints formalism is well suited to represent them and to express their relations. Numerous inference and type checking algorithms in logic, functional and imperative programming are based on a resolution procedure for set constraints.

---

[1] $exp = exp'$ for $exp \subseteq exp' \wedge exp' \subseteq exp$.

Most of the earliest algorithms consider systems of set constraints with weak power of expression. More often than not, these set constraints always have a least solution — w.r.t. inclusion — which corresponds to a (tuple of) regular set of terms. In this case, types are usual sorts. A sort signature defines a tree automaton (see Section 3.4.1 for the correspondence between automata and sorts). For instance, regular equations iontroduced in Section 2.3 such a subclass of set constraints. Therefore, these methods are closely related finite tree automata and use classical algorithms on these recognizers, like the ones presented in Chapter 1.

In order to obtain a more precise information with set constraints in static analysis, one way is to enrich the set constraints vocabulary. In one hand, with a large vocabulary an analysis can be accurate and relevant, but on the other hand, solutions are difficult to obtain.

Nonetheless, an essential property must be preserved: the decidability of satisfiability. There must exists a procedure which determines whether or not a system of set constraints has solutions. In other words, extracted information must be sufficient to say whether the objects of an analyzed program have a type. It is crucial, therefore, to know which classes of set constraints are decidable, and identifying the complexity of set constraints is of paramount importance.

A second important characteristic to preserve is to represent solutions in a convenient way. We want to obtain a kind of solved form from which one can decide whether a system has solutions and one can "compute" them.

In this chapter, we present an automata-based algorithm for solving systems of positive and negative set constraints where no projection symbols occurs. We define a new class of automata recognizing sets of (codes of) n-tuples of tree languages. Given a system of set constraints, there exists an automaton of this class which recognizes the set of solutions of the system. Therefore properties of our class of automata directly translate to set constraints.

In order to introduce our automata, we discuss the case of unary symbols, i.e. the case of strings over finite alphabet. For instance, let us consider the following constraints over the alphabet composed of two unary symbols a and b and a constant 0:

$$\begin{aligned} Xaa \cup Xbb &\subseteq X \qquad (5.3)\\ Y &\subseteq X \end{aligned}$$

This system of set constraints can be encoded in a formula of the monadic second order theory of 2 successors named a and b:

$$\begin{aligned} &\forall u\ (u \in X \Rightarrow (uaa \in X \wedge ubb \in X))\wedge \\ &\forall u\ u \in Y \Rightarrow u \in X \end{aligned}$$

We have depicted in Fig 5.1 (a beginning of) an infinite tree which is a model of the formula. Each node corresponds to a string over a and b. The root is associated with the empty string; going down to the left concatenates a a; going down to the right concatenates a b. Each node of the tree is labelled with a couple of points. The two components correspond to sets X and Y. A black point in the first component means that the current node belongs to X. Conversely, a white point in the first component means that the current node does not belong to X. Here we have $X = \{\varepsilon, aa, bb, \dots\}$ and $Y = \{\varepsilon, bb, \dots\}$.

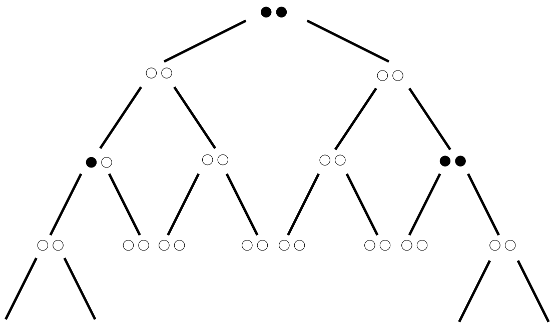


Figure 5.1: An infinite tree for the representation of a couple of word languages (X, Y). Each node is associated with a word. A black dot stands for belongs to. $X = \{\varepsilon, aa, bb, \dots\}$ and $Y = \{\varepsilon, bb, \dots\}$.

A tree language that encodes solutions of Eq. 5.3 is Rabin-recognizable by a tree automaton which must avoid the three forbidden patterns depicted in Figure 5.2.

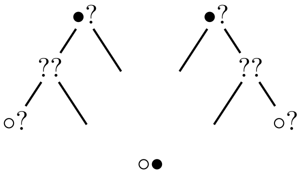


Figure 5.2: The set of three forbidden patterns. '?' stands for black or white dot. The tree depicted in Fig. 5.1 exclude these three patterns.

Given a ranked alphabet of unary symbols and one constant and a system of set constraints over $\{X_1, \dots, X_n\}$, one can encode a solution with a $\{0,1\}^n$-valued infinite tree and the set of solutions is recognized by an infinite tree automaton. Therefore, decidability of satisfiability of systems of set constraints can easily be derived from Rabin's Tree Theorem [Rab69] because infinite tree automata can be considered as an acceptor model for n-tuples of word languages over finite alphabet[2].

We extend this method to set constraints with symbols of arbitrary arity. Therefore, we define an acceptor model for mappings from $T(\mathcal{F})$, where $\mathcal{F}$ is a ranked alphabet, into a set $E = \{0,1\}^n$ of labels. Our automata can be viewed as an extension of infinite tree automata, but we will use weaker acceptance condition. The acceptance condition is: the range of a successful run is in a specified set of accepting set of states. We will prove that we can design an automaton which recognizes the set of solutions of a system of both positive and negative set constraints. For instance, let us consider the following system:

---

[2] The entire class of Rabin's tree languages is not captured by solutions of set of words constraints. Set of words constraints define a class of languages which is strictly smaller than Büchi recognizable tree languages.

$$Y \not\subseteq \bot \tag{5.4}$$

$$X \subseteq f(Y, \sim X) \cup a \tag{5.5}$$

where $\bot$ stands for the empty set and $\sim$ stands for the complement symbol.

The underlying structure is different than in the previous example since it is now the whole set of terms on the alphabet composed of a binary symbol f and a constant a. Having a representation of this structure in mind is not trivial. One can imagine a directed graph whose vertices are terms and such that there exists an edge between each couple of terms in the direct subterm relation (see figure 5.3).

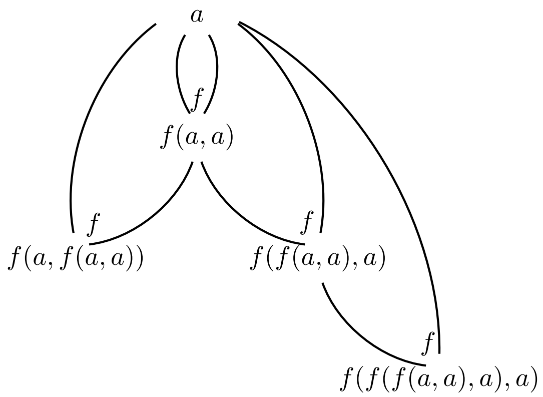


Figure 5.3: The (beginning of the) underlying structure for a two letter alphabet $\{f(,), a\}$.

An automaton have to associate a state with each node following a finite set of rules. In the case of the example above, states are also couples of $\bullet$ or $\circ$.

Each vertex is of infinite out-degree, nonetheless one can define as in the word case forbidden patterns for incoming vertices which such an automaton have to avoid in order to satisfy Eq. (5.5) (see Fig. 5.4, Pattern ? stands for $\circ$ or $\bullet$). The acceptance condition is illustrated using Eq. (5.4). Indeed, to describe a solution of the system of set constraints, the pattern ?$\bullet$ must occur somewhere in a successful "run" of the automaton.

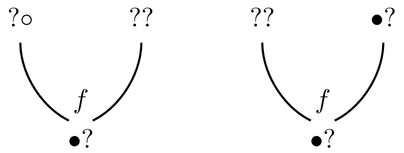


Figure 5.4: Forbidden patterns for (5.5).

Consequently, decidability of systems of set constraints is a consequence of decidability of emptiness in our class of automata. Emptiness decidability is easy for automata without acceptance conditions (it corresponds to the case of positive set constraints only). The proof is more difficult and technical in the general case and is not presented here. Moreover, and this is the main advantage of an automaton-based method, properties of recognizable sets directly translate to sets of solutions of systems of set constraints. Therefore, we are able to prove nice properties. For instance, we can prove that a non empty set of solutions always contain a regular solution. Moreover we can prove the decidability of existence of finite solutions.

5.2 Definitions and Examples

Infinite tree automata are an acceptor model for infinite trees, i.e. for mappings from A^* into E where A is a finite alphabet and E is a finite set of labels. We define and study $\mathcal{F}$-generalized tree set automata which are an acceptor model for mappings from $T(\mathcal{F})$ into E where $\mathcal{F}$ is a finite ranked alphabet and E is a finite set of labels.

5.2.1 Generalized Tree Sets

Let $\mathcal{F}$ be a ranked alphabet and E be a finite set. An ***E-valued $\mathcal{F}$-generalized tree set*** g is a mapping from $T(\mathcal{F})$ into E. We denote by $\mathcal{G}_E$ the set of E-valued $\mathcal{F}$-generalized tree sets.

For the sake of brevity, we do not mention the signature $\mathcal{F}$ which strictly speaking is in order in generalized tree sets. We also use the abbreviation GTS for generalized tree sets.

Throughout the chapter, if $c \in \{0,1\}^n$, then c_i denotes the i^{th} component of the tuple c. If we consider the set $E = \{0,1\}^n$ for some n, a generalized tree set g in $\mathcal{G}_{\{0,1\}^n}$ can be considered as a n-tuple $(L_1, \ldots, L_n)$ of tree languages over the ranked alphabet $\mathcal{F}$ where $L_i = \{t \in T(\mathcal{F}) \mid g(t)_i = 1\}$.

We will need in the chapter the following operations on generalized tree sets. Let g (resp. g') be a generalized tree set in $\mathcal{G}_E$ (resp. $\mathcal{G}_{E'}$). The generalized tree set $g \uparrow g' \in \mathcal{G}_{E \times E'}$ is defined by $g \uparrow g'(t) = (g(t), g'(t))$, for each term t in $T(\mathcal{F})$. Conversely let g be a generalized tree set in $\mathcal{G}_{E \times E'}$ and consider the projection π from $E \times E'$ into the E-component then $\pi(g)$ is the generalized tree set in $\mathcal{G}_E$ defined by $\pi(g)(t) = \pi(g(t))$. Let $G \subseteq \mathcal{G}_{E \times E'}$ and $G' \subseteq \mathcal{G}_E$, then $\pi(G) = \{\pi(g) \mid g \in G\}$ and $\pi^{-1}(G') = \{g \in \mathcal{G}_{E \times E'} \mid \pi(g) \in G'\}$.

5.2.2 Tree Set Automata

A ***generalized tree set automaton*** $\mathcal{A} = (Q, \Delta, \Omega)$ (GTSA) over a finite set E consist of a finite state set Q, a transition relation $\Delta \subseteq \bigcup_p Q^p \times \mathcal{F}_p \times E \times Q$ and a set $\Omega \subseteq 2^Q$ of accepting sets of states.

A ***run*** of $\mathcal{A}$ (or $\mathcal{A}$-run) on a generalized tree set $g \in \mathcal{G}_E$ is a mapping $r : T(\mathcal{F}) \to Q$ with:

$$(r(t_1), \ldots, r(t_p), f, g(f(t_1, \ldots, t_p)), r(f(t_1, \ldots, t_p))) \in \Delta$$

for $t_1, \ldots, t_p \in T(\mathcal{F})$ and $f \in \mathcal{F}_p$. The run r is ***successful*** if the range of r is in Ω i.e. $r(T(\mathcal{F})) \in \Omega$.

A generalized tree set $g \in \mathcal{G}_E$ is ***accepted*** by the automaton $\mathcal{A}$ if some run r of $\mathcal{A}$ on g is successful. We denote by $\mathcal{L}(\mathcal{A})$ the set of E-valued generalized tree sets accepted by a generalized tree set automaton $\mathcal{A}$ over E. A set $G \subseteq \mathcal{G}_E$ is recognizable if $G = \mathcal{L}(\mathcal{A})$ for some generalized tree set automaton $\mathcal{A}$.

In the following, a rule $(q_1, \dots, q_p, f, l, q)$ is also denoted by $f(q_1, \dots, q_p) \xrightarrow{l} q$. Consider a term $t = f(t_1, \dots, t_p)$ and a rule $f(q_1, \dots, q_p) \xrightarrow{l} q$, this rule can be applied in a run r on a generalized tree set g for the term t if $r(t_1) = q_1$,…,$r(t_p) = q_p$, t is labeled by l, i.e. $g(t) = l$. If the rule is applied, then $r(t) = q$.

A generalized tree set automaton $\mathcal{A} = (Q, \Delta, \Omega)$ over E is

- ***deterministic*** if for each tuple $(q_1, \dots, q_p, f, l) \in Q^p \times \mathcal{F}_p \times E$ there is at most one state $q \in Q$ such that $(q_1, \dots, q_p, f, l, q) \in \Delta$.
- ***strongly deterministic*** if for each tuple $(q_1, \dots, q_p, f) \in Q^p \times \mathcal{F}_p$ there is at most one pair $(l, q) \in E \times Q$ such that $(q_1, \dots, q_p, f, l, q) \in \Delta$.
- ***complete*** if for each tuple $(q_1, \dots, q_p, f, l) \in Q^p \times \mathcal{F}_p \times E$ there is at least one state $q \in Q$ such that $(q_1, \dots, q_p, f, l, q) \in \Delta$.
- ***simple*** if Ω is "subset-closed", that is $\omega \in \Omega \Rightarrow (\forall \omega' \subseteq \omega\ \omega' \in \Omega)$.

Successfulness for simple automata just implies some states are *not* assumed along a run. For instance, if the accepting set of a GTSA $\mathcal{A}$ is $\Omega = 2^Q$ then $\mathcal{A}$ is simple and any run is successful. But, if $\Omega = \{Q\}$, then $\mathcal{A}$ is not simple and each state must be assumed at least once in a successful run. The definition of simple automata will be clearer with the relationships with set constraints and the emptiness property (see Section 5.4). Briefly, positive set constraints are related to simple GTSA for which the proof of emptiness decision is straightforward. Another and equivalent definition for simple GTSA relies on the acceptance condition: a run r is successful if and only if $r(T(\mathcal{F})) \subseteq \omega \in \Omega$.

There is in general an *infinite* number of runs — and hence an *infinite* number of GTS recognized — even in the case of deterministic generalized tree set automata (see example 5.2.1.2). Nonetheless, given a GTS g, there is at most one run on g for a deterministic generalized tree set automata. But, in the case of *strongly* deterministic generalized tree set automata, there is at most one run (see example 5.2.1.1) and therefore there is at most one GTS recognized.

Example 5.2.1.

Ex. 5.2.1.1 Let $E = \{0, 1\}$, $\mathcal{F} = \{\mathsf{cons}(,), s(), \mathsf{nil}, 0\}$. Let $\mathcal{A} = (Q, \Delta, \Omega)$ be defined by $Q = \{\mathsf{Nat}, \mathsf{List}, \mathsf{Term}\}$, $\Omega = 2^Q$, and Δ is the following set of rules:

$$0 \xrightarrow{0} \mathsf{Nat}\ ;\ s(\mathsf{Nat}) \xrightarrow{0} \mathsf{Nat}\ ;\quad nil \xrightarrow{1} \mathsf{List} \quad ;$$
$$\mathsf{cons}(\mathsf{Nat}, \mathsf{List}) \xrightarrow{1} \mathsf{List}\ ;$$
$$\mathsf{cons}(q, q') \xrightarrow{0} \mathsf{Term} \quad \forall (q, q') \neq (\mathsf{Nat}, \mathsf{List})\ ;$$
$$s(q) \xrightarrow{0} \mathsf{Term} \quad \forall q \neq \mathsf{Nat}\ .$$

$\mathcal{A}$ is strongly deterministic, simple, and not complete. $\mathcal{L}(\mathcal{A})$ is a singleton set. Indeed, there is a unique run r on a unique generalized tree set $g \in \mathcal{G}_{\{0,1\}^n}$. The run r maps every natural number on state Nat, every list on state List and the other terms on state Term. Therefore g maps a natural number on 0, a list on 1 and the other terms on 0. Hence, we say that $\mathcal{L}(\mathcal{A})$ is the regular tree language L of Lisp-like lists of natural numbers.

Ex. 5.2.1.2 Let $E = \{0, 1\}$, $\mathcal{F} = \{\mathsf{cons}(,), s(), \mathsf{nil}, 0\}$, and let $\mathcal{A}' = (Q', \Delta', \Omega')$ be defined by $Q' = Q$, $\Omega' = \Omega$, and

$$\Delta' = \Delta \cup \{\mathsf{cons}(\mathsf{Nat}, \mathsf{List}) \xrightarrow{0} \mathsf{List}, \mathsf{nil} \xrightarrow{0} \mathsf{List}\}.$$

$\mathcal{A}'$ is deterministic (but not strongly), simple, and not complete, and $\mathcal{L}(\mathcal{A}')$ is the set of all subsets of the regular tree language L of Lisp-like lists of natural numbers. Indeed, successful runs can now be defined on generalized tree sets g such that a term in L is labeled by 0 or 1.

Ex. 5.2.1.3 Let $E = \{0, 1\}^2$, $\mathcal{F} = \{\mathsf{cons}(,), s(), \mathsf{nil}, 0\}$, and let $\mathcal{A} = (Q, \Delta, \Omega)$ be defined by $Q = \{\mathsf{Nat}, \mathsf{Nat}', \mathsf{List}, \mathsf{Term}\}$, $\Omega = 2^Q$, and Δ is the following set of rules:

$$\begin{array}{rcccl}
0 \xrightarrow{(0,0)} \mathsf{Nat} & ; & 0 \xrightarrow{(1,0)} \mathsf{Nat}' & ; & s(\mathsf{Nat}) \xrightarrow{(0,0)} \mathsf{Nat} \\
s(\mathsf{Nat}) \xrightarrow{(1,0)} \mathsf{Nat}' & ; & s(\mathsf{Nat}') \xrightarrow{(0,0)} \mathsf{Nat} & ; & s(\mathsf{Nat}') \xrightarrow{(1,0)} \mathsf{Nat}' \\
\mathsf{nil} \xrightarrow{(0,1)} \mathsf{List} & ; & \mathsf{cons}(\mathsf{Nat}', \mathsf{List}) \xrightarrow{(0,1)} \mathsf{List} & ; & \\
\multicolumn{5}{c}{s(q) \xrightarrow{(0,0)} \mathsf{Term} \quad \forall q \neq \mathsf{Nat}} \\
\multicolumn{5}{c}{\mathsf{cons}(q, q') \xrightarrow{(0,0)} \mathsf{Term} \quad \forall (q, q') \neq (\mathsf{Nat}', \mathsf{List})}
\end{array}$$

$\mathcal{A}$ is deterministic, simple, and not complete, and $\mathcal{L}(\mathcal{A})$ is the set of 2-tuples of tree languages (N', L') where N' is a subset of the regular tree language of natural numbers and L' is the set of Lisp-like lists of natural numbers over N'.

Let us remark that the set N' may be non-regular. For instance, one can define a run on a characteristic generalized tree set g_p of Lisp-like lists of prime numbers. The generalized tree set g_p is such that $g_p(t) = (1, 0)$ when t is a (code of a) prime number.

In the previous examples, we only consider simple generalized tree set automata. Moreover all runs are successful runs. The following examples are non-simple generalized tree set automata in order to make clear the interest of acceptance conditions. For this, compare the sets of generalized tree sets obtained in examples 5.2.1.3 and 5.2.2 and note that with acceptance conditions, we can express that a set is non empty.

Example 5.2.2. *Example 5.2.1.3 continued*

Let $E = \{0, 1\}^2$, $\mathcal{F} = \{\mathsf{cons}(,), \mathsf{nil}, s(), 0\}$, and let $\mathcal{A}' = (Q', \Delta', \Omega')$ be defined by $Q' = Q$, $\Delta' = \Delta$, and $\Omega' = \{\omega \in 2^Q \mid \mathsf{Nat}' \in \omega\}$. $\mathcal{A}'$ is deterministic, not simple, and not complete, and $\mathcal{L}(\mathcal{A}')$ is the set of 2-tuples of tree languages (N', L') where N' is a subset of the regular tree language of natural numbers and L' is the set of Lisp-like lists of natural numbers over N', and $N' \neq \emptyset$. Indeed, for a successful r on g, there must be a term t such that $r(t) = \mathsf{Nat}'$ therefore, there must be a term t labelled by $(1, 0)$, henceforth $N' \neq \emptyset$.

5.2.3 Hierarchy of GTSA-recognizable Languages

Let us define:

- $\mathcal{R}_{\text{GTS}}$, the class of languages recognizable by GTSA,
- $\mathcal{R}_{\text{DGTS}}$, the class of languages recognizable by deterministic GTSA,
- $\mathcal{R}_{\text{SGTS}}$, the class of languages recognizable by Simple GTSA.

The three classes defined above are proved to be different. They are also closely related to classes of languages defined from the set constraint theory point of view.

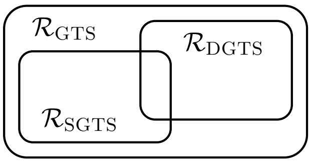

$\mathcal{R}_{\text{GTS}}$
$\mathcal{R}_{\text{DGTS}}$
$\mathcal{R}_{\text{SGTS}}$

Figure 5.5: Classes of GTSA-recognizable languages

Classes of GTSA-recognizable languages have also different closure properties. We will prove in Section 5.3.1 that $\mathcal{R}_{\text{SGTS}}$ and the entire class $\mathcal{R}_{\text{GTS}}$ are closed under union, intersection, projection and cylindrification; $\mathcal{R}_{\text{DGTS}}$ is closed under complementation and intersection.

We propose three examples that illustrate the differences between the three classes. First, $\mathcal{R}_{\text{DGTS}}$ is not a subset of $\mathcal{R}_{\text{SGTS}}$.

Example 5.2.3. Let $E = \{0, 1\}$, $\mathcal{F} = \{f, a\}$ where a is a constant and f is unary. Let us consider the deterministic but non-simple GTSA $\mathcal{A}_1 = (\{q_0, q_1\}, \Delta_1, \Omega_1)$ where Δ_1 is:

$$\begin{array}{ll} a \xrightarrow{0} q_0, & a \xrightarrow{1} q_1, \\ f(q_0) \xrightarrow{0} q_0, & f(q_1) \xrightarrow{0} q_0, \\ f(q_0) \xrightarrow{1} q_1, & f(q_1) \xrightarrow{1} q_0. \end{array}$$

and $\Omega_1 = \{\{q_0, q_1\}, \{q_1\}\}$. Let us prove that

$$\mathcal{L}(\mathcal{A}_1) = \{L \mid L \neq \emptyset\}$$

is not in $\mathcal{R}_{\text{SGTS}}$.

Assume that there exists a simple GTSA $\mathcal{A}_s$ with n states such that $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_s)$. Hence, $\mathcal{A}_s$ recognizes also each one of the singleton sets $\{f^i(a)\}$ for $i > 0$. Let us consider some i greater than $n + 1$, we can deduce that a run r on the GTS g associated with $\{f^i(a)\}$ maps two terms $f^k(a)$ and $f^l(a)$, $k < l < i$ to the same state. We have $g(t) = 0$ for every term $t \trianglelefteq f^l(a)$ and r "loops" between $f^k(a)$ and $f^l(a)$. Therefore, one can build another run r_0 on a GTS g_0 such that $g_0(t) = 0$ for each $t \in T(\mathcal{F})$. Since $\mathcal{A}_s$ is simple, and since the range of r_0 is a subset of the range of r, g_0 is recognized, hence the empty set is recognized which contradicts the hypothesis.

Basically, using simple GTSA it is not possible to enforce a state to be assumed somewhere by every run. Consequently, it is not possible to express global properties of generalized tree languages such as non-emptiness.

Second, $\mathcal{R}_{\text{SGTS}}$ is not a subset of $\mathcal{R}_{\text{DGTS}}$.

Example 5.2.4. Let us consider the non-deterministic but simple GTSA $\mathcal{A}_2 = (\{q_f, q_h\}, \Delta_2, \Omega_2)$ where Δ_2 is:

$$\begin{array}{ll} a \xrightarrow{0} q_f \mid q_h, & a \xrightarrow{1} q_f \mid q_h, \\ f(q_f) \xrightarrow{1} q_f \mid q_h, & h(q_h) \xrightarrow{1} q_f \mid q_h, \\ f(q_h) \xrightarrow{0} q_f \mid q_h, & h(q_f) \xrightarrow{0} q_f \mid q_h, \end{array}$$

and $\Omega_2 = 2^{\{q_f, q_h\}}$. It is easy to prove that $\mathcal{L}(\mathcal{A}_2) = \{L \mid \forall t\ f(t) \in L \Leftrightarrow h(t) \notin L\}$. The proof that no deterministic GTSA recognizes $\mathcal{L}(\mathcal{A}_2)$ is left to the reader.

We terminate with an example of a non-deterministic and non-simple generalized tree set automaton. This example will be used in the proof of Proposition 5.3.2.

Example 5.2.5. Let $\mathcal{A} = (Q, \Delta, \Omega)$ be defined by $Q = \{q, q'\}$, $\Omega = \{Q\}$, and Δ is the following set of rules:

$$\begin{array}{llll} a \xrightarrow{1} q & ; a \xrightarrow{1} q' & ; a \xrightarrow{0} q' & ; f(q) \xrightarrow{1} q\ ; \\ f(q') \xrightarrow{0} q' & ; f(q') \xrightarrow{1} q' & ; f(q') \xrightarrow{1} q\ ; & \end{array}$$

The proof that $\mathcal{A}$ is not deterministic, not simple, and not complete, and $\mathcal{L}(\mathcal{A}) = \{L \subseteq T(\mathcal{F}) \mid \exists t \in T(\mathcal{F})\ ((t \in L) \wedge (\forall t' \in T(\mathcal{F})\ (t \trianglelefteq t') \Rightarrow (t' \in L)))\}$ is left as an exercise to the reader.

5.2.4 Regular Generalized Tree Sets, Regular Runs

As we mentioned it in Example 5.2.1.3, the set recognized by a GTSA may contain GTS corresponding to non-regular languages. But regularity is of major interest for practical reasons because it implies a GTS or a language to be finitely defined.

A generalized tree set $g \in \mathcal{G}_E$ is ***regular*** if there exist a finite set R, a mapping $\alpha : T(\mathcal{F}) \to R$, and a mapping $\beta : R \to E$ satisfying the following two properties.

1. $g = \alpha\beta$ (i.e. $g = \beta \circ \alpha$),
2. α is closed under contexts, i.e. for all context c and terms t_1, t_2, we have

$$(\alpha(t_1) = \alpha(t_2)) \Rightarrow (\alpha(c[t_1]) = \alpha(c[t_2]))$$

In the case $E = \{0,1\}^n$, regular generalized tree sets correspond to n-tuples of regular tree languages.

Although the definition of regularity could lead to the definition of regular run — because a run can be considered as a generalized tree set in $\mathcal{G}_Q$, we use stronger conditions for a run to be regular. Indeed, if we define regular runs as regular generalized tree sets in $\mathcal{G}_Q$, regularity of generalized tree sets and regularity of runs do not correspond in general. For instance, one could define regular runs on non-regular generalized tree sets in the case of non-strongly deterministic generalized tree set automata, and one could define non-regular runs on regular generalized tree sets in the case of non-deterministic generalized tree set automata. Therefore, we only consider regular runs on regular generalized tree sets:

> *A* ***run*** r ***on a generalized tree set*** g ***is regular*** *if* $r \uparrow g \in \mathcal{G}_{E\times Q}$ *is regular. Consequently,* r *and* g *are regular generalized tree sets.*

Proposition 5.2.6. *Let $\mathcal{A}$ be a generalized tree set automaton, if g is a regular generalized tree set in $\mathcal{L}(\mathcal{A})$ then there exists a regular $\mathcal{A}$-run on g.*

Proof. Consider a generalized tree set automaton $\mathcal{A} = (Q, \Delta, \Omega)$ over E and a regular generalized tree set g in $\mathcal{L}(\mathcal{A})$ and let r be a successful run on g. Let L be a finite tree language closed under the subterm relation and such that $\mathcal{F}_0 \subseteq L$ and $r(L) = r(T(\mathcal{F}))$. The generalized tree set g is regular, therefore there exist a finite set R, a mapping $\alpha : T(\mathcal{F}) \rightarrow R$ closed under context and a mapping $\beta : R \rightarrow E$ such that $g = \alpha\beta$. We now define a regular run r' on g.

Let $L_\star = L \cup \{\star\}$ where $\star$ is a new constant symbol and let ϕ be the mapping from $T(\mathcal{F})$ into $Q \times R \times L_\star$ defined by $\phi(t) = (r(t), \alpha(t), u)$ where $u = t$ if $t \in L$ and $u = \star$ otherwise. Hence $R' = \phi(T(\mathcal{F}))$ is a finite set because $R' \subseteq Q \times R \times L_\star$. For each ρ in R', let us fix $t_\rho \in T(\mathcal{F})$ such that $\phi(t_\rho) = \rho$.

The run r' is now (regularly) defined via two mappings α' and β'. Let β' be the projection from $Q \times R \times L_\star$ into Q and let $\alpha' : T(\mathcal{F}) \rightarrow R'$ be inductively defined by:

$$\forall a \in \mathcal{F}_0 \ \alpha'(a) = \phi(a);$$

and

$$\forall f \in \mathcal{F}_p \forall t_1, \dots, t_p \in T(\mathcal{F})$$

$$\alpha'(f(t_1, \dots, t_p)) = \phi(f(t_{\alpha'(t_1)}, \dots, t_{\alpha'(t_p)})).$$

Let $r' = \alpha'\beta'$. First we can easily prove by induction that $\forall t \in L \ \alpha'(t) = \phi(t)$ and deduce that $\forall t \in L \ r'(t) = r(t)$. Thus r' and r coincide on L. It remains to prove that (1) the mapping α' is closed under context, (2) r' is a run on g and (3) r' is a successful run.

(1) From the definition of α' we can easily derive that the mapping α' is closed under context.

(2) We prove that the mapping $r' = \alpha'\beta'$ is a run on g, that is if $t = f(t_1, \dots, t_p)$ then $(r'(t_1), \dots, r'(t_p), f, g(t), r'(t)) \in \Delta$.

Let us consider a term $t = f(t_1, \dots, t_p)$. From the definitions of α', β', and r', we get $r'(t) = r(t')$ with $t' = f(t_{\alpha'(t_1)}, \dots, t_{\alpha'(t_p)})$. The mapping r is a run on g, hence $(r(t_{\alpha'(t_1)}), \dots, r(t_{\alpha'(t_p)}), f, g(t'), r(t')) \in \Delta$, and thus it suffices to prove that $g(t) = g(t')$ and, for all i, $r'(t_i) = r(t_{\alpha'(t_i)})$.

Let $i \in \{1, \dots, p\}$, $r'(t_i) = \beta'(\alpha'(t_i))$ by definition of r'. By definition of $t_{\alpha'(t_i)}$, $\alpha'(t_i) = \phi(t_{\alpha'(t_i)})$, therefore $r'(t_i) = \beta'(\phi(t_{\alpha'(t_i)}))$. Now, using the definitions of ϕ and β', we get $r'(t_i) = r(t_{\alpha'(t_i)})$.

In order to prove that $g(t) = g(t')$, we prove that $\alpha(t) = \alpha(t')$. Let π be the projection from R' into R. We have $\alpha(t') = \pi(\phi(t'))$ by definition of ϕ and π. We have $\alpha(t') = \pi(\alpha'(t))$ using definitions of t' and α'. Now $\alpha(t') = \pi(\phi(t_{\alpha'(t)}))$ because $\phi(t_{\alpha'(t)}) = \alpha'(t)$ by definition of $t_{\alpha'(t)}$. And then $\alpha(t') = \alpha(t_{\alpha'(t)})$ by definition of π and ϕ. Therefore it remains to prove that $\alpha(t_{\alpha'(t)}) = \alpha(t)$. The proof is by induction on the structure of terms.

If $t \in \mathcal{F}_0$ then $t_{\alpha'(t)} = t$, so the property holds (note that this property holds for all $t \in L$). Let us suppose that $t = f(t_1, \dots, t_p)$ and $\alpha(t_{\alpha'(t_i)}) = \alpha(t_i)$ $\forall i \in \{1, \dots, p\}$. First, using induction hypothesis and closure under context of α, we get

$$\alpha(f(t_1, \dots, t_p)) \quad = \quad \alpha(f(t_{\alpha'(t_1)}, \dots, t_{\alpha'(t_p)}))$$

Therefore,

$$\begin{aligned}
\alpha(f(t_1, \dots, t_p)) &= \alpha(f(t_{\alpha'(t_1)}, \dots, t_{\alpha'(t_p)})) \\
&= \pi(\phi(f(t_{\alpha'(t_1)}, \dots, t_{\alpha'(t_p)}))) \text{ (def. of } \phi \text{ and } \pi) \\
&= \pi(\alpha'(f(t_1, \dots, t_p))) \text{ (def. of } \alpha') \\
&= \pi(\phi(t_{\alpha'(f(t_1,\dots,t_p))})) \text{ (def. of } t_{\alpha'(f(t_1,\dots,t_p))}) \\
&= \alpha(t_{\alpha'(f(t_1,\dots,t_p))}) \text{ (def. of } \phi \text{ and } \pi).
\end{aligned}$$

(3) We have $r'(T(\mathcal{F})) = r'(L) = r(L) = r(T(\mathcal{F}))$ using the definition of r', the definition of L, and the equality $r'(L) = r(L)$. The run r is a successful run. Consequently r' is a successful run.

□

Proposition 5.2.7. *A non-empty recognizable set of generalized tree sets contains a regular generalized tree set.*

Proof. Let us consider a generalized tree set automaton $\mathcal{A}$ and a successful run r on a generalized tree set g. There exists a tree language closed under the subterm relation F such that $r(F) = r(T(\mathcal{F}))$. We define a regular run rr on a regular generalized tree set gg in the following way.

The run rr coincides with r on F: $\forall t \in F$, $rr(t) = r(t)$ and $gg(t) = g(t)$. The runs rr and gg are inductively defined on $T(\mathcal{F}) \setminus F$: given $q_1, \dots, q_p$ in $r(T(\mathcal{F}))$, let us fix a rule $f(q_1, \dots, q_p) \xrightarrow{l} q$ such that $q \in r(T(\mathcal{F}))$. The rule exists since r is a run. Therefore, $\forall t = f(t_1, \dots, t_p) \notin F$ such that $rr(t_i) = q_i$ for all $i \leq p$, we define $rr(t) = q$ and $gg(t) = l$, following the fixed rule $f(q_1, \dots, q_p) \xrightarrow{l} q$.

□

From the preceding, we can also deduce that a finite and recognizable set of generalized tree sets only contains regular generalized tree sets.

5.3 Closure and Decision Properties

5.3.1 Closure properties

This section is dedicated to the study of classical closure properties on GTSA-recognizable languages. For all positive results — union, intersection, projection, cylindrification — the proofs are constructive. We show that the class of recognizable sets of generalized tree sets is not closed under complementation and that non-determinism cannot be reduced for generalized tree set automata.

Set operations on sets of GTS have to be distinguished from set operations on sets of terms. In particular, in the case where $E = \{0,1\}^n$, if G_1 and G_2 are sets of GTS in $\mathcal{G}_E$, then $G_1 \cup G_2$ contains all GTS in G_1 and G_2. This is clearly different from the set of all $(L_1^1 \cup L_1^2, \dots, L_n^1 \cup L_n^2)$ where $(L_1^1, \dots, L_n^1)$ belongs to G_1 and $(L_1^2, \dots, L_n^2)$ belongs to G_2.

Proposition 5.3.1. *The class $\mathcal{R}_{\text{GTS}}$ is closed under intersection and union, i.e. if G_1, $G_2 \subseteq \mathcal{G}_E$ are recognizable, then $G_1 \cup G_2$ and $G_1 \cap G_2$ are recognizable.*

This proof is an easy modification of the classical proof of closure properties for tree automata, see Chapter 1.

Proof. Let $\mathcal{A}_1 = (Q_1, \Delta_1, \Omega_1)$ and $\mathcal{A}_2 = (Q_2, \Delta_2, \Omega_2)$ be two generalized tree set automata over E. Without loss of generality we assume that $Q_1 \cap Q_2 = \emptyset$.

Let $\mathcal{A} = (Q, \Delta, \Omega)$ with $Q = Q_1 \cup Q_2$, $\Delta = \Delta_1 \cup \Delta_2$, and $\Omega = \Omega_1 \cup \Omega_2$. It is immediate that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$.

We denote by π_1 and π_2 the projections from $Q_1 \times Q_2$ into respectively Q_1 and Q_2. Let $\mathcal{A}' = (Q', \Delta', \Omega')$ with $Q' = Q_1 \times Q_2$, Δ' is defined by

$$(f(q_1, \dots, q_p) \xrightarrow{l} q \in \Delta') \Leftrightarrow (\forall i \in \{1,2\}\ f(\pi_i(q_1), \dots, \pi_i(q_p)) \xrightarrow{l} \pi_i(q) \in \Delta_i)\ ,$$

where $q_1, \dots, q_p, q \in Q'$, $f \in \mathcal{F}_p$, $l \in E$, and Ω' is defined by

$$\Omega' = \{\omega \in 2^{Q'} \mid \pi_i(\omega) \in \Omega_i\ ,\ i \in \{1,2\}\}.$$

One can easily verify that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

□

Let us remark that the previous constructions also prove that the class $\mathcal{R}_{\text{SGTS}}$ is closed under union and intersection.

The class languages recognizable by deterministic generalized tree set automata is closed under complementation. But, this property is false in the general case of GTSA-recognizable languages.

Proposition 5.3.2. *(a) Let $\mathcal{A}$ be a generalized tree set automaton, there exists a complete generalized tree set automaton $\mathcal{A}_c$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_c)$.*

(b) If $\mathcal{A}_{cd}$ is a deterministic and complete generalized tree set automaton, there exists a generalized tree set automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \mathcal{G}_E - \mathcal{L}(\mathcal{A}_{cd})$.

(c) The class of GTSA-recognizable languages is not closed under complementation.

(d) Non-determinism can not be reduced for generalized tree set automata.

Proof. (a) Let $\mathcal{A} = (Q, \Delta, \Omega)$ be a generalized tree set automaton over E and let q' be a new state, i.e. $q' \notin Q$. Let $\mathcal{A}_c = (Q_c, \Delta_c, \Omega_c)$ be defined by $Q_c = Q \cup \{q'\}$, $\Omega_c = \Omega$, and

$$\begin{aligned} \Delta_c = \Delta \cup \{(q_1, \dots, q_p, f, l, q') \mid \quad & \{(q_1, \dots, q_p, f, l)\} \times Q \cap \Delta = \emptyset; \\ & q_1, \dots, q_p \in Q_c, f \in \mathcal{F}_p, l \in E\}. \end{aligned}$$

$\mathcal{A}_c$ is complete and $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_c)$. Note that $\mathcal{A}_c$ is simple if $\mathcal{A}$ is simple.

(b) $\mathcal{A}_{cd} = (Q, \Delta, \Omega)$ be a deterministic and complete generalized tree set automaton over E. The automaton $\mathcal{A}' = (Q', \Delta', \Omega')$ with $Q' = Q$, $\Delta' = \Delta$, and $\Omega' = 2^Q - \Omega$ recognizes the set $\mathcal{G}_E - \mathcal{L}(\mathcal{A}_{cd})$.

(c) $E = \{0, 1\}$, $\mathcal{F} = \{c, a\}$ where a is a constant and c is of arity 1. Let $G = \{g \in \mathcal{G}_{\{0,1\}^n} \mid \exists t \in T(\mathcal{F})\ ((g(t) = 1) \wedge (\forall t' \in T(\mathcal{F})\ (t \trianglelefteq t') \Rightarrow (g(t') = 1)))\}$. Clearly, G is recognizable by a non deterministic GTSA (see Example 5.2.5). Let $\overline{G} = \mathcal{G}_{\{0,1\}^n} - G$, we have $\overline{G} = \{g \in \mathcal{G}_{\{0,1\}^n} \mid \forall t \in T(\mathcal{F})\ \exists t' \in T(\mathcal{F})\ (t \trianglelefteq t') \wedge (g(t') = 0)\}$ and $\overline{G}$ is not recognizable. Let us suppose that $\overline{G}$ is recognized by an automaton $\mathcal{A} = (Q, \Delta, \Omega)$ with $\mathsf{Card}(Q) = k - 2$ and let us consider the generalized tree set g defined by: $g(c^i(a)) = 0$ if $i = k \times z$ for some integer z, and $g(c^i(a)) = 1$ otherwise. The generalized tree set g is in $\overline{G}$ and we consider a successful run r on g. We have $r(T(\mathcal{F})) = \omega \in \Omega$ therefore there exists some integer n such that $r(\{g(c^i(a)) \mid i \leq n\}) = \omega$. Moreover we can suppose that n is a multiple of k. As $\mathsf{Card}(Q) = k - 2$ there are two terms u and v in the set $\{c^i(a) \mid n{+}1 \leq i \leq n{+}k{-}1\}$ such that $r(u) = r(v)$. Note that by hypothesis, for all i such that $n{+}1 \leq i \leq n{+}k{+}1$, $g(c^i(a)) = 1$. Consequently, a successful run g' could be defined from g on the generalized tree set g' defined by $g'(t) = g(t)$ if $t = c^i(a)$ when $i \leq n$, and $g'(t) = 1$ otherwise. This leads to a contradiction because $g' \notin \overline{G}$.

(d) This result is a consequence of (b) and (c).

□

We will now prove the closure under projection and cylindrification. We will first prove a stronger lemma.

Lemma 5.3.3. *Let $G \subseteq \mathcal{G}_{E_1}$ be a GTSA-recognizable language and let $R \subseteq E_1 \times E_2$. The set $R(G) = \{g' \in \mathcal{G}_{E_2} \mid \exists g \in G\ \forall t \in T(\mathcal{F})\ (g(t), g'(t)) \in R\}$ is recognizable.*

Proof. Let $\mathcal{A} = (Q, \Delta, \Omega)$ such that $\mathcal{L}(\mathcal{A}) = G$. Let $\mathcal{A}' = (Q', \Delta', \Omega')$ where $Q' = Q$, $\Delta' = \{f(q_1, \dots, q_p) \xrightarrow{l'} q \mid \exists l \in E_1\ f(q_1, \dots, q_p) \xrightarrow{l} q \in \Delta$ and $(l, l') \in R\}$ and $\Omega' = \Omega$. We prove that $R(G) = \mathcal{L}(\mathcal{A}')$.

$\supseteq$ Let $g' \in \mathcal{L}(\mathcal{A}')$ and let r' be a successful run on g'. We construct a generalized tree set g such that for all $t \in T(\mathcal{F})$, $(g(t), g'(t)) \in R$ and such that r' is also a successful $\mathcal{A}$-run on g.

Let a be a constant. According to the definition of Δ', $a \xrightarrow{g'(a)} r'(a) \in \Delta'$ implies that there exists l_a such that $(l_a, g'(a)) \in R$ and $a \xrightarrow{l_a} r'(a) \in \Delta$. So let $g(a) = l_a$.

Let $t = f(t_1, \dots, t_p)$ with $\forall i\ r'(t_i) = q_i$. There exists a rule $f(q_1, \dots, q_p) \xrightarrow{g'(t)} r'(t)$ in Δ' because r' is a run on g' and again, from the definition of Δ', there exists $l_t \in E_1$ such that $f(q_1, \dots, q_p) \xrightarrow{l_t} r'(t)$ in Δ with $(l_t(t), g'(t)) \in R$. So, we define $g(t) = l_t$. Clearly, g is a generalized tree set and r' is a successful run on g and for all $t \in T(\mathcal{F})$, $(g(t), g'(t)) \in R$ by construction.

$\subseteq$ Let $g' \in R(G)$ and let $g \in G$ such that $\forall t \in T(\mathcal{F})\ (g(t), g'(t)) \in R$. One can easily prove that any successful $\mathcal{A}$-run on g is also a successful $\mathcal{A}'$-run on g'.

□

Let us recall that if g is a generalized tree set in $\mathcal{G}_{E_1 \times \cdots \times E_n}$, the ith projection of g (on the E_i-component, $1 \leq i \leq n$) is the GTS $\pi_i(g)$ defined by: let π from $E_1 \times \cdots \times E_n$ into E_i, such that $\pi(l_1, \dots, l_n) = l_i$ and let $\pi_i(g)(t) = \pi(g(t))$ for every term t. Conversely, the ith cylindrification of a GTS g denoted by $\pi_i^{-1}(g)$ is the set of GTS g' such that $\pi_i(g') = g$. Projection and cylindrification are usually extended to sets of GTS.

Corollary 5.3.4. *(a) The class of GTSA-recognizable languages is closed under projection and cylindrification.*

(b) Let $G \subseteq \mathcal{G}_E$ and $G' \subseteq \mathcal{G}_{E'}$ be two GTSA-recognizable languages. The set $G \uparrow G' = \{g \uparrow g' \mid g \in G,\ g' \in G'\}$ is a GTSA-recognizable language in $\mathcal{G}_{E \times E'}$.

Proof. (a) The case of projection is an immediate consequence of Lemma 5.3.3 using $E_1 = E \times E'$, $E_2 = E$, and $R = \pi$ where π is the projection from $E \times E'$ into E. The case of cylindrification is proved in a similar way.

(b) Consequence of (a) and of Proposition 5.3.1 because $G \uparrow G' = \pi_1^{-1}(G) \cap \pi_2^{-1}(G')$ where π_1^{-1} (respectively π_2^{-1}) is the inverse projection from E to $E \times E'$ (respectively from E' to $E \times E'$).

Let us remark that the construction preserves simplicity, so $\mathcal{R}_{\mathrm{SGTS}}$ is closed under projection and cylindrification.

□

We now consider the case $E = \{0,1\}^n$ and we give two propositions without proof. Proposition 5.3.5 can easily be deduced from Corollary 5.3.4. The proof of Proposition 5.3.6 is an extension of the constructions made in Examples 5.2.1.1 and 5.2.1.2.

Proposition 5.3.5. *Let $\mathcal{A}$ and $\mathcal{A}'$ be two generalized tree set automata over $\{0,1\}^n$.*

(a) $\{(L_1 \cup L'_1, \dots, L_n \cup L'_n) \mid (L_1, \dots, L_n) \in \mathcal{L}(\mathcal{A})$ and $(L'_1, \dots, L'_n) \in \mathcal{L}(\mathcal{A}')\}$ is recognizable.

(b) $\{(L_1 \cap L'_1, \dots, L_n \cap L'_n) \mid (L_1, \dots, L_n) \in \mathcal{L}(\mathcal{A})$ and $(L'_1, \dots, L'_n) \in \mathcal{L}(\mathcal{A}')\}$ is recognizable.

(c) $\{(\overline{L_1}, \dots, \overline{L_n}) \mid (L_1, \dots, L_n) \in \mathcal{L}(\mathcal{A})\}$ is recognizable, where $\overline{L_i} = T(\mathcal{F}) - L_i$, $\forall i$.

Proposition 5.3.6. *Let $E = \{0,1\}^n$ and let $(F_1, \ldots, F_n)$ be a n-tuple of regular tree languages. There exist deterministic simple generalized tree set automata $\mathcal{A}$, $\mathcal{A}'$, and $\mathcal{A}''$ such that*

- $\mathcal{L}(\mathcal{A}) = \{(F_1, \ldots, F_n)\}$;
- $\mathcal{L}(\mathcal{A}') = \{(L_1, \ldots, L_n) \mid L_1 \subseteq F_1, \ldots, L_n \subseteq F_n\}$;
- $\mathcal{L}(\mathcal{A}'') = \{(L_1, \ldots, L_n) \mid F_1 \subseteq L_1, \ldots, F_n \subseteq L_n\}$.

5.3.2 Emptiness Property

Theorem 5.3.7. *The emptiness property is decidable in the class of generalized tree set automata. Given a generalized tree set automaton $\mathcal{A}$, it is decidable whether $\mathcal{L}(\mathcal{A}) = \emptyset$.*

Labels of the generalized tree sets are meaningless for the emptiness decision thus we consider “label-free” generalized tree set automata. Briefly, the transition relation of a “label-free” generalized tree set automata is a relation $\Delta \subseteq \cup_p Q^p \times \mathcal{F}_p \times Q$.

The emptiness decision algorithm for simple generalized tree set automata is straightforward. Indeed, Let ω be a subset of Q and let $\mathsf{COND}(\omega)$ be the following condition:

$$\forall p \ \forall f \in \mathcal{F}_p \ \forall q_1, \ldots, q_p \in \omega \ \exists q \in \omega \quad (q_1, \ldots, q_p, f, q) \in \Delta$$

We easily prove that there exists a set ω satisfying $\mathsf{COND}(\omega)$ if and only if there exists an $\mathcal{A}$-run. Therefore, the emptiness problem for simple generalized tree set automata is decidable because 2^Q is finite and $\mathsf{COND}(\omega)$ is decidable. Decidability of the emptiness problem for simple generalized tree set automata is NP-complete (see Prop. 5.3.9).

The proof is more intricate in the general case, and it is not given in this book. Without the property of simple GTSA, we have to deal with a reachability problem of a set of states since we have to check that there exists $\omega \in \Omega$ and a run r such that r assumes exactly all the states in ω.

We conclude this section with a complexity result of the emptiness problem in the class of generalized tree set automata.

Let us remark that a finite initial fragment of a “label-free” generalized tree set corresponds to a finite set of terms that is closed under the subterm relation. The size or the number of nodes in such an initial fragment is the number of different terms in the subterm-closed set of terms (the cardinality of the set of terms). The size of a GTSA is given by:

$$\|\mathcal{A}\| = |Q| + \sum_{f(q_1,\ldots,q_p) \xrightarrow{l} q \in \Delta} (arity(f) + 3) + \sum_{\omega \in \Omega} |\omega|.$$

Let us consider a GTSA $\mathcal{A}$ with n states. The proof shows that one must consider at most all initial fragments of runs —hence corresponding to finite tree languages closed under the subterm relation— of size smaller than $B(\mathcal{A})$, a polynomial in n, in order to decide emptiness for $\mathcal{A}$. Let us remark that the polynomial bound $B(\mathcal{A})$ can be computed. The emptiness proofs relies on the following lemma:

Lemma 5.3.8. *There exists a polynomial function f of degree 4 such that:*

> *Let $\mathcal{A} = (Q, \Delta, \Omega)$ be a GTSA. There exists a successful run r_s such that $r_s(T(\mathcal{F})) = \omega \in \Omega$ if and only if there exists a run r_m and a closed tree language F such that:*
>
> - $r_m(T(\mathcal{F})) = r_m(F) = \omega$;
> - $\mathsf{Card}(F) \leq f(n)$ *where n is the number of states in ω.*

Proposition 5.3.9. *The emptiness problem in the class of (simple) generalized tree set automata is NP-complete.*

Proof. Let $\mathcal{A} = (Q, \Delta, \Omega)$ be a generalized tree set automaton over E. Let $n = \mathsf{Card}(Q)$.

We first give a non-deterministic and polynomial algorithm for deciding emptiness: (1) take a tree language F closed under the subterm relation such that the number of different terms in it is smaller than $B(\mathcal{A})$; (2) take a run r on F; (3) compute $r(F)$; (4) check whether $r(F) = \omega$ is a member of Ω; (5) check whether ω satisfies $\mathsf{COND}(\omega)$.

From Theorem 5.3.7, this algorithm is correct and complete. Moreover, this algorithm is polynomial in n since (1) the size of F is polynomial in n: step (2) consists in labeling the nodes of F with states following the rules of the automaton – so there is a polynomial number of states, step (3) consists in collecting the states; step (4) is polynomial and non-deterministic and finally, step (5) is polynomial.

We reduce the satisfiability problem of boolean expressions into the emptiness problem for generalized tree set automata. We first build a generalized tree set automaton $\mathcal{A}$ such that $L(\mathcal{A})$ is the set of (codes of) satisfiable boolean expressions over n variables $\{x_1, \ldots, x_n\}$.

Let $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2$ where $\mathcal{F}_0 = \{x_1, \ldots, x_n\}$, $\mathcal{F}_1 = \{\neg\}$, and $\mathcal{F}_2 = \{\wedge, \vee\}$. A boolean expression is a term of $T(\mathcal{F})$. Let $\mathsf{Bool} = \{0, 1\}$ be the set of boolean values. Let $\mathcal{A} = (Q, \Delta, \Omega)$, be a generalized tree set automaton such that $Q = \{q_0, q_1\}$, $\Omega = 2^Q$ and Δ is the following set of rules:

$$\begin{aligned}
x_j &\xrightarrow{i} q_i \text{ where } j \in \{1, \ldots, n\} \text{ and } i \in \mathsf{Bool} \\
\neg(q_i) &\xrightarrow{\neg i} q_{\neg i} \text{ where } i \in \mathsf{Bool} \\
\vee(q_{i_1}, q_{i_2}) &\xrightarrow{i_1 \vee i_2} q_{i_1 \vee i_2} \text{ where } i_1, i_2 \in \mathsf{Bool} \\
\wedge(q_{i_1}, q_{i_2}) &\xrightarrow{i_1 \wedge i_2} q_{i_1 \wedge i_2} \text{ where } i_1, i_2 \in \mathsf{Bool}
\end{aligned}$$

One can easily prove that $L(\mathcal{A}) = \{L_v \mid v \textit{ is a valuation of } \{x_1, \ldots, x_n\}\}$ where $L_v = \{t \mid t \textit{ is a boolean expression which is true under } v\}$. L_v corresponds to a run r_v on a GTS g_v and g_v labels each x_j either by 0 or 1. Hence, g_v can be considered as a valuation v of $x_1, \ldots, x_n$. This valuation is extended in g_v to every node, that is to say that every term (representing a boolean expression) is labeled either by 0 or 1 accordingly to the usual interpretation of $\neg$, $\wedge$, $\vee$. A given boolean expression is hence labeled by 1 if and only if it is true under the valuation v.

Now, we can derive an algorithm for the satisfiability of any boolean expression e: build $\mathcal{A}_e$ a generalized tree set automaton such that $\mathcal{L}(\mathcal{A})$ is the set of all tree languages containing e: $\{L \mid e \in L\}$; build $\mathcal{A}_e \cap \mathcal{A}$ and decide emptiness.

We get then the reduction because $\mathcal{A}_e \cap \mathcal{A}$ is empty if and only if e is not satisfiable.

Now, it remains to prove that the reduction is polynomial. The size of $\mathcal{A}$ is $2 * n + 10$. The size of $\mathcal{A}_e$ is the length of e plus a constant. So we get the result. ☐

5.3.3 Other Decision Results

Proposition 5.3.10. *The inclusion problem and the equivalence problem for deterministic generalized tree set automata are decidable.*

Proof. These results are a consequence of the closure properties under intersection and complementation (Propositions 5.3.1, 5.3.2), and the decidability of the emptiness property (Theorem 5.3.7). ☐

Proposition 5.3.11. *Let $\mathcal{A}$ be a generalized tree set automaton. It is decidable whether or not $\mathcal{L}(\mathcal{A})$ is a singleton set.*

Proof. Let $\mathcal{A}$ be a generalized tree set automaton. First it is decidable whether $\mathcal{L}(\mathcal{A})$ is empty or not (Theorem 5.3.7). Second if $\mathcal{L}(\mathcal{A})$ is non empty then a regular generalized tree set g in $\mathcal{L}(\mathcal{A})$ can be constructed (see the proof of Theorem 5.3.7). Construct the strongly deterministic generalized tree set automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}')$ is a singleton set reduced to the generalized tree set g. Finally, build $\mathcal{A} \cap \overline{\mathcal{A}}'$ to decide the equivalence of $\mathcal{A}$ and $\mathcal{A}'$. Note that we can build $\overline{\mathcal{A}}'$, since $\mathcal{A}'$ is deterministic (see Proposition 5.3.2). ☐

Proposition 5.3.12. *Let $L = (L_1, \ldots, L_n)$ be a tuple of regular tree language and let $\mathcal{A}$ be a generalized tree set automaton over $\{0,1\}^n$. It is decidable whether $L \in \mathcal{L}(\mathcal{A})$.*

Proof. This result just follows from closure under intersection and emptiness decidability.

First construct a (strongly deterministic) generalized tree set automaton $\mathcal{A}_L$ such that $L(\mathcal{A})$ is reduced to the singleton set $\{L\}$. Second, construct $\mathcal{A} \cap \mathcal{A}_L$ and decide whether $L(\mathcal{A} \cap \mathcal{A}_L)$ is empty or not. ☐

Proposition 5.3.13. *Given a generalized tree set automaton over $E = \{0, 1, \}^n$ and $I \subseteq \{1, \ldots, n\}$. The following two problems are decidable:*

1. *It is decidable whether or not there exists $(L_1, \ldots, L_n)$ in $\mathcal{L}(\mathcal{A})$ such that all the L_i are finite for $i \in I$.*

2. *Let $x_1 \ldots, x_n$ be natural numbers. It is decidable whether or not there exists $(L_1, \ldots, L_n)$ in $\mathcal{L}(\mathcal{A})$ such that $\mathsf{Card}(L_i) = x_i$ for each $i \in I$.*

The proof is technical and not given in this book. It relies on Lemma 5.3.8 of the emptiness decision proof.

5.4 Applications to Set Constraints

In this section, we consider the satisfiability problem for systems of set constraints. We show a decision algorithm using generalized tree set automata.

5.4.1 Definitions

Let $\mathcal{F}$ be a finite and non-empty set of function symbols. Let $\mathcal{X}$ be a set of variables. We consider special symbols $\top$, $\bot$, $\sim$, $\cup$, $\cap$ of respective arities 0, 0, 1, 2, 2. A *set expression* is a term in $T_{\mathcal{F}'}(\mathcal{X})$ where $\mathcal{F}' = \mathcal{F} \cup \{\top, \bot, \sim, \cup, \cap\}$.

A set constraint is either a *positive* set constraint of the form $e \subseteq e'$ or a *negative* set constraint of the form $e \not\subseteq e'$ (or $\neg(e \subseteq e')$) where e and e' are set expressions, and a system of set constraints is defined by $\bigwedge_{i=1}^{k} SC_i$ where the SC_i are set constraints.

An interpretation $\mathcal{I}$ is a mapping from $\mathcal{X}$ into $2^{T(\mathcal{F})}$. It can immediately be extended to set expressions in the following way:

$$\begin{aligned}
\mathcal{I}(\top) &= T(\mathcal{F});\\
\mathcal{I}(\bot) &= \emptyset;\\
\mathcal{I}(f(e_1, \dots, e_p)) &= f(\mathcal{I}(e_1), \dots, \mathcal{I}(e_p));\\
\mathcal{I}(\sim e) &= T(\mathcal{F}) \setminus \mathcal{I}(e);\\
\mathcal{I}(e \cup e') &= \mathcal{I}(e) \cup \mathcal{I}(e');\\
\mathcal{I}(e \cap e') &= \mathcal{I}(e) \cap \mathcal{I}(e').
\end{aligned}$$

We deduce an interpretation of set constraints in $\mathsf{Bool} = \{0, 1\}$, the Boolean values. For a system of set constraints SC, all the interpretations $\mathcal{I}$ such that $\mathcal{I}(SC) = 1$ are called *solutions* of SC. In the remainder, we will consider systems of set constraints of n variables $X_1, \dots, X_n$. We will make no distinction between a *solution* $\mathcal{I}$ of a system of set constraints and a *n-tuple of tree languages* $(\mathcal{I}(X_1), \dots, \mathcal{I}(X_n))$. We denote by $\mathsf{SOL}(SC)$ the set of all solutions of a system of set constraints SC.

5.4.2 Set Constraints and Automata

Proposition 5.4.1. *Let SC be a system of set constraints (respectively of positive set constraints) of n variables $X_1, \dots, X_n$. There exists a deterministic (respectively deterministic and simple) generalized tree set automaton $\mathcal{A}$ over $\{0,1\}^n$ such that $\mathcal{L}(\mathcal{A})$ is the set of characteristic generalized tree sets of the n-tuples $(L_1, \dots, L_n)$ of solutions of SC.*

Proof. First we reduce the problem to a single set constraint. Let $SC = C_1 \wedge \dots \wedge C_k$ be a system of set constraints. A solution of SC satisfies all the constraints C_i. Let us suppose that, for every i, there exists a deterministic generalized tree set automaton $\mathcal{A}_i$ such that $\mathsf{SOL}(C_i) = \mathcal{L}(\mathcal{A})$. As all variables in $\{X_1, \dots, X_n\}$ do not necessarily occur in C_i, using Corollary 5.3.4, we can construct a deterministic generalized tree set automaton $\mathcal{A}_i^n$ over $\{0,1\}^n$ satisfying: $\mathcal{L}(\mathcal{A}_i^n)$ is the set of $(L_1, \dots, L_n)$ which corresponds to solutions of C_i when restricted to the variables of C_i. Using closure under intersection (Proposition

5.3.1), we can construct a deterministic generalized tree set automaton $\mathcal{A}$ over $\{0,1\}^n$ such that $\mathsf{SOL}(SC) = \mathcal{L}(\mathcal{A})$.

Therefore we prove the result for a set constraint SC of n variables $X_1, \dots, X_n$. Let $\mathcal{E}(exp)$ be the set of set variables and of set expression exp with a root symbol in $\mathcal{F}$ which occur in the set expression exp:

$$\mathcal{E}(exp) = \big\{exp' \in T_{\mathcal{F}'}(\mathcal{X}) \mid exp' \trianglelefteq exp \text{ and such that} \\ \text{either } \mathcal{H}ead(exp') \in \mathcal{F} \text{ or } exp' \in \mathcal{X}\big\}.$$

If $SC \equiv exp_1 \subseteq exp_2$ or $SC \equiv exp_1 \not\subseteq exp_2$ then $\mathcal{E}(SC) = \mathcal{E}(exp_1) \cup \mathcal{E}(exp_2)$.

Let us consider a set constraint SC and let φ be a mapping φ from $\mathcal{E}(SC)$ into Bool. Such a mapping is easily extended first to any set expression occurring in SC and second to the set constraint SC. The symbols $\cup$, $\cap$, $\sim$, $\subseteq$ and $\not\subseteq$ are respectively interpreted as $\vee$, $\wedge$, $\neg$, $\Rightarrow$ and $\neg\Rightarrow$.

We now define the generalized tree set automaton $\mathcal{A} = (Q, \Delta, \Omega)$ over $E = \{0,1\}^n$.

- The set of states is Q is the set $\{\varphi \mid \varphi : \mathcal{E}(SC) \to \mathsf{Bool}\}$.
- The transition relation is defined as follows: $f(\varphi_1, \dots, \varphi_p) \xrightarrow{l} \varphi \in \Delta$ where $\varphi_1, \dots, \varphi_p \in Q$, $f \in \mathcal{F}_p$, $l = (l_1, \dots, l_n) \in \{0,1\}^n$, and $\varphi \in Q$ satisfies:

$$\forall i \in \{1, \dots, n\}\ \varphi(X_i) = l_i \tag{5.6}$$

$$\forall e \in \mathcal{E}(SC) \setminus \mathcal{X}\ \ (\varphi(e) = 1) \Leftrightarrow \begin{pmatrix} e = f(e_1, \dots, e_p) \\ \forall i \quad 1 \leq i \leq p \quad \varphi_i(e_i) = 1 \end{pmatrix} \tag{5.7}$$

- The set of accepting sets of states Ω is defined depending on the case of a positive or a negative set constraint.
  - If SC is positive, $\Omega = \{\omega \in 2^Q \mid \forall \varphi \in \omega\ \varphi(SC) = 1\}$;
  - If SC is negative, $\Omega = \{\omega \in 2^Q \mid \exists \varphi \in \omega\ \varphi(SC) = 1\}$.

In the case of a positive set constraint, we can choose the state set $Q = \{\varphi \mid \varphi(SC) = 1\}$ and $\Omega = 2^Q$. Consequently, $\mathcal{A}$ is deterministic and simple.

The correctness of this construction is easy to prove and is left to the reader. □

5.4.3 Decidability Results for Set Constraints

We now summarize results on set constraints. These results are immediate consequences of the results of Section 5.4.2. We use Proposition 5.4.1 to encode sets of solutions of systems of set constraints with generalized tree set automata and then, each point is deduced from Theorem 5.3.7, or Propositions 5.3.6, 5.3.13, 5.3.10, 5.3.11.

Properties on sets of solutions

Satisfiability The satisfiability problem for systems of set constraints is decidable.

Regular solution There exists a regular solution, that is a tuple of regular tree languages, in any non-empty set of solutions.

Inclusion, Equivalence Given two systems of set constraints SC and SC', it is decidable whether or not $\mathsf{SOL}(SC) \subseteq \mathsf{SOL}(SC')$.

Unicity Given a system SC of set constraints, it is decidable whether or not there is a unique solution in $\mathsf{SOL}(SC)$.

Properties on solutions

fixed cardinalities, singletons Given a system SC of set constraints over $(X_1, \ldots, X_n)$, $I \subseteq \{1, \ldots, n\}$, and $x_1 \ldots, x_n$ natural numbers;

- it is decidable whether or not there is a solution $(L_1, \ldots, L_n) \in \mathsf{SOL}(SC)$ such that $\mathsf{Card}(L_i) = x_i$ for each $i \in I$.
- it is decidable whether or not all the L_i are finite for $i \in I$.

In both cases, proofs are constructive and exhibits a solution.

Membership Given SC a system of set constraints over $(X_1, \ldots, X_n)$ and a n-tuple $(L_1, \ldots, L_n)$ of regular tree languages, it is decidable whether or not $(L_1, \ldots, L_n) \in \mathsf{SOL}(SC)$.

Proposition 5.4.2. *Let SC be a system of positive set constraints, it is decidable whether or not there is a least solution in* $\mathsf{SOL}(SC)$.

Proof. Let SC be a system of positive set constraints. Let $\mathcal{A}$ be the deterministic, simple generalized tree set automaton over $\{0,1\}^n$ such that $\mathcal{L}(\mathcal{A}) = \mathsf{SOL}(SC)$ (see Proposition 5.4.1). We define a partial ordering $\preceq$ on $\mathcal{G}_{\{0,1\}^n}$ by:

$$\begin{array}{ll} \forall l, l' \in \{0,1\}^n & l \preceq l' \Leftrightarrow (\forall i \; l(i) \leq l'(i)) \\ \forall g, g' \in \mathcal{G}_{\{0,1\}^n} & g \preceq g' \Leftrightarrow (\forall t \in T(\mathcal{F}) \; g(t) \preceq g'(t)) \end{array}$$

The problem we want to deal with is to decide whether or not there exists a least generalized tree set w.r.t. $\preceq$ in $\mathcal{L}(\mathcal{A})$. To this aim, we first build a minimal solution if it exists, and second, we verify that this solution is unique.

Let ω be a subset of states such that $\mathsf{COND}(\omega)$ (see the sketch of proof page 152). Let $\mathcal{A}_\omega = (\omega, \Delta_\omega, 2^\omega)$ be the generalized tree set automaton $\mathcal{A}$ restricted to state set ω.

Now let ${\Delta_\omega}^{min}$ defined by: for each $(q_1, \ldots, q_p, f) \in \omega^p \times \mathcal{F}_p$, choose in the set Δ_ω one rule $(q_1, \ldots, q_p, f, l, q)$ such that l is minimal w.r.t. $\preceq$. Let ${\mathcal{A}_\omega}^{min} = (\omega, {\Delta_\omega}^{min}, 2^\omega)$. Consequently,

1. There exists only one run r_ω on a unique generalized tree set g_ω in ${\mathcal{A}_\omega}^{min}$ because for all $q_1, \ldots, q_p \in \omega$ and $f \in \mathcal{F}_p$ there is only one rule $(q_1, \ldots, q_p, f, l, q)$ in ${\Delta_\omega}^{min}$;
2. the run r_ω on g_ω is regular;

3. the generalized tree set g_ω is minimal w.r.t. $\preceq$ in $\mathcal{L}(\mathcal{A}_\omega)$.

Points 1 and 2 are straightforward. The third point follows from the fact that $\mathcal{A}$ is deterministic. Indeed, let us suppose that there exists a run r' on a generalized tree set g' such that $g' \prec g_\omega$. Therefore, $\forall t\ g'(t) \preceq g_\omega(t)$, and there exists (w.l.o.g.) a minimal term $u = f(u_1, \ldots, u_p)$ w.r.t. the subterm ordering such that $g'(u) \prec g_\omega(u)$. Since $\mathcal{A}$ is deterministic and $\forall v \lhd u\ g_\omega(v) = g'(v)$, we have $r_\omega(u_i) = r'(u_i)$. Hence, the rule $(r_\omega(u_1), \ldots, r_\omega(u_p), f, g_\omega(u), r_\omega(u))$ is not such that $g_\omega(u)$ is minimal in Δ_ω, which contradicts the hypothesis.

Consider the generalized tree sets g_ω for all subsets of states ω satisfying $\mathsf{COND}(\omega)$. If there is no such g_ω, then there is no least generalized tree set g in $\mathcal{L}(\mathcal{A})$. Otherwise, each generalized tree set defines a n-tuple of regular tree languages and inclusion is decidable for regular tree languages. Hence we can identify a minimal generalized tree set g among all g_ω. This GTS g defines a n-tuple $(F_1, \ldots, F_n)$ of regular tree languages. Let us remark this construction does not ensure that $(F_1, \ldots, F_n)$ is minimal in $\mathcal{L}(\mathcal{A})$.

There is a deterministic, simple generalized tree set automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}')$ is the set of characteristic generalized tree sets of all $(L_1, \ldots, L_n)$ satisfying $F_1 \subseteq L_1, \ldots, F_n \subseteq L_n$ (see Proposition 5.3.6). Let $\mathcal{A}''$ be the deterministic generalized tree set automaton such that $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}')$ (see Proposition 5.3.1). There exists a least generalized tree set w.r.t. $\preceq$ in $\mathcal{L}(\mathcal{A})$ if and only if the generalized tree set automata $\mathcal{A}$ and $\mathcal{A}''$ are equivalent. Since equivalence of generalized tree set automata is decidable (see Proposition 5.3.10) we get the result. ☐

5.5 Bibliographical Notes

We now survey decidability results for satisfiability of set constraints and some complexity issues.

Decision procedures for solving set constraints arise with [Rey69], and Mishra [Mis84]. The aim of these works was to obtain new tools for type inference and type checking [AM91, Hei92, HJ90b, JM79, Mis84, Rey69].

First consider systems of set constraints of the form:

$$X_1 = exp_1, \ldots, X_n = exp_n \tag{5.8}$$

where the X_i are distinct variables and the exp_i are disjunctions of set expressions of the form $f(X_{i_1}, \ldots, X_{i_p})$ with $f \in \mathcal{F}_p$. These systems of set constraints are essentially tree automata, therefore they have a unique solution and each X_i is interpreted as a regular tree language.

Suppose now that the exp_i are set expressions without complement symbols. Such systems are always satisfiable and have a least solution which is regular. For example, the system

$$\begin{aligned}
&\mathsf{Nat} = s(\mathsf{Nat}) \cup 0\\
&X = X \cap \mathsf{Nat}\\
&\mathsf{List} = \mathsf{cons}(X, \mathsf{List}) \cup \mathsf{nil}
\end{aligned}$$

has a least solution

$$\mathsf{Nat} = \{s^i(0) \mid i \geq 0\}, X = \emptyset, \mathsf{List} = \{\mathsf{nil}\}.$$

[HJ90a] investigate the class of definite set constraints which are of the form $exp \subseteq exp'$, where no complement symbol occurs and exp' contains no set operation. Definite set constraints have a least solution whenever they have a solution. The algorithm presented in [HJ90a] provides a specific set of transformation rules and, when there exists a solution, the result is a regular presentation of the least solution, in other words a system of the form (5.8).

Solving definite set constraints is EXPTIME-complete [CP97]. Many developments or improvements of Heinzte and Jaffar's method have been proposed and some are based on tree automata [DTT97].

The class of positive set constraints is the class of systems of set constraints of the form $exp \subseteq exp'$, where no projection symbol occur. In this case, when a solution exists, set constraints do not necessarily have a least solution. Several algorithms for solving systems in this class were proposed, [AW92] generalize the method of [HJ90a], [GTT93, GTT99] give an automata-based algorithm, and [BGW93] use the decision procedure for the first order theory of monadic predicates. Results on the computational complexity of solving systems of set constraints are presented in a paper of [AKVW93]. The systems form a natural complexity hierarchy depending on the number of elements of $\mathcal{F}$ of each arity. The problem of existence of a solution of a system of positive set constraints is NEXPTIME-complete.

The class of positive and negative set constraints is the class of systems of set constraints of the form $exp \subseteq exp'$ or $exp \not\subseteq exp'$, where no projection symbol occur. In this case, when a solution exists, set constraints do not necessarily have, neither a minimal solution, nor a maximal solution. Let $\mathcal{F} = \{a, b()\}$. Consider the system $(b(X) \subseteq X) \wedge (X \not\subseteq \bot)$, this system has no minimal solution. Consider the system $(X \subseteq b(X) \cup a) \wedge (\top \not\subseteq X)$, this system has no maximal solution. The satisfiability problem in this class turned out to be much more difficult than the positive case. [AKW95] give a proof based on a reachability problem involving Diophantine inequalities. NEXPTIME-completeness was proved by [Ste94]. [CP94a] gives a proof based on the ideas of [BGW93].

The class of positive set constraints with projections is the class of systems of set constraints of the form $exp \subseteq exp'$ with projection symbols. Set constraints of the form $f_i^{-1}(X) \subseteq Y$ can easily be solved, but the case of set constraints of the form $X \subseteq f_i^{-1}(Y)$ is more intricate. The problem was proved decidable by [CP94b].

The expressive power of these classes of set constraints have been studied and have been proved to be different [Sey94]. In [CK96, Koz93], an axiomatization is proposed which enlightens the reader on relationships between many approaches on set constraints.

Furthermore, set constraints have been studied in a logical and topological point of view [Koz95, MGKW96]. This last paper combine set constraints with Tarskian set constraints, a more general framework for which many complexity results are proved or recalled. Tarskian set constraints involve variables, relation and function symbols interpreted relative to a first order structure.

Topological characterizations of classes of GTSA recognizable sets, have also been studied in [Tom94, Sey94]. Every set in $\mathcal{R}_{\mathrm{SGTS}}$ is a compact set and every set in $\mathcal{R}_{\mathrm{GTS}}$ is the intersection between a compact set and an open set. These remarks give also characterizations for the different classes of set constraints.

Chapter 6

Tree Transducers

6.1 Introduction

Finite state transformations of words, also called a-transducers or rational transducers in the literature, model many kinds of processes, such as coffee machines or lexical translators. But these transformations are not powerful enough to model syntax directed transformations, and compiler theory is an important motivation to the study of finite state transformations of trees. Indeed, translation of natural or computing languages is directed by syntactical trees, and a translator from LaTeX into HTML is a tree transducer. Unfortunately, from a theoretical point of view, tree transducers do not inherit nice properties of word transducers, and the classification is very intricate. So, in the present chapter we focus on some aspects. In Sections 6.2 and 6.3, toy examples introduce in an intuitive way different kinds of transducers. In Section 6.2, we summarize main results in the word case. Indeed, this book is mainly concerned with trees, but the word case is useful to understand the tree case and its difficulties. The bimorphism characterization is the ideal illustration of the link between the "machine" point of view and the "homomorphic" one. In Section 6.3, we motivate and illustrate bottom-up and top-down tree transducers, using compilation as leitmotiv. We precisely define and present the main classes of tree transducers and their properties in Section 6.4, where we observe that general classes are not closed under composition, mainly because of alternation of copying and nondeterministic processing. Nevertheless most useful classes, as those used in Section 6.3, have closure properties. In Section 6.5 we present the homomorphic point of view.

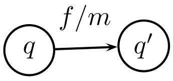
Most of the proofs are tedious and are omitted. This chapter is a very incomplete introduction to tree transducers. Tree transducers are extensively studied for themselves and for various applications. But as they are somewhat complicated objects, we focus here on the definitions and main general properties. It is useful for every theoretical computer scientist to know main notions about tree transducers, because they are the main model of syntax directed manipulations, and that the heart of software manipulations and interfaces are syntax directed. Tree transducers are an essential frame to develop practical modular syntax directed algorithms, thought an effort of algorithmic engineering remains to do. Tree transducers theory can be fertilized by other area or can be useful for other areas (example: Ground tree transducers for decidability of the first order theory of ground rewriting). We will be happy if after reading this chapter, the reader wants for further lectures, as monograph of Z. Fülöp and H. Vögler (December 1998 [FV98]).

6.2 The Word Case

6.2.1 Introduction to Rational Transducers

We assume that the reader roughly knows popular notions of language theory: homomorphisms on words, finite automata, rational expressions, regular grammars. See for example the recent survey of A. Mateescu and A. Salomaa [MS96]. A rational transducer is a finite word automaton W with output. In a word automaton, a transition rule $f(q) \rightarrow q'(f)$ means "if W is in some state q, if it reads the input symbol f, then it enters state q' and moves its head one symbol to the right". For defining a rational transducer, it suffices to add an output, and a transition rule $f(q) \rightarrow q'(m)$ means "if the transducer is in some state q, if it reads the input symbol f, then it enters state q', writes the word m on the output tape, and moves its head one symbol to the right". Remark that with these notations, we identify a finite automaton with a rational transducer which writes what it reads. Note that m is not necessarily a symbol but can be a word, including the empty word. Furthermore, we assume that it is not necessary to read an input symbol, i.e. we accept transition rules of the form $\varepsilon(q) \rightarrow q'(m)$ (ε denotes the empty word).

Graph presentations of finite automata are popular and convenient. So it is for rational transducers. The rule $f(q) \rightarrow q'(m)$ will be drawn



Example 6.2.1. **(Language L_1)** Let $\mathcal{F} = \{\langle, \rangle, ;, 0, 1, A, ..., Z\}$. In the following, we will consider the language L_1 defined on $\mathcal{F}$ by the regular grammar (the axiom is **program**):

program → ⟨ **instruct**
instruct → `LOAD` **register** | `STORE` **register** | `MULT` **register**
→ | `ADD` **register**
register → 1**tailregister**
tailregister → 0**tailregister** | 1**tailregister** | ; **instruct** | ⟩

($a \rightarrow b|c$ is an abbreviation for the set of rules $\{a \rightarrow b, a \rightarrow c\}$)

L_1 is recognized by deterministic automaton A_1 of Figure 6.1. Semantic of L_1 is well known: `LOAD i` loads the content of register `i` in the accumulator; `STORE i` stores the content of the accumulator in register `i`; `ADD i` adds in the accumulator the content of the accumulator and the content of register `i`; `MULT i` multiplies in the accumulator the content of the accumulator and the content of register `i`.

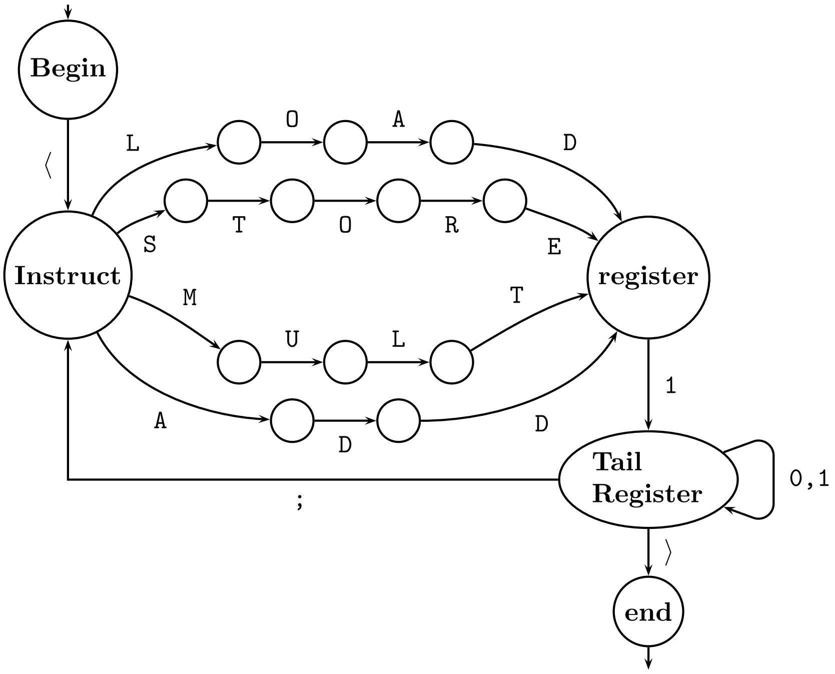

Begin
⟨
Instruct
L
O
A
D
S
T
O
R
E
register
M
U
L
T
A
D
D
1
Tail Register
0,1
;
⟩
end

Figure 6.1: A recognizer of L_1

A ***rational transducer*** is a tuple $R = (Q, \mathcal{F}, \mathcal{F}', Q_i, Q_f, \Delta)$ where Q is a set of states, $\mathcal{F}$ and $\mathcal{F}'$ are finite nonempty sets of input letters and output letters, $Q_i, Q_f \subseteq Q$ are sets of initial and final states and Δ is a set of transduction rules of the following type:

$$f(q) \to q'(m),$$

where $f \in \mathcal{F} \cup \{\varepsilon\}$, $m \in {\mathcal{F}'}^*$, $q, q' \in Q$.

R is ***ε-free*** if there is no rule $f(q) \to q'(m)$ with $f = \varepsilon$ in Δ.

The ***move relation*** $\to_R$ is defined by: let $t, t' \in \mathcal{F}^*$, $u \in {\mathcal{F}'}^*$, $q, q' \in Q$, $f \in \mathcal{F}$, $m \in {\mathcal{F}'}^*$,

$$(tqft', u) \underset{R}{\to} (tfq't', um) \Leftrightarrow f(q) \to q'(m) \in \Delta,$$

and $\to_R^*$ is the reflexive and transitive closure of $\to_R$. A (partial) transduction of R on $tt't''$ is a sequence of move steps of the form $(tqt't'', u) \to_R^* (tt'q't'', uu')$. A transduction of R from $t \in \mathcal{F}^*$ into $u \in {\mathcal{F}'}^*$ is a transduction of the form $(qt, \varepsilon) \to_R^* (tq', u)$ with $q \in Q_i$ and $q' \in Q_f$.

The relation T_R induced by R can now be formally defined by:

$$T_R = \{(t, u) \mid (qt, \varepsilon) \overset{*}{\underset{R}{\to}} (tq', u) \text{ with } t \in \mathcal{F}^*, u \in {\mathcal{F}'}^*, q \in Q_i, q' \in Q_f\}.$$

A relation in $\mathcal{F}^* \times {\mathcal{F}'}^*$ is a rational transduction if and only if it is induced by some rational transducer. We also need the following definitions: let $t \in \mathcal{F}^*$, $T_R(t) = \{u \mid (t, u) \in T_R\}$. The translated of a language L is the language defined by $T_R(L) = \{u \mid \exists t \in L, u \in T_R(t)\}$.

Example 6.2.2.

Ex. 6.2.2.1 Let us name French-L_1 the translation of L_1 in French (`LOAD` is translated into `CHARGER` and `STORE` into `STOCKER`). Transducer of Figure 6.2 achieves this translation. This example illustrates the use of rational transducers as lexical transducers.

Ex. 6.2.2.2 Let us consider the rational transducer *Diff* defined by $Q = \{q_i, q_s, q_l, q_d\}$, $\mathcal{F} = \mathcal{F}' = \{a, b\}$, $Q_i = \{q_i\}$, $Q_f = \{q_s, q_l, q_d\}$, and Δ is the set of rules:

type i(dentical) $a(q_i) \to q_i(a)$, $b(q_i) \to q_i(b)$

type s(horter) $\varepsilon(q_i) \to q_s(a)$, $\varepsilon(q_i) \to q_s(b)$, $\varepsilon(q_s) \to q_s(a)$, $\varepsilon(q_s) \to q_s(b)$

type l(onger) $a(q_i) \to q_l(\varepsilon)$, $b(q_i) \to q_l(\varepsilon)$, $a(q_l) \to q_l(\varepsilon)$, $b(q_l) \to q_l(\varepsilon)$

type d(ifferent) $a(q_i) \to q_d(b)$, $b(q_i) \to q_d(a)$, $a(q_d) \to q_d(\varepsilon)$, $b(q_d) \to q_d(\varepsilon)$, $\varepsilon(q_d) \to q_d(a)$, $\varepsilon(q_d) \to q_d(b)$.

It is easy to prove that $T_{\mathit{Diff}} = \{(m, m') \mid m \neq m', m, m' \in \{a, b\}^*\}$.

We give without proofs some properties of rational transducers. For more details, see [Sal73] or [MS96] and Exercises 6.1, 6.2, 6.4 for 1, 4 and 5. The homomorphic approach presented in the next section can be used as an elegant way to prove 2 and 3 (Exercise 6.6).

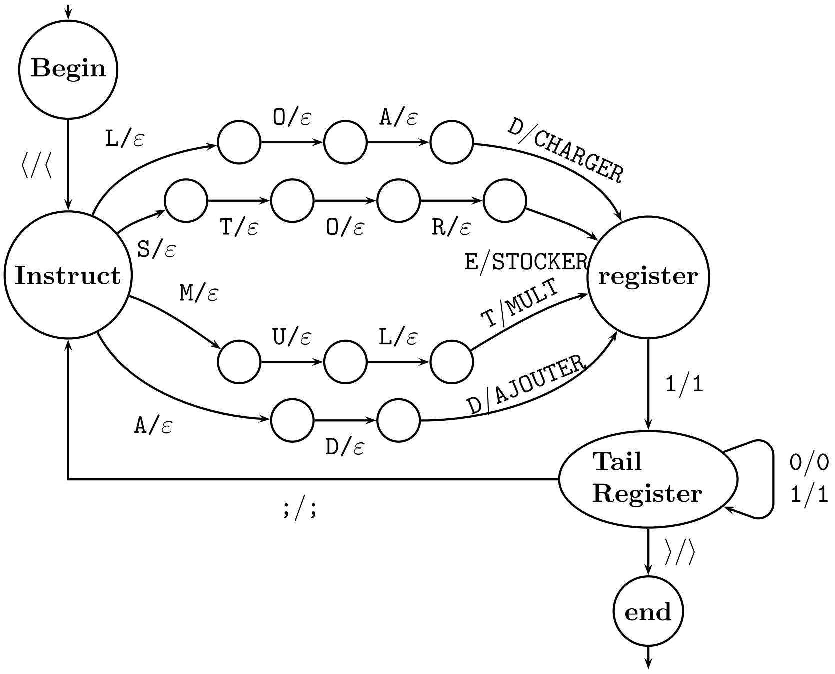

Begin
⟨/⟨
Instruct
L/ε
O/ε
A/ε
D/CHARGER
S/ε
T/ε
O/ε
R/ε
E/STOCKER
register
M/ε
U/ε
L/ε
T/MULT
A/ε
D/ε
D/AJOUTER
1/1
Tail
Register
0/0
1/1
;/;
⟩/⟩
end

Figure 6.2: A rational transducer from L_1 into French-L_1.

Proposition 6.2.3 (Main properties of rational transducers)**.**

1. *The class of rational transductions is closed under union but not closed under intersection.*
2. *The class of rational transductions is closed under composition.*
3. *Regular languages and context-free languages are closed under rational transduction.*
4. *Equivalence of rational transductions is undecidable.*
5. *Equivalence of deterministic rational transductions is decidable.*

6.2.2 The Homomorphic Approach

A ***bimorphism*** is defined as a triple $B = (\Phi, L, \Psi)$ where L is a recognizable language and Φ and Ψ are homomorphisms. The relation induced by B (also denoted by B) is defined by $B = \{(\Phi(t), \Psi(t)) \mid t \in L\}$. Bimorphism (Φ, L, Ψ) is ε-free if Φ is ε-free (an homomorphism is ε-free if the image of a letter is never reduced to ε). Two bimorphisms are equivalent if they induce the same relation.

We can state the following theorem, generally known as Nivat Theorem [Niv68] (see Exercises 6.5 and 6.6 for a sketch of proof).

Theorem 6.2.4 (Bimorphism theorem)**.** *Given a rational transducer, an equivalent bimorphism can be constructed. Conversely, any bimorphism defines a rational transduction. Construction preserves* ε*-freeness.*

Example 6.2.5.

Ex. 6.2.5.1 The relation $\{(a(ba)^n, a^n) \mid n \in \mathbb{N}\} \cup \{((ab)^n, b^{3n}) \mid n \in \mathbb{N}\}$ is processed by transducer R and bimorphism B of Figure 6.3

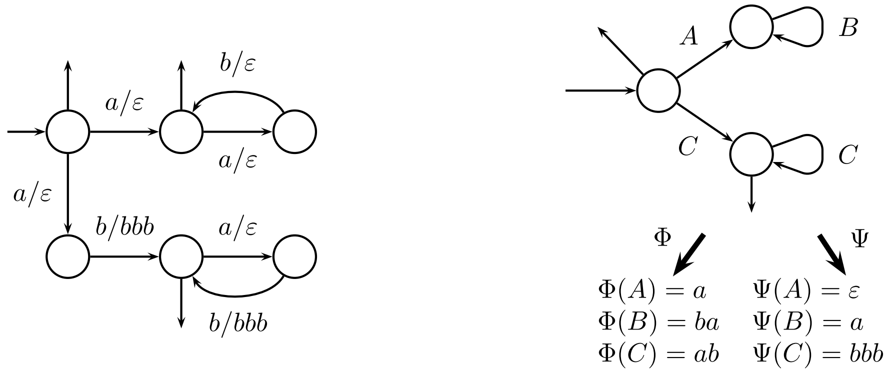


Figure 6.3: Transducer R and an equivalent bimorphism $B = \{(\Phi(t), \Psi(t)) \mid t \in \epsilon + AB^* + CC^*\}$.

Ex. 6.2.5.2 Automaton L of Figure 6.4 and morphisms Φ and Ψ below define a bimorphism equivalent to the transducer in Figure 6.2

$\Phi(\beta) = \langle$ $\Phi(\lambda) = \texttt{LOAD}$ $\Phi(\sigma) = \texttt{STORE}$ $\Phi(\mu) = \texttt{MULT}$
$\Phi(\alpha) = \texttt{ADD}$ $\Phi(\rho) = ;$ $\Phi(\omega) = 1$ $\Phi(\zeta) = 0$
$\Phi(\theta) = \rangle$
$\Psi(\beta) = \langle$ $\Psi(\lambda) = \texttt{CHARGER}$ $\Psi(\sigma) = \texttt{STOCKER}$ $\Psi(\mu) = \texttt{MULT}$
$\Psi(\alpha) = \texttt{ADD}$ $\Psi(\rho) = ;$ $\Psi(\omega) = 1$ $\Psi(\zeta) = 0$
$\Psi(\theta) = \rangle$

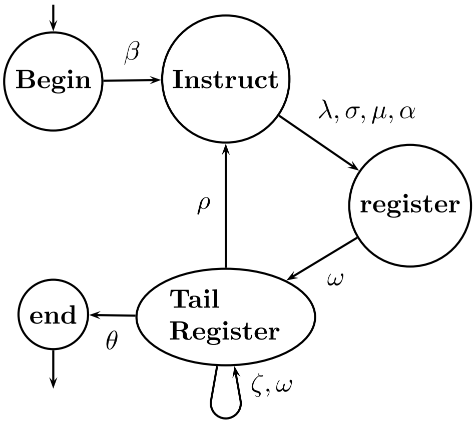


Figure 6.4: The control automaton L.

Nivat characterization of rational transducers makes intuitive sense. Automaton L can be seen as a control of the actions, morphism Ψ can be seen as output function and Φ^{-1} as an input function. Φ^{-1} analyses the input — it is a kind of part of lexical analyzer — and it generates symbolic names; regular grammatical structure on theses symbolic names is controlled by L. Examples 6.2.5.1 and 6.2.5.2 are an obvious illustration. L is the common structure to English and French versions, Φ generates the English version and Ψ generates the French one. This idea is the major idea of compilation, but compilation of computing languages or translation of natural languages are directed by syntax, that is to say by syntactical trees. This is the motivation for the sequel of the chapter. But unfortunately, from a formal point of view, we will loose most of the nice properties obtained in the word case. Expressivity of non-linear tree transducers will explain in part this complication, but, even in the linear case, there is a new phenomenon in trees, the understanding of which can be introduced by the "problem of homomorphism inversion" that we describe in Exercise 6.7.

6.3 Introduction to Tree Transducers

Tree transducers and their generalizations model many syntax directed transformations (see exercises). We use here a toy example of compiler to illustrate how usual tree transducers can be considered as modules of compilers.

We consider a simple class of arithmetic expressions (with usual syntax) as source language. We assume that this language is analyzed by a `LL(1)` parser.

We consider two target languages: L_1 defined in Example 6.2.1 and an other language L_2. A transducer A translates syntactical trees in abstract trees (Figure 6.5). A second tree transducer R illustrates how tree transducers can be seen as part of compilers which compute attributes over abstract trees. It decorates abstract trees with numbers of registers (Figure 6.7). Thus R translates abstract trees into attributed abstract trees. After that, tree transducers T_1 and T_2 generate target programs in L_1 and L_2, respectively, starting from attributed abstract trees (Figures 6.7 and 6.8). This is an example of nonlinear transducer. Target programs are yields of generated trees. So composition of transducers model succession of compilation passes, and when a class of transducers is closed by composition (see section 6.4), we get universal constructions to reduce the number of compiler passes and to meta-optimize compilers.

We now define **the source language**. Let us consider the terminal alphabet $\{(,),+,\times,a,b,\ldots,z\}$. First, the context-free word grammar G_1 is defined by rules (E is the axiom):

$$\begin{array}{rcl} E & \rightarrow & M \mid M+E \\ M & \rightarrow & F \mid F \times M \\ F & \rightarrow & I \mid (E) \\ I & \rightarrow & a \mid b \mid \cdots \mid z \end{array}$$

Another context-free word grammar G_2 is defined by (E is the axiom):

$$\begin{array}{rcl} E & \rightarrow & ME' \\ E' & \rightarrow & +E \mid \varepsilon \\ M & \rightarrow & FM' \\ M' & \rightarrow & \times M \mid \varepsilon \\ F & \rightarrow & I \mid (E) \\ I & \rightarrow & a \mid b \mid \cdots \mid z \end{array}$$

It is easy to prove that G_1 and G_2 are equivalent, i.e. they define the same source language. On the one hand, G_1 is more natural, on the other hand G_2 could be preferred for syntactical analysis reason, because G_2 is $\mathsf{LL(1)}$ and G_1 is not LL. We consider **syntactical trees** as derivation trees for the tree grammar G_2.

Let us consider word $u = (a+b) \times c$ of the source language. We define the abstract tree associated with u as the tree $\times(+(a,b),c)$ defined over $\mathcal{F} = \{+(,),\times(,),a,b,c\}$. **Abstract trees** are ground terms over $\mathcal{F}$. The following transformation associates with a syntactical tree t its corresponding abstract tree $A(t)$.

$$\begin{array}{rr} I(x) \rightarrow x & F(x) \rightarrow x \\ M(x, M'(\varepsilon)) \rightarrow x & E(x, E'(\varepsilon)) \rightarrow x \\ M(x, M'(\times, y)) \rightarrow \times(x,y) & E(x, E'(+,y)) \rightarrow +(x,y) \\ F((,x,)) \rightarrow x & \end{array}$$

We have not precisely defined the use of the arrow $\rightarrow$, but it is intuitive. Likewise we introduce examples before definitions of different kinds of tree transducers (section 6.4 supplies a formal frame).

To illustrate nondeterminism, let us introduce two new transducers A and A'. Some brackets are optional in the source language, hence A' is nondeterministic. Note that A works from frontier to root and A' works from root to frontier.

A: an Example of Bottom-up Tree Transducer

The following linear deterministic bottom-up tree transducer A carries out transformation of derivation trees for G_2, i.e. syntactical trees, into their corresponding abstract trees. Empty word ε is identified as a constant symbol in syntactical trees. States of A are q, q_ε, q_I, q_F, $q_{M'\varepsilon}$, $q_{E'\varepsilon}$, q_E, $q_\times$, $q_{M'\times}$, q_+, $q_{E'+}$, $q_($, and $q_)$. Final state is q_E. The set of transduction rules is:

$$
\begin{array}{rlrl}
a & \rightarrow q(a) & b & \rightarrow q(b) \\
c & \rightarrow q(c) & \varepsilon & \rightarrow q_\varepsilon(\varepsilon) \\
) & \rightarrow q_)()) & (& \rightarrow q_((() \\
+ & \rightarrow q_+(+) & \times & \rightarrow q_\times(\times) \\
I(q(x)) & \rightarrow q_I(x) & F(q_I(x)) & \rightarrow q_F(x) \\
M'(q_\varepsilon(x)) & \rightarrow q_{M'\varepsilon}(x) & E'(q_\varepsilon(x)) & \rightarrow q_{E'\varepsilon}(x) \\
M(q_F(x), q_{M'\varepsilon(y)}) & \rightarrow q_M(x) & E(q_M(x), q_{E'\varepsilon}(y)) & \rightarrow q_E(x) \\
M'(q_\times(x), q_M(y)) & \rightarrow q_{M'\times}(y) & M(q_F(x), q_{M'\times}(y)) & \rightarrow q_M(\times(x,y)) \\
E'(q_+(x), q_E(y)) & \rightarrow q_{E'+}(y) & E(q_M(x), q_{E'+}(y)) & \rightarrow q_E(+(x,y)) \\
F(q_((x), q_E(y), q_)(z)) & \rightarrow q_F(y) & &
\end{array}
$$

The notion of (successful) run is an intuitive generalization of the notion of run for finite tree automata. The reader should note that FTAs can be considered as a special case of bottom-up tree transducers whose output is equal to the input. We give in Figure 6.5 a run of A which translates the derivation tree t of $(a+b)\times c$ w.r.t. G_2 into the corresponding abstract tree $\times(+(a,b),c)$.

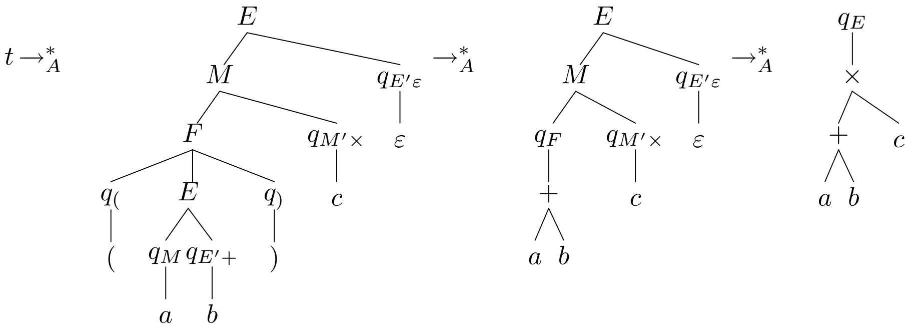


Figure 6.5: Example of run of A

A': an Example of Top-down Tree Transducer

The inverse transformation A^{-1}, which computes the set of syntactical trees associated with an abstract tree, is computed by a nondeterministic top-down tree transducer A'. The states of A' are q_E, q_F, q_M. The initial state is q_E. The set of transduction rules is:

$$\begin{array}{rlrl}
q_E(x) & \rightarrow E(q_M(x), E'(\varepsilon)) & q_E(+(x,y)) & \rightarrow E(q_M(x), E'(+, q_E(y))) \\
q_M(x) & \rightarrow M(q_F(x), M'(\varepsilon)) & q_M(\times(x,y)) & \rightarrow M(q_F(x), M'(\times, q_M(y))) \\
q_F(x) & \rightarrow F((, q_E(x),)) & q_F(a) & \rightarrow F(I(a)) \\
q_F(b) & \rightarrow F(I(b)) & q_F(c) & \rightarrow F(I(c))
\end{array}$$

Transducer A' is nondeterministic because there are ε-rules like $q_E(x) \rightarrow E(q_M(x), E'(\varepsilon))$. We give in Figure 6.6 a run of A' which transforms the abstract tree $+(a, \times(b,c))$ into a syntactical tree t' of the word $a + b \times c$.

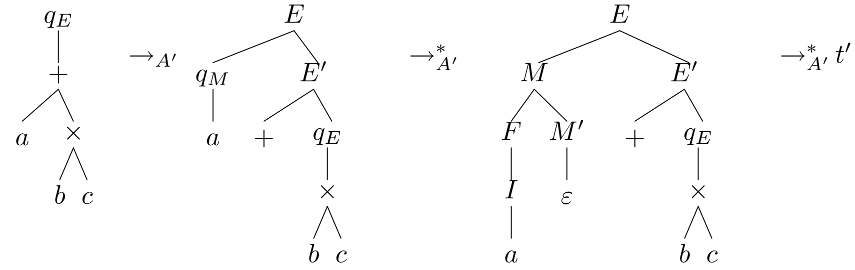


Figure 6.6: Example of run of A'

Compilation

The compiler now transforms abstract trees into programs for some target languages. We consider two target languages. The first one is L_1 of Example 6.2.1. To simplify, we omit ";", because they are not necessary — we introduced semicolons in Section 6.2 to avoid ε-rules, but this is a technical detail, because word (and tree) automata with ε-rules are equivalent to usual ones. The second target language is an other very simple language L_2, namely sequences of two instructions $+(i,j,k)$ (put the sum of contents of registers i and j in the register k) and $\times(i,j,k)$. In a first pass, we attribute to each node of the abstract tree the minimal number of registers necessary to compute the corresponding sub-expression in the target language. The second pass generates target programs.

First pass: computation of register numbers by **a deterministic linear bottom-up transducer** R.

States of a tree automaton can be considered as values of (finitely valued) attributes, but formalism of tree automata does not allow decorating nodes of trees with the corresponding values. On the other hand, this decoration is easy with a transducer. Computation of finitely valued inherited (respectively synthesized) attributes is modeled by top-down (respectively bottom-up) tree transducers. Here, we use a bottom-up tree transducer R. States of R are $q_0, \ldots, q_n$. All states are final states. The set of rules is:

$$
\begin{array}{rlrl}
a & \rightarrow q_0(a) & b & \rightarrow q_0(b) \\
c & \rightarrow q_0(c) & & \\
+(q_i(x), q_i(y)) & \rightarrow q_{i+1}(\underset{i+1}{+}(x,y)) & \times(q_i(x), q_i(y)) & \rightarrow q_{i+1}(\underset{i+1}{\times}(x,y)) \\
\text{if } i > j & & & \\
+(q_i(x), q_j(y)) & \rightarrow q_i(\underset{i}{+}(x,y) & \times(q_i(x), q_j(y)) & \rightarrow q_i(\underset{i}{\times}(x,y)) \\
\multicolumn{4}{l}{\text{if } i < j\text{, we permute the order of subtrees}} \\
+(q_i(x), q_j(y)) & \rightarrow q_j(\underset{j}{+}(y,x)) & \times(q_i(x), q_j(y)) & \rightarrow q_j(\underset{j}{\times}(y,x))
\end{array}
$$

A run $t \rightarrow_R^* q_i(u)$ means that i registers are necessary to evaluate t. Root of t is then relabelled in u by symbol $\underset{i}{+}$ or $\underset{i}{\times}$.

Second pass: generation of target programs in L_1 or L_2, by **top-down deterministic transducers** T_1 **and** T_2. T_1 contains only one state q. Set of rules of T_1 is:

$$
\begin{array}{rl}
q(\underset{i}{+}(x,y)) & \rightarrow \diamond(q(x), \texttt{STORE}i, q(y), \texttt{ADD}i, \texttt{STORE}i) \\
q(\underset{i}{\times}(x,y)) & \rightarrow \diamond(q(x), \texttt{STORE}i, q(y), \texttt{MULT}i, \texttt{STORE}i) \\
q(a) & \rightarrow \diamond(\texttt{LOAD}, a) \\
q(b) & \rightarrow \diamond(\texttt{LOAD}, b) \\
q(c) & \rightarrow \diamond(\texttt{LOAD}, c)
\end{array}
$$

where $\diamond(,,,,)$ and $\diamond(,)$ are new symbols.

State set of T_2 is $\{q, q'\}$ where q is the initial state. Set of rules of T_2 is:

$$
\begin{array}{rlrl}
q(\underset{i}{+}(x,y)) & \rightarrow \#(q(x), q(y), +, (, q'(x), q'(y), i,)) & q'(\underset{i}{+}(x,y)) & \rightarrow i \\
q(\underset{i}{\times}(x,y)) & \rightarrow \#(q(x), q(y), \times, (, q'(x), q'(y), i,)) & q'(\underset{i}{\times}(x,y)) & \rightarrow i \\
q(a) & \rightarrow \varepsilon & q'(a) & \rightarrow a \\
q(b) & \rightarrow \varepsilon & q'(b) & \rightarrow b \\
q(c) & \rightarrow \varepsilon & q'(c) & \rightarrow c
\end{array}
$$

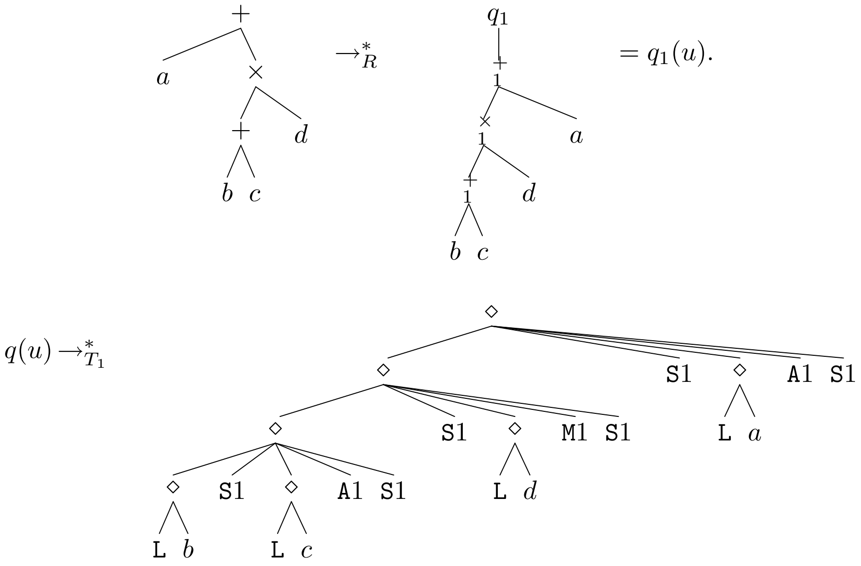
where # is a new symbol of arity 8.

The reader should note that target programs are words formed with leaves of trees, i.e. yields of trees. Examples of transductions computed by T_1 and T_2 are given in Figures 6.7 and 6.8. The reader should also note that T_1 is an homomorphism. Indeed, an homomorphism can be considered as a particular case of deterministic transducer, namely a transducer with only one state (we can consider it as bottom-up as well as top-down). The reader should also note that T_2 is deterministic but not linear.

6.4 Properties of Tree Transducers

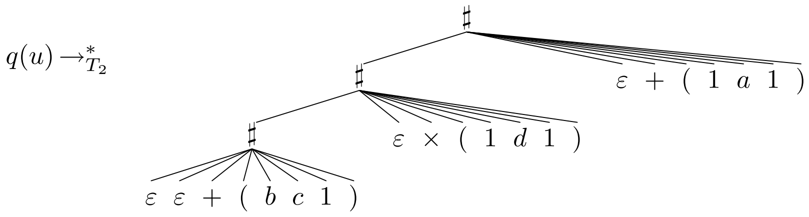
6.4.1 Bottom-up Tree Transducers

We now give formal definitions. In this section, we consider academic examples, without intuitive semantic, to illustrate phenomena and properties. Tree transducers are both generalization of word transducers and tree automata. We first


$\rightarrow_R^*$
$= q_1(u)$.
q_1
$q(u) \rightarrow_{T_1}^*$
S1 A1 S1 M1 S1 L
a b c d

where `L` stands for `LOAD`, `S` stands for `STORE`, `A` stands for `ADD`, `M` stands for `MULT`. The corresponding program is the yield of this tree:
`LOAD`b `STORE1` `LOAD`c `ADD1` `STORE1` `STORE1` `LOAD`d `MULT1` `STORE1` `STORE1` `LOAD`a `ADD1` `STORE1`

Figure 6.7: Decoration with synthesized attributes of an abstract tree, and translation into a target program of L_1.


$q(u) \rightarrow_{T_2}^*$
♯
ε + (1 a 1)
ε × (1 d 1)
ε ε + (b c 1)

The corresponding program is the yield of this tree: $+(bc1) \times (1d1) + (1a1)$

Figure 6.8: Translation of an abstract tree into a target program of L_2

consider bottom-up tree transducers. A transition rule of a NFTA is of the type $f(q_1(x_1), \dots, q_n(x_n)) \to q(f(x_1, \dots, x_n))$. Here we extend the definition (as we did in the word case), accepting to change symbol f into any term.

A ***bottom-up Tree Transducer*** (NUTT) is a tuple $U = (Q, \mathcal{F}, \mathcal{F}', Q_f, \Delta)$ where Q is a set of (unary) states, $\mathcal{F}$ and $\mathcal{F}'$ are finite nonempty sets of input symbols and output symbols, $Q_f \subseteq Q$ is a set of final states and Δ is a set of transduction rules of the following two types:

$$f(q_1(x_1), \dots, q_n(x_n)) \to q(u) \ ,$$

where $f \in \mathcal{F}_n$, $u \in T(\mathcal{F}', \mathcal{X}_n)$, $q, q_1, \dots, q_n \in Q$, or

$$q(x_1) \to q'(u) \quad (\varepsilon\text{-rule}),$$

where $u \in T(\mathcal{F}', \mathcal{X}_1)$, $q, q' \in Q$.

As for NFTA, there is no initial state, because when a symbol is a leave a (i.e. a constant symbol), transduction rules are of the form $a \to q(u)$, where u is a ground term. These rules can be considered as “initial rules”. Let $t, t' \in T(\mathcal{F} \cup \mathcal{F}' \cup Q)$. The move relation $\to_U$ is defined by:

$$t \xrightarrow[U]{} t' \Leftrightarrow \begin{cases} \exists f(q_1(x_1), \dots, q_n(x_n)) \to q(u) \in \Delta \\ \exists C \in \mathcal{C}(\mathcal{F} \cup \mathcal{F}' \cup Q) \\ \exists u_1, \dots, u_n \in T(\mathcal{F}') \\ t = C[f(q_1(u_1), \dots, q_n(u_n))] \\ t' = C[q(u\{x_1 \leftarrow u_1, \dots, x_n \leftarrow u_n\})] \end{cases}$$

This definition includes the case of ε-rule as a particular case. The reflexive and transitive closure of $\to_U$ is $\to_U^*$. A transduction of U from a ground term $t \in T(\mathcal{F})$ to a ground term $t' \in T(\mathcal{F}')$ is a sequence of move steps of the form $t \to_U^* q(t')$, such that q is a final state. The relation induced by U is the relation (also denoted by U) defined by:

$$U = \{(t, t') \mid t \xrightarrow[U]{*} q(t'), t \in T(\mathcal{F}), t' \in T(\mathcal{F}'), q \in Q_f\}.$$

The domain of U is the set $\{t \in T(\mathcal{F}) \mid (t, t') \in U\}$. The image by U of a set of ground terms L is the set $U(L) = \{t' \in T(\mathcal{F}') \mid \exists t \in L, (t, t') \in U\}$.

A transducer is ***ε-free*** if it contains no ε-rule. It is ***linear*** if all transition rules are linear (no variable occurs twice in the right-hand side). It is ***non-erasing*** if, for every rule, at least one symbol of $\mathcal{F}'$ occurs in the right-hand side. It is said to be ***complete*** (or non-deleting) if, for every rule $f(q_1(x_1), \dots, q_n(x_n)) \to q(u)$, for every $x_i (1 \leq i \leq n)$, x_i occurs at least once in u. It is ***deterministic*** (DUTT) if it is ε-free and there is no two rules with the same left-hand side.

Example 6.4.1.

Ex. 6.4.1.1 Tree transducer A defined in Section 6.3 is a linear DUTT. Tree transducer R in Section 6.3 is a linear and complete DUTT.

Ex. 6.4.1.2 States of U_1 are q, q'; $\mathcal{F} = \{f(), a\}$; $\mathcal{F}' = \{g(,), f(), f'(), a\}$; q' is the final state; the set of transduction rules is:

$$\begin{aligned} a &\to q(a) \\ f(q(x)) &\to q(f(x)) \mid q(f'(x)) \mid q'(g(x, x)) \end{aligned}$$

U_1 is a complete, non linear NUTT. We now give the transductions of the ground term $f(f(f(a)))$. For the sake of simplicity, $fffa$ stands for $f(f(f(a)))$. We have:

$$U_1(\{fffa\}) = \{g(ffa, ffa), g(ff'a, ff'a), g(f'fa, f'fa), g(f'f'a, f'f'a)\}.$$

U_1 illustrates an ability of NUTT, that we describe following Gécseg and Steinby:

> *B1- "Nprocess and copy" A NUTT can first process an input subtree nondeterministically and then make copies of the resulting output tree.*

Ex. 6.4.1.3 States of U_2 are q, q', q_f; $\mathcal{F} = \{f(,), g(), a\}$; $\mathcal{F} = \mathcal{F}' \cup \{f'()\}$; q_f is the final state; the set of transduction rules is defined by:

$$\begin{aligned} a &\rightarrow q(a) \mid q'(a) \\ g(q(x)) &\rightarrow q(g(x)) \\ g(q'(x)) &\rightarrow q'(g(x)) \\ f(q'(x), q'(y)) &\rightarrow q'(f(x, y)) \\ f(q'(x), q(y)) &\rightarrow q_f(f'(x)) \end{aligned}$$

U_2 is a non complete NUTT. The tree transformation induced by U_2 is $\{(f(t_1, t_2), f'(t_1)) \mid t_2 = g^m(a) \text{ for some } m\}$. It illustrates an ability of NUTT, that we describe following Gécseg and Steinby:

> *B2- "check and delete" A NUTT can first check regular constraints on input subterms and delete these subterms afterwards.*

Bottom-up tree transducers translate the input trees from leaves to root, so bottom-up tree transducers are also called frontier-to-root transducers. Top-down tree transducers work in opposite direction.

6.4.2 Top-down Tree Transducers

A ***top-down Tree Transducer*** (NDTT) is a tuple $D = (Q, \mathcal{F}, \mathcal{F}', Q_i, \Delta)$ where Q is a set of (unary) states, $\mathcal{F}$ and $\mathcal{F}'$ are finite nonempty sets of input symbols and output symbols, $Q_i \subseteq Q$ is a set of initial states and Δ is a set of transduction rules of the following two types:

$$q(f(x_1, \ldots, x_n)) \rightarrow u[q_1(x_{i_1}), \ldots, q_p(x_{i_p})] \ ,$$

where $f \in \mathcal{F}_n$, $u \in \mathcal{C}^p(\mathcal{F}')$, $q, q_1, \ldots, q_p \in Q$, , $x_{i_1}, \ldots, x_{i_p} \in \mathcal{X}_n$, or

$$q(x) \rightarrow u[q_1(x), \ldots, q_p(x)] \quad (\varepsilon\text{-rule}),$$

where $u \in \mathcal{C}^p(\mathcal{F}')$, $q, q_1, \ldots, q_p \in Q$, $x \in \mathcal{X}$.

As for top-down NFTA, there is no final state, because when a symbol is a leave a (i.e. a constant symbol), transduction rules are of the form $q(a) \rightarrow u$, where u is a ground term. These rules can be considered as "final rules". Let $t, t' \in T(\mathcal{F} \cup \mathcal{F}' \cup Q)$. The move relation $\rightarrow_D$ is defined by:

$$t \xrightarrow[D]{} t' \Leftrightarrow \begin{cases} \exists q(f(x_1,\ldots,x_n)) \to u[q_1(x_{i_1}),\ldots,q_p(x_{i_p})] \in \Delta \\ \exists C \in \mathcal{C}(\mathcal{F} \cup \mathcal{F}' \cup Q) \\ \exists u_1,\ldots,u_n \in T(\mathcal{F}) \\ t = C[q(f(u_1,\ldots,u_n))] \\ t' = C[u[q_1(v_1),\ldots,q_p(v_p)])] \text{ where } v_j = u_k \text{ if } x_{i_j} = x_k \end{cases}$$

This definition includes the case of ε-rule as a particular case. $\rightarrow_D^*$ is the reflexive and transitive closure of $\rightarrow_D$. A transduction of D from a ground term $t \in T(\mathcal{F})$ to a ground term $t' \in T(\mathcal{F}')$ is a sequence of move steps of the form $q(t) \rightarrow_D^* t'$, where q is an initial state. The transformation induced by D is the relation (also denoted by D) defined by:

$$D = \{(t,t') \mid q(t) \xrightarrow[D]{*} t', t \in T(\mathcal{F}), t' \in T(\mathcal{F}'), q \in Q_i\}.$$

The domain of D is the set $\{t \in T(\mathcal{F}) \mid (t,t') \in D\}$. The image of a set of ground terms L by D is the set $D(L) = \{t' \in T(\mathcal{F}') \mid \exists t \in L, (t,t') \in D\}$. ε-free, linear, non-erasing, complete (or non-deleting), deterministic top-down tree transducers are defined as in the bottom-up case.

Example 6.4.2.

Ex. 6.4.2.1 Tree transducers A', T_1, T_2 defined in Section 6.3 are examples of NDTT.

Ex. 6.4.2.2 Let us now define a non-deterministic and non linear NDTT D_1. States of D_1 are q, q'. The set of input symbols is $\mathcal{F} = \{f(), a\}$. The set of output symbols is $\mathcal{F}' = \{g(,), f(), f'(), a\}$. The initial state is q. The set of transduction rules is:

$$\begin{aligned} q(f(x)) &\to g(q'(x), q'(x)) && \text{(copying rule)} \\ q'(f(x)) &\to f(q'(x)) \mid f'(q'(x)) && \text{(non deterministic relabeling)} \\ q'(a) &\to a \end{aligned}$$

D_1 transduces $f(f(f(a)))$ (or briefly $fffa$) into the set of 16 trees:

$$\{g(ffa, ffa), g(ffa, ff'a), g(ffa, f'fa), \ldots, g(f'f'a, f'fa), g(f'f'a, f'f'a)\}.$$

D_1 illustrates a new property.

> *D- "copy and Nprocess" A NDTT can first make copies of an input subtree and then process different copies independently and nondeterministically .*

6.4.3 Structural Properties

In this section, we use tree transducers U_1, U_2 and D_1 of the previous section in order to point out differences between top-down and bottom-up tree transducers.

Theorem 6.4.3 (Comparison Theorem)**.**

1. *There is no top-down tree transducer equivalent to* U_1 *or to* U_2.
2. *There is no bottom-up tree transducer equivalent to* D_1.
3. *Any linear top-down tree transducer is equivalent to a linear bottom-up tree transducer. In the linear complete case, classes of bottom-up and top-down tree transducers are equal.*

It is not hard to verify that neither NUTT nor NDTT are closed under composition. Therefore, comparison of D-property "copy and Nprocess" and U-property "Nprocess and copy" suggests an important question:

> *does alternation of copying and non-determinism induces an infinite hierarchy of transformations?*

The answer is affirmative [Eng78, Eng82], but it was a relatively long-standing open problem. The fact that top-down transducers copy before non-deterministic processes, and bottom-up transducers copy after non-deterministic processes (see Exercise 6.11) suggests too that we get by composition two intricate infinite hierarchies of transformation. The following theorem summarizes results.

Theorem 6.4.4 (Hierarchy theorem)**.** *By composition of NUTT, we get an infinite hierarchy of transformations. Any composition of* n *NUTT can be processed by composition of* $n+1$ *NDTT, and conversely (i.e. any composition of* n *NDTT can be processed by composition of* $n+1$ *NUTT).*

Transducer A' of Section 6.3 shows that it can be useful to consider ε-rules, but usual definitions of tree transducers in literature exclude this case of non determinism. This does not matter, because it is easy to check that all important results of closure or non-closure hold simultaneously for general classes and ε-free classes. Deleting is also a minor phenomenon. Indeed, it gives rise to the "check and delete" property, which is specific to bottom-up transducers, but it does not matter for hierarchy theorem, which remains true if we consider complete transducers.

Section 6.3 suggests that for practical use, non-determinism and non-linearity are rare. Therefore, it is important to note that if we assume linearity or determinism, hierarchy of Theorem 6.4.5 collapses. Following results supply algorithms to compose or simplify transducers.

Theorem 6.4.5 (Composition Theorem)**.**

1. *The class of linear bottom-up transductions is closed under composition.*
2. *The class of deterministic bottom-up transductions is closed under composition.*
3. *Any composition of deterministic top-down transductions is equivalent to a deterministic complete top-down transduction composed with a linear homomorphism.*

The reader should note that bottom-up determinism and top-down determinism are incomparable (see Exercise 6.8).

Recognizable tree languages play a crucial role because derivation tree languages for context-free word grammars are recognizable. Fortunately, we get:

Theorem 6.4.6 (Recognizability Theorem)**.** *The domain of a tree transducer is a recognizable tree language. The image of a recognizable tree language by a linear tree transducer is recognizable.*

6.4.4 Complexity Properties

We present now some decidability and complexity results. As for structural properties, the situation is more complicated than in the word case, especially for top-down tree transducers. Most of problems are untractable in the worst case, but empirically "not so much complex" in real cases, though there is a lack of "algorithmic engineering" to get efficient algorithms. As in the word case, emptiness is decidable, and equivalence is undecidable in the general case but is decidable in the k-valued case (a transducer is k-valued if there is no tree which is transduced in more than k different terms; so a deterministic transducer is a particular case of 1-valued transducer).

Theorem 6.4.7 (Decidability and complexity)**.** *Emptiness of tree transductions is decidable. Equivalence of k-valued tree transducers is decidable.*

Emptiness for bottom-up transducers is essentially the same as emptiness for tree automata and therefore PTIME complete. Emptiness for top-down automata, however, is essentially the same as emptiness for alternating top-down tree automata, giving DEXPTIME completeness for emptiness. The complexity PTIME for testing single-valuedness in the bottom-up case is contained in Seidl [Sei92]. Ramsey theory gives combinatorial properties onto which equivalence tests for k-valued tree transducers [Sei94a].

Theorem 6.4.8 (Equivalence Theorem)**.** *Equivalence of deterministic tree transducers is decidable.*

6.5 Homomorphisms and Tree Transducers

Exercise 6.10 illustrates how decomposition of transducers using homomorphisms can help to get composition results, but we are far from the nice bimorphism theorem of the word case, and in the tree case, there is no illuminating theorem, but many complicated partial statements. Seminal paper of Engelfriet [Eng75] contains a lot of decomposition and composition theorems. Here, we only present the most significant results.

A delabeling is a linear, complete, and symbol-to-symbol tree homomorphism (see Section 1.4). This very special kind of homomorphism changes only the label of the input letter and possibly order of subtrees. Definition of tree bimorphisms is not necessary, it is the same as in the word case. We get the following characterization theorem. We say that a bimorphism is linear, (respectively complete, etc) if the two morphisms are linear, (respectively complete, etc).

Theorem 6.5.1. *The class of bottom-up tree transductions is equivalent to the class of bimorphisms* (Φ, L, Ψ) *where* Φ *is a delabeling.*

Relation defined by (Φ, L, Ψ) *is computed by a transduction which is linear (respectively complete,* ε*-free) if* Ψ *is linear (respectively complete,* ε*-free).*

Remark that Nivat Theorem illuminates the symmetry of word transductions: the inverse relation of a rational transduction is a rational transduction. In the tree case, non-linearity obviously breaks this symmetry, because a tree transducer can copy an input tree and process several copies, but it can never check equality of subtrees of an input tree. If we want to consider symmetric relations, we have two main situations. In the non-linear case, it is easy to prove that composition of two bimorphisms simulates a Turing machine. In the linear and the linear complete cases, we get the following results.

Theorem 6.5.2 (Tree Bimorphisms)**.** .

1. *The class* LCFB *of linear complete* ε*-free tree bimorphisms satisfies* $\mathsf{LCFB} \subset \mathsf{LCFB}^2 = \mathsf{LCFB}^3$*.*
2. *The class* LB *of linear tree bimorphisms satisfies* $\mathsf{LB} \subset \mathsf{LB}^2 \subset \mathsf{LB}^3 \subset \mathsf{LB}^4 = \mathsf{LB}^5$*.*

Proof of $\mathsf{LCFB}^2 = \mathsf{LCFB}^3$ requires many refinements and we omit it.

To prove $\mathsf{LCFB} \subset \mathsf{LCFB}^2$ we use twice the same homomorphism $\Phi(a) = a, \Phi(f(x)) = f(x), \Phi(g(x, y)) = g(x, y), \Phi(h(x, y, z)) = g(x, g(y, z))$.

For any subterms $(t_1, \ldots, t_{2p+2})$, let

$$t = h(t_1, t_2, h(t_3, t_4, h(t_{2i+1}, t_{2i+2}, \ldots, h(t_{2p-1}, t_{2p}, g(t_{2p+1}, t_{2p+2}) \ldots)))$$

and

$$t' = g(t_1, h(t_2, t_3, h(t_4, \ldots, h(t_{2i}, t_{2i+1}, h(t_{2i+2}, t_{2i+3}, \ldots, h(t_{2p}, t_{2p+1}, t_{2p+2}) \ldots))).$$

We get $t' \in (\Phi \circ \Phi^{-1})(t)$. Assume that $\Phi \circ \Phi^{-1}$ can be processed by some $\Psi^{-1} \circ \Psi'$. Consider for simplicity subterms t_i of kind $f^{ni}(a)$. Roughly, if lengths of t_i are different enough, Ψ and Ψ' must be supposed linear complete. Suppose that for some u we have $\Psi(u) = t$ and $\Psi'(u) = t'$, then for any context u' of u, $\Psi(u')$ is a context of t with an odd number of variables, and $\Psi'(u')$ is a context of t' with an even number of variables. That is impossible because homomorphisms are linear complete.

Point 2 is a refinement of point 1 (see Exercise 6.15).

This example shows a stronger fact: the relation cannot be processed by any bimorphism, even non-linear, nor by any bottom-up transducer A direct characterization of these transformations is given in [AD82] by a special class of top-down tree transducers, which are not linear but are "globally" linear, and which are used to prove $\mathsf{LCFB}^2 = \mathsf{LCFB}^3$.

6.6 Exercises

Exercises 6.1 to 6.7 are devoted to the word case, which is out of scope of this book. For this reason, we give precise hints for them.

Exercise 6.1. *The class of rational transductions is closed under rational operations.* Hint: for closure under union, connect a new initial state to initial state with $(\varepsilon, \varepsilon)$-rules (parallel composition). For concatenation, connect by the same way final states of the first transducer to initial states of the second (serial composition). For iteration, connect final states to initial states (loop operation).

Exercise 6.2. *The class of rational transductions is not closed under intersection.* Hint: consider rational transductions $\{(a^n b^p, a^n) \mid n, p \in \mathbb{N}\}$ and $\{(a^n b^p, a^p) \mid n, p \in \mathbb{N}\}$.

Exercise 6.3. *Equivalence of rational transductions is undecidable.* Hint: Associate the transduction $T_P = \{(f(u), g(u)) \mid u \in \Sigma^+\}$ with each instance $P = (f, g)$ of the Post correspondance Problem such that T_P defines $\{(\Phi(m), \Psi(m)) \mid m \in \Sigma^*\}$. Consider *Diff* of example 6.2.2.2. $\mathit{Diff} \neq \mathit{Diff} \cup T_P$ if and only if P satisfies Post property.

Exercise 6.4. *Equivalence of deterministic rational transductions is decidable.* Hint: design a pumping lemma to reduce the problem to a bounded one by suppression of loops (if difference of lengths between two transduced subwords is not bounded, two transducers cannot be equivalent).

Exercise 6.5. *Build a rational transducer equivalent to a bimorphism.* Hint: let $f(q) \to q'(f)$ a transition rule of L. If $\Phi(f) = \varepsilon$, introduce transduction rule $\varepsilon(q) \to q'(\Psi(f))$. If $\Phi(f) = a_0 \ldots a_n$, introduce new states $q_1, \ldots, q_n$ and transduction rules $a_0(q) \to q_1(\varepsilon), \ldots a_i(q_i) \to q_{i+1}(\varepsilon), \ldots a_n(q_n) \to q'(\Psi(f))$.

Exercise 6.6. *Build a bimorphism equivalent to a rational transducer.* Hint: consider the set Δ of transition rules as a new alphabet. We may speak of the first state q and the second state q' in a letter "$f(q) \to q'(m)$". The control language L is the set of words over this alphabet, such that (i) the first state of the first letter is initial (ii) the second state of the last letter is final (iii) in every two consecutive letters of a word, the first state of the second equals the second state of the first. We define Φ and Ψ by $\Phi(f(q) -> q'(m)) = f$ and $\Psi(f(q) -> q'(m)) = m$.

Exercise 6.7. *Homomorphism inversion and applications.* An homomorphism Φ is non-increasing if for every symbol a, $\Phi(a)$ is the empty word or a symbol.

1. For any morphism Φ, find a bimorphism (Φ', L, Ψ) equivalent to Φ^{-1}, with Φ' non-increasing, and such that furthermore Φ' is ε-free if Φ is ε-free. Hint: Φ^{-1} is equivalent to a transducer R (Exercise 6.5), and the output homomorphism Φ' associated to R as in Exercise 6.6 is non-increasing. Furthermore, if Φ is ε-free, R and Φ' are ε-free.
2. Let Φ and Ψ two homomorphisms. If Φ is non-increasing, build a transducer equivalent to $\Psi \circ \Phi^{-1}$ (recall that this notation means that we apply Ψ before Φ^{-1}). Hint and remark: as Φ is non-increasing, Φ^{-1} satisfies the inverse homomorphism property $\Phi^{-1}(MM') = \Phi^{-1}(M)\Phi^{-1}(M')$ (for any pair of words or languages M and M'). This property can be used to do constructions "symbol by symbol". Here, it suffices that the transducer associates $\Phi^{-1}(\Psi(a))$ with a, for every symbol a of the domain of Ψ.
3. Application: prove that classes of regular and context-free languages are closed under bimorphisms (we admit that intersection of a regular language with a regular or context-free language, is respectively regular or context-free).
4. Other application: prove that bimorphisms are closed under composition. Hint: remark that for any application f and set E, $\{(x, f(x)) \mid f(x) \in E\} = \{(x, f(x)) \mid x \in f^{-1}(E)\}$.

Exercise 6.8. We identify words with trees over symbols of arity 1 or 0. Let relations $U = \{(f^n a, f^n a) \mid n \in \mathbb{N}\} \cup \{(f^n b, g^n b) \mid n \in \mathbb{N}\}$ and $D = \{(ff^n a, ff^n a) \mid n \in \mathbb{N}\} \cup \{(gf^n a, gf^n b) \mid n \in \mathbb{N}\}$. Prove that U is a deterministic linear complete bottom-up transduction but not a deterministic top-down transduction. Prove that D is a deterministic linear complete top-down transduction but not a deterministic bottom-up transduction.

Exercise 6.9. Prove point 3 of Comparison Theorem. Hint. Use rule-by-rule techniques as in Exercise 6.10.

Exercise 6.10. Prove Composition Theorem (Th. 6.4.5). Hints: Prove 1 and 2 using composition "rule-by-rule", illustrated as following. States of $A \circ B$ are products of states of A and states of B. Let $f(q(x)) \rightarrow_A q'(g(x, g(x, a)))$ and $g(q_1(x), g(q_2(y), a)) \rightarrow_B q_4(u)$. Subterms substituted to x and y in the composition must be equal, and determinism implies $q_1 = q_2$. Then we build new rule $f((q, q1)(x)) \rightarrow_{A \circ B} (q', q_4)(u)$. For 3, Using ad hoc kinds of "rule-by-rule" constructions, prove $\mathsf{DDTT} \subset \mathsf{DCDTT} \circ \mathsf{LHOM}$ and $\mathsf{LHOM} \circ \mathsf{DCDTT} \subset \mathsf{DCDTT} \circ \mathsf{LHOM}$ (L means linear, C complete, D deterministic - and suffix DTT means top-down tree transducer as usually).

Exercise 6.11. Prove $\mathsf{NDTT} = \mathsf{HOM} \circ \mathsf{NLDTT}$ and $\mathsf{NUTT} = \mathsf{HOM} \circ \mathsf{NLBTT}$. Hint: to prove $\mathsf{NDTT} \subset \mathsf{HOM} \circ \mathsf{NLDTT}$ use a homomorphism H to produce in advance as may copies of subtrees of the input tree as the NDTT may need, ant then simulate it by a linear NDTT.

Exercise 6.12. Use constructions of composition theorem to reduce the number of passes in process of Section 6.3.

Exercise 6.13. Prove recognizability theorem. Hint: as in exercise 6.10, "naive" constructions work.

Exercise 6.14. Prove Theorem 6.5.1. Hint: "naive" constructions work.

Exercise 6.15. Prove point 2 of Theorem 6.5.2. Hint: E denote the class of homomorphisms which are linear and symbol-to-symbol. L, LC, LCF denotes linear, linear complete, linear complete ε-free homomorphisms, respectively. Prove $\mathsf{LCS} = \mathsf{L} \circ \mathsf{E} = \mathsf{E} \circ \mathsf{L}$ and $\mathsf{E}^{-1} \circ \mathsf{L} \subset \mathsf{L} \circ \mathsf{E}^{-1}$. Deduce from these properties and from point 1 of Theorem 6.5.2 that $\mathsf{LB}^4 = \mathsf{E} \circ \mathsf{LCFB}^2 \circ \mathsf{E}^{-1}$. To prove that $\mathsf{LB}^3 \neq \mathsf{LB}^4$, consider $\Psi_1 \circ \Psi_2^{-1} \circ \Phi \circ \Phi^{-1} \circ \Psi_2 \circ \Psi_1^{-1}$, where Φ is the homomorphism used in point 1 of Theorem 6.5.2; Ψ_1 identity on a, $f(x)$, $g(x, y)$, $h(x, y, z)$, $\Psi_1(e(x)) = x$; Ψ_2 identity on a, $f(x)$, $g(x, y)$ and $\Psi_2(c(x, y, z) = b(b(x, y), z)$.

Exercise 6.16. Sketch of proof of $\mathsf{LCFB}^2 = \mathsf{LCFB}^3$ (difficult). Distance $D(x, y, u)$ of two nodes x and y in a tree u is the sum of the lengths of two branches which join x and y to their younger common ancestor in u. $D(x, u)$ denotes the distance of x to the root of u.

Let H the class of deterministic top-down transducers T defined as follows: $q_0, \ldots, q_n$ are states of the transducer, q_0 is the initial state. For every context, consider the result u_i of the run starting from $q_i(u)$. $\exists k, \forall$ context u such that for every variable x of u, $D(x, u) > k$:

- u_0 contains at least an occurrence of each variable of u,
- for any i, u_i contains at least a non variable symbol,
- if two occurrences x' and x'' of a same variable x occur in u_i, $D(x', x", u_i) < k$.

Remark that LCF is included in H and that there is no right hand side of rule with two occurrences of the same variable associated with the same state. Prove that

1. $\mathsf{LCF}^{-1} \subseteq \mathsf{Delabeling}^{-1} \circ H$

2. $H \circ \mathsf{Delabeling}^{-1} \subseteq \mathsf{Delabeling}^{-1} \circ H$
3. $H \subseteq \mathsf{LCFB}^2$
4. Conclude. Compare with Exercise 6.7

Exercise 6.17. Prove that the image of a recognizable tree language by a linear tree transducer is recognizable.

6.7 Bibliographic notes

First of all, let us precise that several surveys have been devoted (at least in part) to tree transducers for 25 years. J.W. Thatcher [Tha73], one of the main pioneer, did the first one in 1973, and F. Gécseg and M. Steinby the last one in 1996 [GS96]. Transducers are formally studied too in the book of F. Gécseg and M. Steinby [GS84] and in the survey of J.-C. Raoult [Rao92]. Survey of M. Dauchet and S. Tison [DT92] develops links with homomorphisms.

In section 6.2, some examples are inspired by the old survey of Thatcher, because seminal motivation remain, namely modelization of compilers or, more generally, of syntax directed transformations as interfacing softwares, which are always up to date. Among main precursors, we can distinguish Thatcher [Tha73], W.S. Brainerd [Bra69], A. Aho, J.D. Ullman [AU71], M. A. Arbib, E. G. Manes [AM78]. First approaches where very linked to practice of compilation, and in some way, present tree transducers are evolutions of generalized syntax directed translations (B.S. Backer [Bak78] for example), which translate trees into strings. But crucial role of tree structure have increased later.

Many generalizations have been introduced, for example generalized finite state transformations which generalize both the top-down and the bottom-up tree transducers (J. Engelfriet [Eng77]); modular tree transducers (H. Vogler [EV91]); synchronized tree automata (K. Salomaa [Sal94]); alternating tree automata (G.Slutzki [Slu85]); deterministic top-down tree transducers with iterated look-ahead (G. Slutzki, S. Vàgvölgyi [SV95]). Ground tree transducers GTT are studied in Chapter 3 of this book. The first and the most natural generalization was introduction of top-down tree transducers with look-ahead. We have seen that "check and delete" property is specific to bottom-up tree transducers, and that missing of this property in the non-complete top-down case induces non closure under composition, even in the linear case (see Composition Theorem). Top-down transducers with regular look-ahead are able to recognize before the application of a rule at a node of an input tree whether the subtree at a son of this node belongs to a given recognizable tree language. This definition remains simple and gives to top-down transducers a property equivalent to "check and delete".

Contribution of Engelfriet to the theory of tree transducers is important, especially for composition, decomposition and hierarchy main results ([Eng75, Eng78, Eng82]).

We did not many discuss complexity and decidability in this chapter, because the situation is classical. Since many problems are undecidable in the word case, they are obviously undecidable in the tree case. Equivalence decidability holds as in the word case for deterministic or finite-valued tree transducers (Z. Zachar [Zac79], Z. Esik [Esi83], H. Seidl [Sei92, Sei94a]).

Chapter 7

Alternating Tree Automata

7.1 Introduction

Complementation of non-deterministic tree (or word) automata requires a determinization step. This is due to an asymmetry in the definition. Two transition rules with the same left hand side can be seen as a single rule with a disjunctive right side. A run of the automaton on a given tree has to choose some member of the disjunction. Basically, determinization gathers the disjuncts in a single state.

Alternating automata restore some symmetry, allowing both disjunctions and conjunctions in the right hand sides. Then complementation is much easier: it is sufficient to exchange the conjunctions and the disjunction signs, as well as final and non-final states. In particular, nothing similar to determinization is needed.

This nice formalism is more concise. The counterpart is that decision problems are more complex, as we will see in Section 7.5.

There are other nice features: for instance, if we see a tree automaton as a finite set of monadic Horn clauses, then moving from non-deterministic to alternating tree automata consists in removing a very simple assumption on the clauses. This is explained in Section 7.6. In the same vein, removing another simple assumption yields *two-way* alternating tree automata, a more powerful device (yet not more expressive), as described in Section 7.6.3.

Finally, we also show in Section 7.2.3 that, as far as emptiness is concerned, tree automata correspond to alternating word automata on a single-letter alphabet, which shows the relationship between computations (runs) of a word alternating automaton and computations of a tree automaton.

7.2 Definitions and Examples

7.2.1 Alternating Word Automata

Let us start first with alternating *word automata*.

If Q is a finite set of states, $\mathcal{B}^+(Q)$ is the set of positive propositional formulas over the set of propositional variables Q. For instance, $q_1 \wedge (q_2 \vee q_3) \wedge (q_2 \vee q_4) \in \mathcal{B}^+(\{q_1, q_2, q_3, q_4\})$.

Alternating word automata are defined as deterministic word automata, except that the transition function is a mapping from $Q \times A$ to $\mathcal{B}^+(Q)$ instead of being a mapping from $Q \times A$ to Q. We assume a subset Q_0 of Q of *initial states* and a subset Q_f of Q of *final states*.

Example 7.2.1. Assume that the alphabet is $\{0, 1\}$ and the set of states is $\{q_0, q_1, q_2, q_3, q_4, q'_1, q'_2\}$, $Q_0 = \{q_0\}$, $Q_f = \{q_0, q_1, q_2, q_3, q_4\}$ and the transitions are:

$$\begin{array}{lcllcl}
q_0 0 & \to & (q_0 \wedge q_1) \vee q'_1 & q_0 1 & \to & q_0 \\
q_1 0 & \to & q_2 & q_1 1 & \to & \mathrm{true} \\
q_2 0 & \to & q_3 & q_2 1 & \to & q_3 \\
q_3 0 & \to & q_4 & q_3 1 & \to & q_4 \\
q_4 0 & \to & \mathrm{true} & q_4 1 & \to & \mathrm{true} \\
q'_1 0 & \to & q'_1 & q'_1 1 & \to & q'_2 \\
q'_2 0 & \to & q'_2 & q'_2 1 & \to & q'_1
\end{array}$$

A *run* of an alternating word automaton $\mathcal{A}$ on a word w is a finite tree ρ labeled with $Q \times \mathbb{N}$ such that:

- The root of ρ is labeled by some pair $(q, 0)$.
- If $\rho(p) = (q, i)$ and i is strictly smaller than the length of w, $w(i + 1) = a$, $\delta(q, a) = \phi$, then there is a set $S = \{q_1, \ldots, q_n\}$ of states such that $S \models \phi$, positions $p1, \ldots, pn$ are the successor positions of p in ρ and $\rho(pj) = (q_j, i + 1)$ for every $j = 1, ...n$.

The notion of satisfaction used here is the usual one in propositional calculus: the set S is the set of propositions assigned to true, while the propositions not belonging to S are assumed to be assigned to false. Therefore, we have the following:

- there is no run on w such that $w(i + 1) = a$ for some i, $\rho(p) = (q, i)$ and $\delta(q, i) = \mathrm{false}$
- if $\delta(q, w(i + 1)) = \mathrm{true}$ and $\rho(p) = (q, i)$, then p can be a leaf node, in which case it is called a *success node*.
- All leaf nodes are either success nodes as above or labeled with some (q, n) such that n is the length of w.

A run of an alternating automaton is *successful* on w if and only if

- all leaf nodes are either success nodes or labeled with some (q, n), where n is the length of w, such that $q \in Q_f$.
- the root node $\rho(\epsilon) = (q_0, 0)$ with $q_0 \in Q_0$.

Example 7.2.2. Let us come back to Example 7.2.1. We show on Figure 7.1 two runs on the word 00101, one of which is successful.

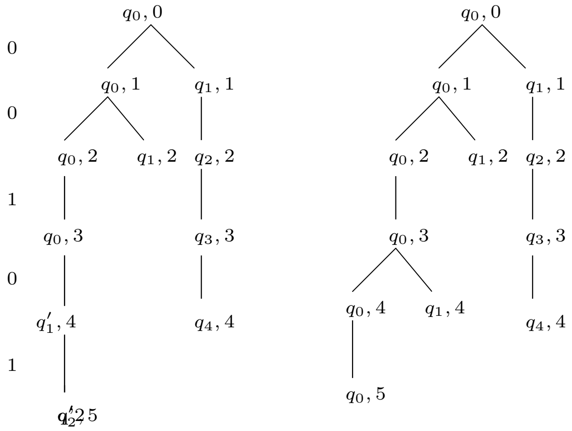


Figure 7.1: Two runs on the word 00101 of the automaton defined in Example 7.2.1. The right one is successful.

Note that non-deterministic automata are the particular case of alternating automata in which only disjunctions (no conjunctions) occur in the transition relation. In such a case, if there is a succesful run on w, then there is also a successful run, which is a string.

Note also that, in the definition of a run, we can always choose the set S to be a minimal satisfaction set: if there is a successful run of the automaton, then there is a successful one in which we always choose a minimal set S of states.

7.2.2 Alternating Tree Automata

Now, let us switch to alternating tree automata: the definitions are simple adaptations of the previous ones.

Definition 7.2.3. *An* alternating tree automaton *over* $\mathcal{F}$ *is a tuple* $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ *where* Q *is a set of states,* $I \subseteq Q$ *is a set of initial states and* Δ *is a mapping from* $Q \times \mathcal{F}$ *to* $\mathcal{B}^+(Q \times \mathbb{N})$ *such that* $\Delta(q, f) \in \mathcal{B}^+(Q \times \{1, \ldots, Arity(f)\})$ *where* $Arity(f)$ *is the arity of* f.

Note that this definition corresponds to a *top-down* automaton, which is more convenient in the alternating case.

Definition 7.2.4. *Given a term* $t \in T(\mathcal{F})$ *and an alternating tree automaton* $\mathcal{A}$ *on* $\mathcal{F}$, *a* run *of* $\mathcal{A}$ *on* t *is a tree* ρ *on* $Q \times \mathbb{N}^*$ *such that* $\rho(\varepsilon) = (q, \varepsilon)$ *for some state* q *and*

> *if* $\rho(\pi) = (q, p)$, $t(p) = f$ *and* $\delta(q, f) = \phi$, *then there is a subset* $S = \{(q_1, i_1), \ldots, (q_n, i_n)\}$ *of* $Q \times \{1, \ldots, Arity(f)\}$ *such that* $S \models \phi$, *the successor positions of* π *in* ρ *are* $\{\pi 1, \ldots, \pi n\}$ *and* $\rho(\pi \cdot j) = (q_j, p \cdot i_j)$ *for every* $j = 1..n$.

A run ρ *is* successful *if* $\rho(\varepsilon) = (q, \varepsilon)$ *for some initial state* $q \in I$.

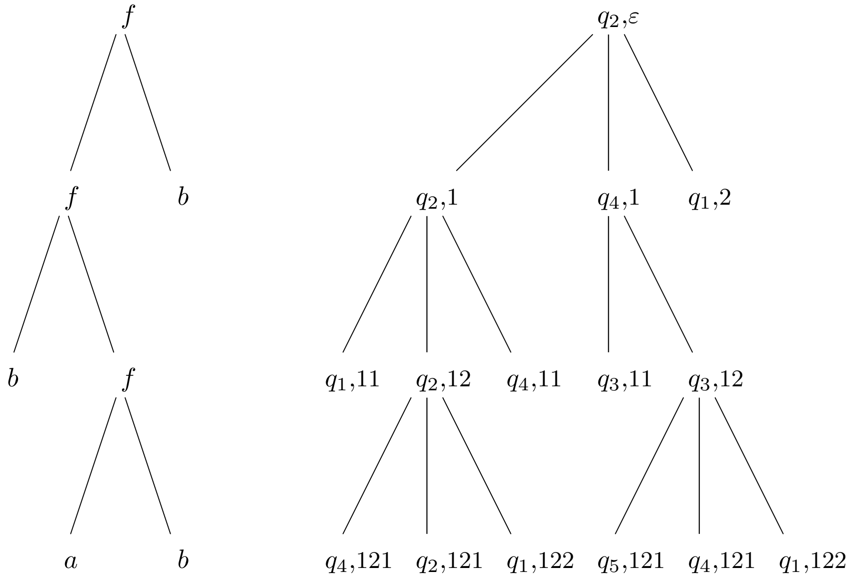


Figure 7.2: A run of an alternating tree automaton

Note that (completely specified) non-deterministic top-down tree automata are the particular case of alternating tree automata. For a set of non-deterministic rules $q(f(x_1, \ldots, x_n)) \rightarrow f(q_1(x_1), \ldots, q_n(x_n))$, $Delta(q, f)$ is defined by:

$$\Delta(q, f) = \bigvee_{(q_1,\ldots,q_n)\in S} \bigwedge_{i=1}^{Arity(f)} (q_i, i)$$

Example 7.2.5. Consider the automaton on the alphabet $\{f(,), a, b\}$ whose transition relation is defined by:

Δ	f	a	b
q_2	$[((q_1, 1) \wedge (q_2, 2)) \vee ((q_1, 2) \wedge (q_2, 1))] \wedge (q_4, 1)$	true	false
q_1	$((q_2, 1) \wedge (q_2, 2)) \vee ((q_1, 2) \wedge (q_1, 1))$	false	true
q_4	$((q_3, 1) \wedge (q_3, 2)) \vee ((q_4, 1) \wedge (q_4, 2))$	true	true
q_3	$((q_3, 1) \wedge (q_2, 2)) \vee ((q_4, 1) \wedge (q_1, 2)) \wedge (q_5, 1))$	false	true
q_5	false	true	false

Assume $I = \{q_2\}$. A run of the automaton on the term $t = f(f(b, f(a, b)), b)$. is depicted on Figure 7.2.

In the case of a non-deterministic top-down tree automaton, the different notions of a run coincide as, in such a case, the run obtained from Definition 7.2.4

on a tree t is a tree whose set of positions is the set of positions of t, possibly changing the ordering of sons.

Words over an alphabet A can be seen as trees over the set of unary function symbols A and an additional constant #. For convenience, we read the words from right to left. For instance, *aaba* is translate into the tree $a(b(a(a(\#))))$. Then an alternating word automaton $\mathcal{A}$ can be seen as an alternating tree automaton whose initial states are the final states of $\mathcal{A}$, the transitions are the same and there is additional rules $\delta(q^0, \#) = \text{true}$ for the initial state q^0 of $\mathcal{A}$ and $\delta(q, \#) = \text{false}$ for other states.

7.2.3 Tree Automata versus Alternating Word Automata

It is interesting to remark that, guessing the input tree, it is possible to reduce the emptiness problem for (non-deterministic, bottom-up) tree automata to the emptiness problem for an alternating word automaton on a single letter alphabet: assume that $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ is a non-deterministic tree automaton, then construct the alternating word automaton on a one letter alphabet $\{a\}$ as follows: the states are $Q \times \mathcal{F}$, the initial states are $Q_f \times \mathcal{F}$ and the transition rules:

$$\delta((q, f), a) = \bigvee_{f(q_1,\dots,q_n)\to q\in\Delta} \bigwedge_{i=1}^{n} \bigvee_{f_j\in\mathcal{F}} ((q_i, f_j), i)$$

Conversely, it is also possible to reduce the emptiness problem for an alternating word automaton over a one letter alphabet $\{a\}$ to the emptiness problem of non-deterministic tree automata, introducing a new function symbol for each conjunction; assume the formulas in disjunctive normal form (this can be assumed w.l.o.g, see Exercise 7.1), then replace each transition $\delta(q, a) = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{k_i} (q_{i,j}, i)$ with $f_i(q_{i,1}, \dots, q_{i,k_i}) \to q$.

7.3 Closure Properties

One nice feature of alternating automata is that it is very easy to perform the Boolean operations (for short, we confuse here the automaton and the language recognized by the automaton). First, we show that we can consider automata with only one initial state, without loss of generality.

Lemma 7.3.1. *Given an alternating tree automaton $\mathcal{A}$, we can compute in linear time an automaton $\mathcal{A}'$ with only one initial state and which accepts the same language as $\mathcal{A}$.*

Proof. Add one state q^0 to $\mathcal{A}$, which will become the only initial state, and the transitions:

$$\delta(q^0, f) = \bigvee_{q\in I} \delta(q, f)$$

□

Proposition 7.3.2. *Union, intersection and complement of alternating tree automata can be performed in linear time.*

Proof. We consider w.l.o.g. automata with only one initial state. Given $\mathcal{A}_1$ and $\mathcal{A}_2$, with a disjoint set of states, we compute an automaton $\mathcal{A}$ whose states are those of $\mathcal{A}_1$ and $\mathcal{A}_2$ and one additional state q^0. Transitions are those of $\mathcal{A}_1$ and $\mathcal{A}_2$ plus the additional transitions for the union:

$$\delta(q^0, f) = \delta_1(q_1^0, f) \vee \delta_2(q_2^0, f)$$

where q_1^0, q_2^0 are the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively. For the intersection, we add instead the transitions:

$$\delta(q^0, f) = \delta_1(q_1^0, f) \wedge \delta_2(q_2^0, f)$$

Concerning the complement, we simply exchange $\wedge$ and $\vee$ (resp. true and false) in the transitions. The resulting automaton $\widetilde{\mathcal{A}}$ will be called the *dual automaton* in what follows.

The proof that these constructions are correct for union and intersection are left to the reader. Let us only consider here the complement.

We prove, by induction on the size of t that, for every state q, t is accepted either by $\mathcal{A}$ or $\widetilde{\mathcal{A}}$ in state q and not by both automata.

If t is a constant a, then $\delta(q, a)$ is either true or false. If $\delta(q, a) =$ true, then $\widetilde{\delta}(q, a) =$ false and t is accepted by $\mathcal{A}$ and not by $\widetilde{\mathcal{A}}$. The other case is symmetric.

Assume now that $t = f(t_1, \ldots, t_n)$ and $\delta(q, f) = \phi$. Let S be the set of pairs (q_j, i_j) such that t_{i_j} is accepted from state q_j by $\mathcal{A}$. t is accepted by $\mathcal{A}$, iff $S \models \phi$. Let $\widetilde{S}$ be the complement of S in $Q \times [1..n]$. By induction hypothesis, $(q_j, i) \in \widetilde{S}$ iff t_i is accepted in state q_j by $\widetilde{\mathcal{A}}$.

We show that $\widetilde{S} \models \widetilde{\phi}$ iff $S \not\models \phi$. ($\widetilde{\phi}$ is the dual formula, obtained by exchanging $\wedge$ and $\vee$ on one hand and true and false on the other hand in ϕ). We show this by induction on the size of ϕ: if ϕ is true (resp. false), then $S \models \phi$ and $\widetilde{S} \not\models \widetilde{\phi}$ (resp. $\widetilde{S} = \emptyset$) and the result is proved. Now, let, ϕ be, e.g., $\phi_1 \wedge \phi_2$. $S \not\models \phi$ iff either $S \not\models \phi_1$ or $S \not\models \phi_2$, which, by induction hypothesis, is equivalent to $\widetilde{S} \models \widetilde{\phi_1}$ or $\widetilde{S} \models \widetilde{\phi_2}$. By construction, this is equivalent to $\widetilde{S} \models \widetilde{\phi}$. The case $\phi = \phi_1 \vee \phi_2$ is similar.

Now t is accepted in state q by $\mathcal{A}$ iff $S \models \phi$ iff $\widetilde{S} \not\models \widetilde{\phi}$ iff t not accepted in state q by $\widetilde{\mathcal{A}}$. □

7.4 From Alternating to Deterministic Automata

The expressive power of alternating automata is exactly the same as finite (bottom-up) tree automata.

Theorem 7.4.1. *If $\mathcal{A}$ is an alternating tree automaton, then there is a finite deterministic bottom-up tree automaton $\mathcal{A}'$ which accepts the same language. $\mathcal{A}'$ can be computed from $\mathcal{A}$ in deterministic exponential time.*

Proof. Assume $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$, then $\mathcal{A}' = (2^Q, \mathcal{F}, Q_f, \delta)$ where $Q_f = \{S \in 2^Q \mid S \cap I \neq \emptyset\}$ and δ is defined as follows:

$$f(S_1, \ldots, S_n) \rightarrow \{q \in Q \mid S_1 \times \{1\} \cup \ldots \cup S_n \times \{n\} \models \Delta(q, f)\}$$

A term t is accepted by $\mathcal{A}'$ in state S iff t is accepted by $\mathcal{A}$ in all states $q \in S$. This is proved by induction on the size of t: if t is a constant, then t is accepted in all states q such that $\Delta(q, t) = \text{true}$. Now, if $t = f(t_1, \ldots, t_n)$ we let $S_1, \ldots, S_n$ are the set of states in which $t_1, \ldots, t_n$ are respectively accepted by $\mathcal{A}$. t is accepted by $\mathcal{A}$ in a state q iff there is $S_0 \subseteq Q \times \{1, \ldots, n\}$ such that $S_0 \models \Delta(q, f)$ and, for every pair $(q_i, j) \in S_0$, t_j is accepted in q_i. In other words, t is accepted by $\mathcal{A}$ in state q iff there is an $S_0 \subseteq S_1 \times \{1\} \cup \ldots \cup S_n \times \{n\}$ such that $S_0 \models \Delta(q, f)$, which is in turn equivalent to $S_1 \times \{1\} \cup \ldots \cup S_n \times \{n\} \models \Delta(q, f)$. We conclude by an application of the induction hypothesis. ☐

Unfortunately the exponential blow-up is unavoidable, as a consequence of Proposition 7.3.2 and Theorems 1.7.7 and 1.7.4.

7.5 Decision Problems and Complexity Issues

Theorem 7.5.1. *The emptiness problem and the universality problem for alternating tree automata are DEXPTIME-complete.*

Proof. The DEXPTIME membership is a consequence of Theorems 1.7.4 and 7.4.1.

The DEXPTIME-hardness is a consequence of Proposition 7.3.2 and Theorem 1.7.7. ☐

The membership problem (given t and $\mathcal{A}$, is t accepted by $\mathcal{A}$?) can be decided in polynomial time. This is left as an exercise.

7.6 Horn Logic, Set Constraints and Alternating Automata

7.6.1 The Clausal Formalism

Viewing every state q as a unary predicate symbol P_q, tree automata can be translated into Horn clauses in such a way that the language recognized in state q is exactly the interpretation of P_q in the least Herbrand model of the set of clauses.

There are several advantages of this point of view:

- Since the logical setting is declarative, we don't have to distinguish between top-down and bottom-up automata. In particular, we have a definition of bottom-up alternating automata for free.

- Alternation can be expressed in a simple way, as well as push and pop operations, as described in the next section.

- There is no need to define a run (which would correspond to a proof in the logical setting)

- Several decision properties can be translated into decidability problems for such clauses. Typically, since all clauses belong to the *monadic fragment*, there are decision procedures e.g. relying on ordered resolution strategies.

There are also weaknesses: complexity issues are harder to study in this setting. Many constructive proofs, and complexity results have been obtained with tree automata techniques.

Tree automata can be translated into Horn clauses. With a tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ is associated the following set of Horn clauses:

$$P_q(f(x_1, \dots, x_n)) \leftarrow P_{q_1}(x_1), \dots, P_{q_n}(x_n)$$

if $f(q_1, \dots, q_n) \rightarrow q \in \Delta$. The language accepted by the automaton is the union of interpretations of P_q, for $q \in Q_f$, in the least Herbrand model of clauses.

Also, alternating tree automata can be translated into Horn clauses. Alternation can be expressed by variable sharing in the body of the clause. Consider an alternating tree automaton $(Q, \mathcal{F}, I, \Delta)$. Assume that the transitions are in disjunctive normal form (see Exercise 7.1). With a transition $\Delta(q, f) = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{k_i} (q_j, i_j)$ is associated the clauses

$$P_q(f(x_1, \dots, x_n)) \leftarrow \bigwedge_{j=1}^{k_i} P_{q_j}(x_{i_j})$$

We can also add ϵ-transitions, by allowing clauses

$$P(x) \leftarrow Q(x)$$

In such a setting, automata with equality constraints between brothers, which are studied in Section 4.3, are simply an extension of the above class of Horn clauses, in which we allow repeated variables in the head of the clause.

Allowing variable repetition in an arbitrary way, we get alternating automata with contraints between brothers, a class of automata for which emptiness is decidable in deterministic exponential time. (It is expressible in Löwenheim's class with equality, also called sometimes the *monadic class*).

Still, for tight complexity bounds, for closure properties (typically by complementation) of automata with equality tests between brothers, we refer to Section 4.3. Note that it is not easy to derive the complexity results obtained with tree automata techniques in a logical framework.

7.6.2 The Set Constraints Formalism

We introduced and studied general set constraints in Chapter 5. Set constraints and, more precisely, *definite set constraints* provide with an alternative description of tree automata.

Definite set constraints are conjunctions of inclusions

$$e \subseteq t$$

where e is a set expression built using function application, intersection and variables and t is a term set expression, constructed using function application and variables only.

Given an assignment σ of variables to subsets of $T(\mathcal{F})$, we can interpret the set expressions as follows:

$$\begin{aligned} [\![f(e_1, \dots, e_n)]\!]_\sigma &\stackrel{\text{def}}{=} \{f(t_1, \dots, t_n) \mid t_i \in [\![e_i]\!]_\sigma\} \\ [\![e_1 \cap e_2]\!]_\sigma &\stackrel{\text{def}}{=} [\![e_1]\!]_\sigma \cap [\![e_2]\!]_\sigma \\ [\![X]\!]_\sigma &\stackrel{\text{def}}{=} X\sigma \end{aligned}$$

Then σ is a solution of a set constraint if inclusions hold for the corresponding interpretation of expressions.

When we restrict the left members of inclusions to variables, we get another formalism for alternating tree automata: such set constraints have always a least solution, which is accepted by an alternating tree automaton. More precisely, we can use the following translation from the alternating automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$: assume again that the transitions are in disjunctive normal form (see Exercise 7.1) and construct, the inclusion constraints

$$f(X_{1,f,q,d}, \ldots, X_{n,f,q,d}) \subseteq X_q$$

$$\bigcap_{(q',j) \in d} X_{q'} \subseteq X_{j,f,q,d}$$

for every $(q, f) \in Q \times \mathcal{F}$ and d a disjunct of $\Delta(q, f)$. (An intersection over an empty set has to be understood as the set of all trees).

Then, the language recognized by the alternating tree automaton is the union, for $q \in I$, of $X_q\sigma$ where σ is the least solution of the constraint.

Actually, we are constructing the constraint in exactly the same way as we constructed the clauses in the previous section. When there is no alternation, we get an alternative definition of non-deterministic automata, which corresponds to the algebraic characterization of Chapter 2.

Conversely, if all right members of the definite set constraint are variables, it is not difficult to construct an alternating tree automaton which accepts the least solution of the constraint (see Exercise 7.4).

7.6.3 Two Way Alternating Tree Automata

Definite set constraints look more expressive than alternating tree automata, because inclusions

$$X \subseteq f(Y, Z)$$

cannot be directly translated into automata rules.

We define here *two-way tree automata* which will easily correspond to definite set constraints on one hand and allow to simulate, e.g., the behavior of standard pushdown word automata.

It is convenient here to use the clausal formalism in order to define such automata. A clause

$$P(u) \leftarrow P_1(x_1), \ldots, P_n(x_n)$$

where u is a linear, non-variable term and $x_1, \ldots, x_n$ are (not necessarily distinct) variables occurring in u, is called a *push clause*. A clause

$$P(x) \leftarrow Q(t)$$

where x is a variable and t is a linear term, is called a *pop clause*. A clause

$$P(x) \leftarrow P_1(x), \ldots, P_n(x)$$

is called an *alternating clause* (or an *intersection clause*).

Definition 7.6.1. *An* alternating two-way tree automaton *is a tuple* $(Q, Q_f, \mathcal{F}, C)$ *where* Q *is a finite set of unary function symbols,* Q_f *is a subset of* Q *and* C *is a finite set of clauses each of which is a push clause, a pop clause or an alternating clause.*

Such an automaton *accepts* a tree t if t belongs to the interpretation of some $P \in Q_f$ in the least Herbrand model of the clauses.

Example 7.6.2. Consider the following alternating two-way automaton on the alphabet $\mathcal{F} = \{a, f(,)\}$:

$$\begin{array}{lrcl}
1. & P_1(f(f(x_1,x_2),x_3)) & \leftarrow & P_2(x_1), P_2(x_2), P_2(x_3) \\
2. & P_2(a) & & \\
3. & P_1(f(a,x)) & \leftarrow & P_2(x) \\
4. & P_3(f(x,y)) & \leftarrow & P_1(x), P_2(y) \\
5. & P_4(x) & \leftarrow & P_3(x), P_1(x) \\
6. & P_2(x) & \leftarrow & P_4(f(x,y)) \\
7. & P_1(y) & \leftarrow & P_4(f(x,y))
\end{array}$$

The clauses 1,2,3,4 are push clauses. Clause 5 is an alternating clause and clauses 6,7 are pop clauses.

If we compute the least Herbrand model, we successively get for the five first steps:

step	1	2	3	4	5
P_1		$f(a,a), f(f(a,a),a)$			a
P_2	a				$f(a,a)$
P_3			$f(f(a,a),a), f(f(f(a,a),a),a)$		
P_4				$f(f(a,a),a)$	

These automata are often convenient in expressing some problems (see the exercises and bibliographic notes). However they do not increase the expressive power of (alternating) tree automata:

Theorem 7.6.3. *For every alternating two-way tree automaton, it is possible to compute in deterministic exponential time a tree automaton which accepts the same language.*

We do not prove the result here (see the bibliographic notes instead). A simple way to compute the equivalent tree automaton is as follows: first flatten the clauses, introducing new predicate symbols. Then saturate the set of clauses, using ordered resolution (w.r.t. subterm ordering) and keeping only non-subsumed clauses. The saturation process terminates in exponential time. The desired automaton is obtained by simply keeping only the push clauses of this resulting set of clauses.

Example 7.6.4. Let us come back to Example 7.6.2 and show how we get an equivalent finite tree automaton.

First flatten the clauses: Clause 1 becomes

$$\begin{array}{lrcl}
1. & P_1(f(x,y)) & \leftarrow & P_5(x), P_2(y) \\
8. & P_5(f(x,y)) & \leftarrow & P_2(x), P_2(y)
\end{array}$$

Now we start applying resolution;

$$\begin{array}{llrcl}
\text{From } 4+5: & 9. & P_4(f(x,y)) & \leftarrow & P_1(x), P_2(y), P_1(f(x,y)) \\
\text{Form } 9+6: & 10. & P_2(x) & \leftarrow & P_1(x), P_2(y) \\
\text{From } 10+2: & 11. & P_2(x) & \leftarrow & P_1(x)
\end{array}$$

Clause 11 subsumes 10, which is deleted.

$$\begin{array}{lllll}
\text{From } 9+7\text{:} & 12. & P_1(y) & \leftarrow & P_1(x), P_2(y) \\
\text{From } 12+1\text{:} & 13. & P_1(y) & \leftarrow & P_2(y), P_5(x), P_2(z) \\
\text{From } 13+8\text{:} & 14. & P_1(y) & \leftarrow & P_2(y), P_2(x_1), P_2(x_2), P_2(z)
\end{array}$$

Clause 14. can be simplified and, by superposition with 2. we get

$$\text{From } 14+2\text{:} \quad 15. \quad P_1(y) \quad \leftarrow \quad P_2(y)$$

At this stage, from 11. and 15. we have $P_1(x) \leftrightarrow P_2(x)$, hence, for simplicity, we will only consider P_1, replacing every occurrence of P_2 with P_1.

$$\begin{array}{lllll}
\text{From } 1+5\text{:} & 16. & P_4(f(x,y)) & \leftarrow & P_3(f(x,y)), P_5(x), P_1(y) \\
\text{From } 1+9\text{:} & 17. & P_4(f(x,y)) & \leftarrow & P_1(x), P_1(y), P_5(x) \\
\text{From } 2+5\text{:} & 18. & P_4(a) & \leftarrow & P_3(a) \\
\text{From } 3+5\text{:} & 19. & P_4(f(a,x)) & \leftarrow & P_3(f(a,x)), P_1(x) \\
\text{From } 3+9\text{:} & 20. & P_4(f(a,x)) & \leftarrow & P_1(x), P_1(a) \\
\text{From } 2+20\text{:} & 21. & P_4(f(a,x)) & \leftarrow & P_1(x)
\end{array}$$

Clause 21. subsumes both 20 and 19. These two clauses are deleted.

$$\begin{array}{lllll}
\text{From } 5+6\text{:} & 22. & P_1(x) & \leftarrow & P_3(f(x,y)), P_1(f(x,y)) \\
\text{From } 5+7\text{:} & 23. & P_1(y) & \leftarrow & P_3(f(x,y)), P_1(f(x,y)) \\
\text{From } 16+6\text{:} & 24. & P_1(x) & \leftarrow & P_3(f(x,y)), P_5(x), P_1(y) \\
\text{From } 23+1\text{:} & 25. & P_1(y) & \leftarrow & P_3(f(x,y)), P_5(x), P_1(y)
\end{array}$$

Now every new inference yields a redundant clause and the saturation terminates, yielding the automaton:

$$\begin{array}{llll}
1. & P_1(f(x,y)) & \leftarrow & P_5(x), P_2(y) \\
2. & P_1(a) & & \\
3. & P_1(f(a,x)) & \leftarrow & P_1(x) \\
4. & P_3(f(x,y)) & \leftarrow & P_1(x), P_1(y) \\
8. & P_5(f(x,y)) & \leftarrow & P_1(x), P_1(y) \\
11. & P_1(x) & \leftarrow & P_1(y) \\
15. & P_2(x) & \leftarrow & P_1(x) \\
21. & P_4(f(a,x)) & \leftarrow & P_1(x)
\end{array}$$

Of course, this automaton can be simplified: P_1 and P_2 accept all terms in $T(\mathcal{F})$.

It follows from Theorems 7.6.3, 7.5.1 and 1.7.4 that the emptiness problem (resp. universality problems) are DEXPTIME-complete for two-way alternating automata.

7.6.4 Two Way Automata and Definite Set Constraints

There is a simple reduction of two-way automata to definite set constraints:

A push clause $P(f(x_1, \ldots, x_n)) \leftarrow P_1(x_{i_1}), \ldots, P_n(x_{i_n})$ corresponds to an inclusion constraint

$$f(e_1, \ldots, e_n) \subseteq X_P$$

where each e_j is the intersection, for $i_k = j$ of the variables X_{P_k}. A (conditional) pop clause $P(x_i) \leftarrow Q(f(x_1, \ldots, x_n)), P_1(x_1), \ldots P_k(x_k)$ corresponds to

$$f(e_1, \ldots, e_n) \cap X_Q \subseteq f(\top, \ldots, X_P, \top, \ldots)$$

where, again, each e_j is the intersection, for $i_k = j$ of the variables X_{P_k} and $\top$ is a variable containing all term expressions. Intersection clauses $P(x) \leftarrow Q(x), R(x)$ correspond to constraints

$$X_Q \cap X_R \subseteq X_P$$

Conversely, we can translate the definite set constraints into two-way automata, with additional restrictions on some states. We cannot do better since a definite set constraint could be unsatisfiable.

Introducing auxiliary variables, we only have to consider constraints:

1. $f(X_1, \ldots, X_n) \subseteq X$,
2. $X_1 \cap \ldots \cap X_n \subseteq X$,
3. $X \subseteq f(X_1, \ldots, X_n)$.

The first constraints are translated to push clauses, the second kind of constraints is translated to intersection clauses. Consider the last constraints. It can be translated into the pop clauses:

$$P_{X_i}(x_i) \leftarrow P_X(f(x_1, \ldots, x_n))$$

with the provision that all terms in P_X are headed with f.

Then the procedure which solves definite set constraints is essentially the same as the one we sketched for the proof of Theorem 7.6.3, except that we have to add unit negative clauses which may yield failure rules

---

Example 7.6.5. Consider the definite set constraint

$$f(X, Y) \cap X \subseteq f(Y, X), \;\; f(a, Y) \subseteq X, \;\; a \subseteq Y, \;\; f(f(Y, Y), Y) \subseteq X$$

Starting from this constraint, we get the clauses of Example 7.6.2, with the additional restriction

$$26. \;\; \neg P_4(a)$$

since every term accepted in P_4 has to be headed with f.

If we saturate this constraint as in Example 7.6.4, we get the same clauses, of course, but also negative clauses resulting from the new negative clause:

$$\text{From } 26 + 18 \;\; 27. \;\; \neg P_3(a)$$

And that is all: the constraint is satisfiable, with a minimal solution described by the automaton resulting from the computation of Example 7.6.4.

$$\text{Pairing} \quad \frac{u \quad v}{< u, v >} \qquad\qquad \text{Encryption} \quad \frac{u \quad v}{\{u\}_v}$$

$$\text{Unpairing L} \quad \frac{< u, v >}{u} \qquad\qquad \text{Unpairing R} \quad \frac{< u, v >}{v}$$

$$\text{Decryption} \quad \frac{\{u\}_v \quad v}{u}$$

Figure 7.3: The Dolev-Yao intruder capabilities

7.6.5 Two Way Automata and Pushdown Automata

Two-way automata, though related to pushdown automata, are quite different. In fact, for every pushdown automaton, it is easy to construct a two-way automaton which accepts the possible contents of the stack (see Exercise 7.5). However, two-way tree (resp. word) automata have the same expressive power as standard tree (resp. word) automata: they only accept regular languages, while pushdown automata accept context-free languages, which strictly contain regular languages.

Note still that, as a corollary of Theorem 7.6.3, the language of possible stack contents in a pushdown automaton is regular.

7.7 An (other) example of application

Two-way automata naturally arise in the analysis of cryptographic protocols. In this context, terms are constructed using the function symbols $\{\_\}_{\_}$ (binary encryption symbols), $< \_, \_ >$ (pairing) and constants (and other symbols which are irrelevant here). The so-called Dolev-Yao model consists in the deduction rules of Figure 7.3, which express the capabilities of an intruder. For simplicity, we only consider here symmetric encryption keys, but there are similar rules for public key cryptosystems. The rules basically state that an intruder can encrypt a known message with a known key, can decrypt a known message encrypted with k, provided he knows k and can form and decompose pairs.

It is easy to construct a two-way automaton which, given a regular set of terms R, accepts the set of terms that can be derived by an intruder using the rules of Figure 7.3 (see Exercise 7.6).

7.8 Exercises

Exercise 7.1. Show that, for every alternating tree automaton, it is possible to compute in polynomial time an alternating tree automaton which accepts the same language and whose transitions are in disjunctive normal form, i.e. each transition has the form

$$\delta(q,f) = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{k_i} (q_j, l_j)$$

Exercise 7.2. Show that the membership problem for alternating tree automata can be decided in polynomial time.

Exercise 7.3. An alternating automaton is *weak* if there is an ordering on the set of states such that, for every state q and every function symbol f, every state q' occurring in $\delta(q,f)$ satisfies $q' \leq q$.

Prove that the emptiness of weak alternating tree automata is in PTIME.

Exercise 7.4. Given a definite set constraint whose all right hand sides are variables, show how to construct (in polynomial time) k alternating tree automata which accept respectively $X_1\sigma, \ldots, X_k\sigma$ where σ is the least solution of the constraint.

Exercise 7.5. A *pushdown* automaton on words is a tuple $(Q, Q_f, A, \Gamma, \delta)$ where Q is a finite set of states, $Q_f \subseteq Q$, A is a finite alphabet of input symbols, Γ is a finite alphabet of stack symbols and δ is a transition relation defined by rules: $qa \xrightarrow{w} q'$ and $qa \xrightarrow{w^{-1}} q'$ where $q, q' \in Q$, $a \in A$ and $w, w' \in \Gamma^*$.

A configuration is a pair of a state and a word $\gamma \in \Gamma^*$. The automaton may move when reading a, from (q, γ) to (q', γ') if either there is a transition $qa \xrightarrow{w} q'$ and $\gamma' = w \cdot \gamma$ or there is a transition $qa \xrightarrow{w^{-1}} q'$ and $\gamma = w \cdot \gamma'$.

1. Show how to compute (in polynomial time) a two-way automaton which accepts w in state q iff the configuration (q, w) is reachable.
2. This can be slightly generalized considering alternating pushdown automata: now assume that the transitions are of the form: $qa \xrightarrow{w} \phi$ and $qa \xrightarrow{w^{-1}} \phi$ where $\phi \in \mathcal{B}^+(Q)$. Give a definition of a run and of an accepted word, which is consistent with both the definition of a pushdown automaton and the definition of an alternating automaton.
3. Generalize the result of the first question to alternating pushdown automata.
4. Generalize previous questions to tree automata.

Exercise 7.6. Given a finite tree automaton A over the alphabet $\{a, \{\_\}_\_, <\_,\_>\}$, construct a two-way tree automaton which accepts the set of terms t which can be deduced by the rule of Figure 7.3 and the rule

$$\frac{}{t} \quad \text{If } t \text{ is accepted by } A$$

7.9 Bibliographic Notes

Alternation has been considered for a long time as a computation model, e.g. for Turing machines. The seminal work in this area is [CKS81], in which the relationship between complexity classes defined using (non)-deterministic machines and alternating machines is studied.

Concerning tree automata, alternation has been mainly considered in the case of infinite trees. This is especially useful to keep small representations of automata associated with temporal logic formulas, yielding optimal model-checking algorithms [KVW00].

Two-way automata and their relationship with clauses have been first considered in [FSVY91] for the analysis of logic programs. They also occur naturally in the context of definite set constraints, as we have seen (the completion mechanisms are presented in, e.g., [HJ90a, CP97]), and in the analysis of cryptographic protocols [Gou00].

There several other definitions of two-way tree automata. We can distinguish between two-way automata which have the same expressive power as regular languages and what we refer here to pushdown automata, whose expressive power is beyond regularity.

Decision procedures based on ordered resolution strategies could be found in [Jr.76].

Alternating automata with contraints between brothers define a class of languages expressible in Löwenheim's class with equality, also called sometimes the *monadic class*. See for instance [BGG97].

Chapter 8

Automata for Unranked Trees

8.1 Introduction

Finite ordered unranked trees have attracted attention as a formal model for semi-structured data represented by XML documents. Consider for example the (incomplete) HTML document on the left-hand side of Figure 8.1. The use of opening and closing tags surrounding things like the whole document, the head and body of the document, headings, tables, etc. gives a natural tree structure to the document that is shown on the right-hand side of Figure 8.1.

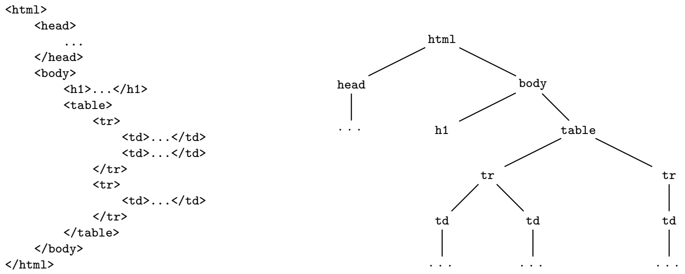
This structure is common to all XML documents and while sometimes the order of the stored data (e.g. the order of the rows in a table) might not be important, its representation as a document naturally enforces an order which has to be taken into account. So we can view unordered unranked trees as a way to model semi-structured data, whereas ordered unranked trees are used to model the documents.

Specifying a class of documents having a desired structure, e.g. the the class of valid HTML documents, corresponds to identifying a set of trees. For example, expressing that a table consists of rows which themselves contain cells, means that we require for the trees that a node labeled `table` has a sequence of successors labeled `tr` and nodes labeled `tr` have a sequence of successors labeled `td`.

There are several formalisms allowing to specify such requirements, e.g. Document Type Definitions (DTDs) and XML schema. To provide a formal background a theory of finite automata on ordered unranked trees has been developed. Actually, such automata models have already appeared in early works on tree automata, e.g. by Thatcher in the late 1960s, but later the focus has been on automata for ranked trees.

The research was reanimated in the 1990s and since then there has been a lot of work in this area. In this chapter we present the basic theory that has been developed for automata on ordered and unranked trees. In the following we simply speak of unranked trees.

We center our presentation around the model of hedge automaton, which is accepted as a natural and fundamental model for automata on unranked



```
<html>
    <head>
        ...
    </head>
    <body>
        <h1>...</h1>
        <table>
            <tr>
                <td>...</td>
                <td>...</td>
            </tr>
            <tr>
                <td>...</td>
            </tr>
        </table>
    </body>
</html>
```

Figure 8.1: An HTML document and its representation as a tree

trees. As we will see, many results can be obtained by encoding unranked trees into ranked ones and then applying the theory of ranked tree automata. Nevertheless, the use of hedge automata as an automaton model directly working on unranked trees is justified because, first of all, the choice of the particular encoding using ranked trees depends on the application, and second, the use of encodings blurs the separation between the two unboundedness aspects (the height of trees is unbounded but also the length of the successor sequences for each node). For example, in the formalism of DTDs there is only a restricted way to specify allowed successor sequences of nodes with a certain label. To reflect such restrictions in an automaton model working on encodings by ranked trees requires a careful choice of the encoding, whereas it directly transfers to hedge automata.

The chapter is structured as follows. In Section 8.2 we give basic definitions and introduce the model of hedge automaton. How to encode unranked trees by ranked ones is discussed in Section 8.3. Section 8.4 presents the equivalence between weak monadic second-order logic and hedge automata, similar to the results in Section 3.3. In Section 8.5 we consider decision problems for hedge automata and Section 8.6 deals with the minimization problem. Finally, we present several formalisms for specifying classes of XML documents, compare their expressive power, and relate them to the model of hedge automata in Section 8.7.

8.2 Definitions and Examples

8.2.1 Unranked Trees and Hedges

In the Preliminaries, finite ordered trees t over some set of labels have been defined as mappings from a finite prefix-closed set $\mathcal{P}os(t) \subseteq N^*$ to the set of labels. Ranked trees are trees where the set of labels is a ranked alphabet and the mapping obeys the restrictions imposed by the ranks of the symbols.

In this chapter we drop this restriction, i.e. we consider a set $\mathcal{U}$ of unranked labels and allow each position in the domain of the tree to have an arbitrary (but finite) number of successors.

A more general definition allows to combine ranked and unranked symbols. For this purpose we consider an alphabet $\Sigma = \mathcal{U} \cup \bigcup_{i=0}^{n} \mathcal{F}_i$, where $\mathcal{F}$ is a ranked alphabet as usual.

A finite (ordered) tree t over Σ is defined as a partial function $t : N^* \rightarrow \Sigma$ with domain written $\mathcal{P}os(t)$ satisfying the following properties:

1. $\mathcal{P}os(t)$ is finite, nonempty, and prefix-closed.
2. $\forall p \in \mathcal{P}os(t)$,
    - if $t(p) \in \mathcal{F}_n$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \{1, \ldots, n\}$,
    - if $t(p) \in \mathcal{U}$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \{1, \ldots, k\}$ for some $k \geq 0$.

This means that the number of children of a node labeled with a symbol from $\mathcal{U}$ is not determined by this symbol.

For simplicity, we only consider the case that $\mathcal{F}$ is empty, i.e. that Σ only contains unranked symbols. In this case, we call a tree t over Σ an ***unranked tree***. The set of all unranked trees over Σ is denoted by $T(\Sigma)$.

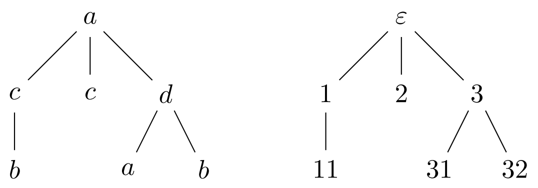
Many definitions that have been given for ranked trees and that do not make use of the ranked nature of the tree, such as ***height*** or ***subtree***, directly carry over to unranked trees.

A tree with root symbol a and subtrees $t_1, \ldots, t_n$ directly below the root is written as $a(t_1 \cdots t_n)$. The sequence $t_1 \cdots t_n$ of unranked trees is also called a ***hedge***. The notion of hedge can be used to give an inductive definition of unranked trees:

- A sequence of unranked trees is a hedge (including the empty sequence ε).
- If h is a hedge and $a \in \Sigma$ is a label, then $a(h)$ is an unranked tree.

If h is the empty hedge (i.e. the empty sequence), then we write a instead of $a()$. The set of all hedges over Σ is denoted by $H(\Sigma)$.

Example 8.2.1. Consider the unranked tree $t = a(c(b)cd(ab))$ and its graphical notation together with the corresponding set of positions:



The subtree at position 3 is $t|_3 = d(ab)$.

8.2.2 Hedge Automata

We want to define a model of finite automaton for unranked trees that is as robust and has similar properties as the model for ranked trees. Recall that a finite tree automaton for ranked trees, as defined in Section 1.1, is specified by a set of transition rules of the form $f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow q(f(x_1, \ldots, x_n))$, also written as $f(q_1, \ldots, q_n) \rightarrow q$, where n is the arity of the symbol f. An input tree is processed in a bottom-up fashion, starting at the leaves and working toward the root using the transition rules.

In unranked trees the number of successors of a position p in a tree t is not determined by its label, and furthermore there is no upper bound on the number of successors that a position can have. Therefore automata for unranked trees cannot be defined by explicitly listing all transition rules in the same style as for ranked tree automata.

To deal with the unbounded branching of unranked trees we have to find a way to symbolically represent an infinite number of transitions.

Example 8.2.2. Assume we want to define an automaton accepting all unranked trees of height 1 over the alphabet $\Sigma = \{a, b\}$ that have an even number of leaves, all leaves are labeled by b, and the root is labeled by a, i.e. the trees a, $a(bb)$, $a(bbbb)$, $\cdots$.

For this purpose, we could use the two states q_b and q. The first transition rule $b \rightarrow q_b$ ensures that the leaves can be labeled by q_b. To capture all the trees listed above we need the infinite set of rules

$$a \rightarrow q, a(q_b q_b) \rightarrow q, a(q_b q_b q_b q_b) \rightarrow q, \cdots$$

This set can be represented by a single rule $a((q_b q_b)^*) \rightarrow q$ using a regular expression for describing the sequences of states below the position with label a that enable the automaton to proceed to state q.

In the previous example we have used a regular expression to represent an infinite number of transition rules. In general, this leads to transition rules of the form $a(R) \rightarrow q$, where $R \subseteq Q^*$ is a regular language over the states of the unranked tree automaton.

Of course it would be possible to allow R to be from a bigger class of languages, e.g. the context-free languages. But as we are interested in defining *finite* automata for unranked trees, i.e. automata using memory of bounded size, we should choose R such that it can be represented by an automaton using only bounded memory. Thus, the class of regular languages is a natural choice in this context.

A ***nondeterministic finite hedge automaton*** (NFHA) over Σ is a tuple $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a finite set of transition rules of the following type:

$$a(R) \rightarrow q$$

where $R \subseteq Q^*$ is a regular language over Q. These languages R occurring in the transition rules are called ***horizontal languages***.

A ***run*** of $\mathcal{A}$ on a tree $t \in T(\Sigma)$ is a tree $r \in T(Q)$ with the same domain as t such that for each node $p \in \mathcal{P}os(r)$ with $a = t(p)$ and $q = r(p)$ there is a transition rule $a(R) \rightarrow q$ of $\mathcal{A}$ with $r(p1) \cdots r(pn) \in R$, where n denotes the number of successors of p. In particular, to apply a rule at a leaf, the empty word ε has to be in the horizontal language of the rule.

An unranked tree t is ***accepted*** by $\mathcal{A}$ if there is a run r of $\mathcal{A}$ on t whose root is labeled by a final state, i.e. with $r(\varepsilon) \in Q_f$. The ***language*** $L(\mathcal{A})$ of $\mathcal{A}$ is the set of all unranked trees accepted by $\mathcal{A}$.

These automata can be seen as working bottom-up, starting at the leaves and ending at the root. For nondeterministic automata this is just a matter of taste, except that in the top-down view one would rather call the set of final states the set of initial states and write the transitions as $q(a) \to R$ instead of $a(R) \to q$. For deterministic automata there is a difference because one has to specify in which direction the automaton should be deterministic. In Section 8.2.3 we consider deterministic bottom-up automata.

As for ranked trees we call two NFHAs ***equivalent*** if they accept the same language, and if t is accepted by $\mathcal{A}$ with a run whose root is labeled q, then we also write $t \xrightarrow[\mathcal{A}]{*} q$.

In the following examples we give the horizontal languages using regular expressions. For the specification of transitions we can use any formalism defining regular languages, for example nondeterministic finite automata (NFAs). But of course the choice of the formalism is important when considering algorithms for NFHAs. This issue is discussed in Section 8.5.

Example 8.2.3. Let $\Sigma = \{a, b, c\}$ and $L \subseteq T(\Sigma)$ be the language of all trees t for which there are two nodes labeled b whose greatest common ancestor is labeled c. More formally, we consider all trees for which there exist nodes $p_1, p_2 \in \mathcal{P}os(t)$ with $t(p_1) = t(p_2) = b$ such that $t(p) = c$ for the greatest common ancestor (the longest common prefix) of p_1 and p_2.

A tree satisfying this condition is shown in Figure 8.2. The corresponding nodes are $p_1 = 12$, $p_2 = 132$, and $p = 1$.

To accept this language, the following NFHA labels b-nodes with q_b, as long as it has not yet verified the required condition in a subtree, passes this q_b upward, and checks if there is a c-node with two successors labeled q_b. This node is then labeled with the final state q_c that is passed up to the root.

Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ with $Q = \{q, q_b, q_c\}$, $Q_f = \{q_c\}$, and Δ given by the following rules:

$$\begin{array}{rrr}
a(Q^*) \to q & a(Q^* q_b Q^*) \to q_b & a(Q^* q_c Q^*) \to q_c \\
b(Q^*) \to q_b & c(Q^* q_b Q^*) \to q_b & b(Q^* q_c Q^*) \to q_c \\
c(Q^*) \to q & c(Q^* q_b Q^* q_b Q^*) \to q_c & c(Q^* q_c Q^*) \to q_c
\end{array}$$

The right-hand side of Figure 8.2 shows a run of $\mathcal{A}$. Consider, for example, node 1. It is labeled c in the input tree. In the run, the sequence of its successor labels is the word $q q_b q_b$. This means that we can apply the rule $c(Q^* q_b Q^* q_b Q^*) \to q_c$ and hence obtain q_c at node 1 in the run.

Example 8.2.4. In Example 1.1.3 on page 22 we have seen an NFTA accepting the set of all true Boolean expressions built from a unary negation symbol and binary conjunction and disjunction symbols. As conjunction and disjunction are associative, we can also view them as operators without fixed arity.

Consider the alphabet $\Sigma = \{or, and, not, 0, 1\}$ and the NFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$

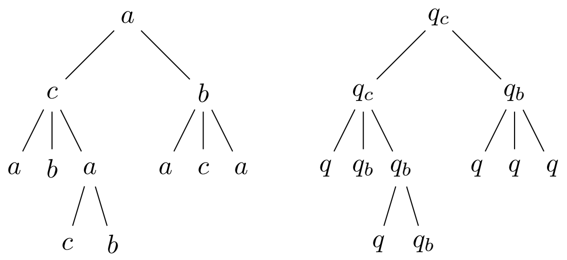


Figure 8.2: A run of the NFHA from Example 8.2.3

with $Q = \{q_0, q_1\}$, $Q_f = \{q_1\}$, and Δ given by the following rules:

$$\begin{array}{rr} 0(\varepsilon) \rightarrow q_0 & or(Q^* q_1 Q^*) \rightarrow q_1 \\ 1(\varepsilon) \rightarrow q_1 & or(q_0 q_0^*) \rightarrow q_0 \\ not(q_0) \rightarrow q_1 & and(Q^* q_0 Q^*) \rightarrow q_0 \\ not(q_1) \rightarrow q_0 & and(q_1 q_1^*) \rightarrow q_1 \end{array}$$

This automaton accepts all trees that form Boolean expressions evaluating to true. The expressions have to be correct in the sense that the leaves are labeled by 0 or 1 and the nodes labeled *not* have exactly one successor.

We call a language $L \subseteq T(\Sigma)$ of unranked trees ***hedge recognizable*** if there is an NFHA accepting this language.

We now discuss some basic properties of NFHAs similar to the ones for NFTAs in Section 1.1.

An NFHA $\mathcal{A}$ is ***complete*** if for each $t \in T(\Sigma)$ there is at least one state q such that $t \xrightarrow[\mathcal{A}]{*} q$. The automaton from Example 8.2.3 is complete because for each label and for each sequence of states, one of the three rules $a(Q^*) \rightarrow q$, $b(Q^*) \rightarrow q_b$, $c(Q^*) \rightarrow q$ can be applied. The automaton from Example 8.2.4 is not complete because, for example, there is no transition rule that can be applied if the label 0 appears at an inner node of the tree.

Due to the closure properties of regular word languages, it is easy to see that each NFHA can be completed by adding a sink state.

Remark 8.2.5. For each NFHA one can compute a complete NFHA with at most one state more that accepts the same language.

Proof. Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be an NFHA. Let $q_\perp$ be a new state that is not in Q. For $a \in \Sigma$ let $a(R_1) \rightarrow q, \ldots, a(R_n) \rightarrow q_n$ be all rules for a. We complete $\mathcal{A}$ by adding the rule $a(R_\perp) \rightarrow q_\perp$ with $R_\perp = (Q \cup \{q_\perp\})^* \setminus (R_1 \cup \cdots \cup R_n)$. ☐

In Section 8.5 we discuss the complexity of deciding whether a given NFHA is complete.

We call $\mathcal{A}$ ***reduced*** if each state of $\mathcal{A}$ is reachable, i.e. for each state q there is a tree t with $t \xrightarrow[\mathcal{A}]{*} q$. The algorithm for computing the set of reachable states is presented in Section 8.5 for solving the emptiness problem. From that we can conclude that for a given NFHA we can compute a reduced one accepting the same language.

Another thing that one might notice in the representation of NFHAs is that it is not necessary to have different rules $a(R_1) \to q$ and $a(R_2) \to q$ because a single rule $a(R_1 \cup R_2) \to q$ allows the same transitions (and the class of regular languages is closed under union). We call an NFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ ***normalized*** if for each $a \in \Sigma$ and $q \in Q$ there is at most one transition rule of the form $a(R) \to q$. The NFHA from Example 8.2.3 is not normalized because it has two transitions for c and q_c, whereas the NFHA from Example 8.2.4 is normalized.

As mentioned above this property can always be obtained easily.

Remark 8.2.6. Each NFHA is equivalent to a normalized NFHA.

As for NFTAs we can allow ϵ-rules. These ϵ-rules can be removed using a construction similar to the one for NFTAs (see Exercise 8.2).

In Section 1.3 it is shown that the class of recognizable languages of ranked trees is closed under set operations like union, intersection, and complement. It is not difficult to adapt the proofs from the case of ranked trees to the unranked setting (see Exercises). But instead of redoing the constructions in the new setting we show in Section 8.3 how to encode unranked trees by ranked ones. This enables us to reuse the results from Section 1.3.

8.2.3 Deterministic Automata

In this section we consider deterministic bottom-up automata. In the case of ranked tree automata, deterministic (bottom-up) automata (DFTAs) are those that do not have two rules with the same left-hand side (compare Section 1.1). This immediately implies that for each ground term t there is at most one state q of the automaton such that $t \xrightarrow{*} q$.

To transfer this property to unranked trees we have to ensure that the state at a node p in the run of a deterministic automaton should be determined uniquely by the label of p and the states at the successors of p.

We obtain the following definition. A ***deterministic finite hedge automaton*** (DFHA) is a finite hedge automaton $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ such that for all rules $a(R_1) \to q_1$ and $a(R_2) \to q_2$ either $R_1 \cap R_2 = \emptyset$ or $q_1 = q_2$.

Example 8.2.7. The NFHA from Example 8.2.3 is not deterministic because it contains, e.g., the two transitions $a(Q^*) \to q$ and $a(Q^* q_b Q^*) \to q_b$. The NFHA from Example 8.2.4 is deterministic.

By using a standard subset construction as for ranked tree automata, it is not difficult to see that every NFHA can be transformed into a deterministic one.

Theorem 8.2.8. *For every NFHA $\mathcal{A}$ there is a DFHA $\mathcal{A}_d$ accepting the same language. The number of states of $\mathcal{A}_d$ is exponential in the number of states of $\mathcal{A}$.*

Proof. The construction is very similar to the one for proving Theorem 1.1.9. The states of $\mathcal{A}_d$ are sets of states of $\mathcal{A}$. For the transitions we have to compute, given a letter a and a sequence $s_1 \cdots s_n$ of sets of states, all possible states of $\mathcal{A}$ that can be reached on an a-labeled node whose successors are labeled by a state sequence composed from elements of the sets $s_1, \dots, s_n$.

Formally, let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ and define $\mathcal{A}_d = (Q_d, \Sigma, Q_{df}, \Delta_d)$ by $Q_d = 2^Q$, $Q_{df} = \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$, and Δ_d containing all the transitions $a(R) \to s$ such that

$$\begin{aligned}&s_1 \cdots s_n \in R \text{ iff}\\ &\quad s = \{q \in Q \mid \exists q_1 \in s_1, \dots, q_n \in s_n, a(R') \to q \in \Delta \text{ with } q_1 \cdots q_n \in R'\}.\end{aligned}$$

This automaton is obviously deterministic because for each a and each s there is only one transition having s as target. But we have to show that the sets R defined as above are regular.

For this purpose, assume that $\mathcal{A}$ is normalized and let $R_{a,q}$ be the language with $a(R_{a,q}) \to q \in \Delta$ for all $a \in \Sigma$ and $q \in Q$. The language

$$S_{a,q} = \{s_1 \cdots s_n \in Q_d^* \mid \exists q_1 \in s_1, \dots, q_n \in s_n \text{ with } q_1 \cdots q_n \in R_{a,q}\}$$

is regular because it is the set of all words obtained from words of $R_{a,q}$ by substituting at each position of the word a set of states containing the respective state at this position. In general, such an operation is called substitution in the context of regular word languages, and the class of regular word languages is closed under substitution.

With this definition we can write the language R with $a(R) \to s \in \Delta_d$ as

$$R = (\bigcap_{q \in s} S_{a,q}) \setminus (\bigcup_{q \notin s} S_{a,q})$$

showing that it is regular.

For the correctness of the construction one easily shows by induction on the height of the trees that

$$t \xrightarrow[\mathcal{A}_d]{*} s \text{ iff } s = \{q \in Q \mid t \xrightarrow[\mathcal{A}]{*} q\}.$$

We omit the details of this straightforward induction. □

8.3 Encodings and Closure Properties

In this section we present two ways of encoding unranked trees by ranked ones:

- the first-child-next-sibling (FCNS) encoding, and
- the extension encoding.

The first one (FCNS) arises when unranked trees have to be represented by data structures using pointers and lists. As for each node the number of children is unbounded, a natural way to represent this is to have for each node a pointer to the list of its successors. The successors are naturally ordered from left to right. Following this scheme, one ends up with a representation in which each node has a pointer to its first child and to its right sibling. This can be viewed as a binary tree that is known as the FCNS encoding of an unranked tree.

The second encoding takes a more algebraic view and the general idea behind it is not specific to unranked trees: Assume we are dealing with a set O of objects, and that there is a finite set of operations and a finite set of basic objects such that each element of O can be constructed by using these operations starting from the basic objects. Then we can represent each element of O by a (not necessarily unique) finite term that can be used to construct this element. Many structures used in computer science can be represented this way (for example natural numbers using 1 as basic object and the addition as operation, or lists using lists of length one and concatenation).

The extension encoding is of this kind. It uses a single operation that allows to construct all unranked trees from the set of trees of height 0.

Once we have given an encoding we can analyze the notion of recognizability using ranked tree automata on the encodings and how it transfers to the unranked trees (by looking at the pre-images of all recognizable sets of encodings). It turns out that the notion of recognizability using hedge automata is the same as the notion of recognizability obtained from both encodings. This shows that hedge automata provide a robust notion of recognizability and it furthermore allows us to easily transfer several closure properties from the ranked to the unranked setting.

8.3.1 First-Child-Next-Sibling Encoding

The idea for the ***first-child-next-sibling (FCNS) encoding*** has already been explained in the introduction of this section. It can be viewed as the binary tree obtained from the unranked tree by representing it using two pointers for each node, one to its first child and the second to its right sibling.

In the terminology of ranked trees, all symbols from the unranked alphabet become binary symbols and an additional constant # is introduced (corresponding to the empty list). For a node of the unranked tree we proceed as follows: As left successor we put its first child from the unranked tree, as right successor we put its right sibling from the unranked tree. Whenever the corresponding node does not exist, then we put # in the encoding.

Example 8.3.1. The unranked tree $a(c(b)cd(bb))$ is encoded as

$$a(c(b(\#,\#),c(\#,d(b(\#,b(\#,\#),\#))),\#).$$

Figure 8.3 shows the tree itself, its representation using two pointers (where the solid arrow points to the first child and the dashed arrow to the right sibling), and the FCNS encoding.

Formally, let $\mathcal{F}^{\Sigma}_{\text{FCNS}} = \{a(,) \mid a \in \Sigma\} \cup \{\#\}$ for an unranked alphabet Σ. Recall that the notation $a(,)$ means that a is a binary symbol. We define the encoding not only for trees but also for hedges. The following inductive definition of the function $\mathsf{fcns} : H(\Sigma) \to T(\mathcal{F}^{\Sigma}_{\text{FCNS}})$ corresponds to the encoding idea described above. Recall that $t[t']_p$ denotes the tree obtained from t by replacing the subtree at position p by t'.

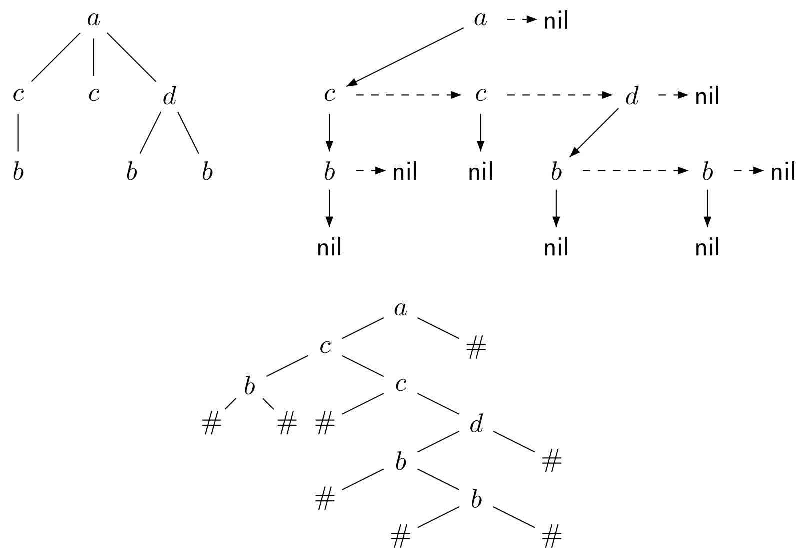


Figure 8.3: An unranked tree, its representation using lists, and its FCNS encoding

- For $a \in \Sigma$: $\mathsf{fcns}(a) = a(\#, \#)$
- For a tree $t = a(t_1 \cdots t_n)$: $\mathsf{fcns}(t) = a(\mathsf{fcns}(t_1 \cdots t_n), \#)$
- For a hedge $h = t_1 \cdots t_n$ with $n \geq 2$: $\mathsf{fcns}(h) = \mathsf{fcns}(t_1)[\mathsf{fcns}(t_2 \cdots t_n)]_2$

The last item from the definition corresponds to the following situation: To define the encoding of a hedge we take the encoding of its first tree, and let the pointer for the right sibling point to the encoding of the remaining hedge.

We extend this mapping to sets of trees in the usual way:

$$\mathsf{fcns}(L) = \{\mathsf{fcns}(t) \mid t \in L\}.$$

It is rather easy to see that the function $\mathsf{fcns} : H(\Sigma) \to T(\mathcal{F}^{\Sigma}_{\mathrm{FCNS}})$ is a bijection, i.e. we can use the inverse function fcns^{-1}.

From the definition one can directly conclude that applying fcns^{-1} to a tree $t \in T(\mathcal{F}^{\Sigma}_{\mathrm{FCNS}})$ yields an unranked tree (rather than just a hedge) iff $t(2) = \#$. As this property can be checked by an NFTA, a simple consequence is that the image $\mathsf{fcns}(T(\Sigma))$ of all unranked trees is a recognizable set. This can easily be generalized:

Proposition 8.3.2. *If $L \subseteq T(\Sigma)$ is hedge recognizable, then $\mathsf{fcns}(L)$ is recognizable.*

Proof. We show how to construct an NFTA for $\mathsf{fcns}(L)$. We start from an NFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ for L and assume that it is normalized, with the horizontal languages accepted by NFAs $\mathcal{B}_{a,q}$ with state set $P_{a,q}$, final states $F_{a,q}$, transition relation $\Delta_{a,q}$, and initial state $p^0_{a,q}$. The idea for the transformation to an NFTA on the FCNS encoding is rather simple: On the branches to the right we simulate

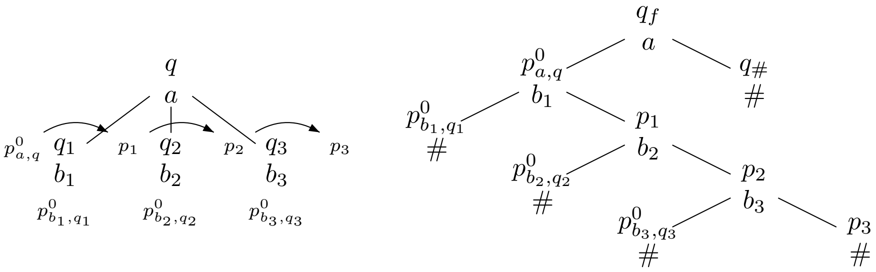


Figure 8.4: Illustration of the construction from Proposition 8.3.2

the NFAs. At the #-symbols a final state of the NFA has to be reached, showing that the successor word is accepted by the NFA.

We define $\mathcal{A}' = (Q', \mathcal{F}^{\Sigma}_{\mathrm{FCNS}}, Q'_f, \Delta')$ as follows, where $P = \bigcup_{a\in\Sigma, q\in Q} P_{a,q}$ and $F = \bigcup_{a\in\Sigma, q\in Q} F_{a,q}$:

- $Q' = \{q_f, q_\#\} \cup P$ for two new states q_f and $q_\#$.
- $Q'_f = \{q_f\}$.
- Δ' contains the following transitions:
  - $\# \to p$ for all $p \in F$
  - $\# \to q_\#$. This rule is used to identify the right leaf below the root, which has to exist in a correct coding of an unranked tree.
  - $b(p', p'') \to p$ for all $p', p'', p \in P$ such that there is $a \in \Sigma$ and $q, q' \in Q$ with $p' = p^0_{b,q'}$, and $(p, q', p'') \in \Delta_{a,q}$.
  - $a(p, q_\#) \to q_f$ if $p = p^0_{a,q}$ for some $q \in Q_f$. This rule is intended to be used only at the root (there are no further transitions using q_f).

Figure 8.4 illustrates how a run on the unranked tree is transferred to the FCNS encoding. On the left-hand side a run of a hedge automaton on an unranked tree is shown. Additionally the runs of the automata $\mathcal{B}_{a,q}$ are shown because this is what is transferred to the FCNS encoding, as depicted on the right-hand side of the figure. Note that the states $p^0_{b_1,q_1}, p^0_{b_2,q_2}, p^0_{b_3,q_3}$ and p_3 are final states of the corresponding NFAs.

For the correctness of this construction one can verify the following property: Consider a hedge $t_1 \cdots t_n$ with $t_i \to^*_{\mathcal{A}} q_i$ for all $i \in \{1, \ldots, n\}$ and assume that $q_1 \cdots q_n$ is accepted by some $\mathcal{B}_{a,q}$ with a run starting in some state $p \in P_{a,q}$. This is the case if and only if $\mathsf{fcns}(t_1 \cdots t_n) \to^*_{\mathcal{A}'} p$. This can be shown by structural induction on the hedge under consideration. We omit the details of this proof. □

For a similar statement in the other direction we remove the hedges by intersecting with the set of all unranked trees after applying the inverse of the FCNS encoding.

Proposition 8.3.3. *If $L \subseteq T(\mathcal{F}^{\Sigma}_{\mathrm{FCNS}})$ is recognizable, then $\mathsf{fcns}^{-1}(L) \cap T(\Sigma)$ is hedge recognizable.*

Proof. From an NFTA for $L \subseteq T(\mathcal{F}^{\Sigma}_{\mathrm{FCNS}})$ one can construct a NFHA accepting the desired language. The details of this construction are left as an exercise. □

These propositions allow to transfer closure properties from ranked tree automata to hedge automata. Before we do this in detail we first introduce another encoding.

8.3.2 Extension Operator

As mentioned in the introduction to this section, we now consider an encoding using an operation for building unranked trees. In this setting, an unranked tree t is represented by the term that evaluates to this tree t.

The encoding is based on the ***extension operator*** @. This operator connects two trees by adding the second tree as the right-most subtree below the root of the first tree. More formally, for two trees $t, t' \in T(\Sigma)$ with $t = a(t_1 \cdots t_n)$ we let

$$t \mathbin{@} t' = a(t_1 \cdots t_n t').$$

If t has height 0, i.e. if $n = 0$, then we get that $a \mathbin{@} t' = a(t')$.

Every unranked tree can be built in a unique way from the symbols of Σ, i.e. the trees of height 0, by applying the extension operator.

Example 8.3.4. The unranked tree $a(bc)$ can be built by starting with the tree a, then adding b as subtree using the extension operator, and then adding c as right-most subtree by using the extension operator again. So we get that $a(bc) = (a \mathbin{@} b) \mathbin{@} c$.

As @ is a binary operator, we can use it to code unranked trees by ranked ones. We use the symbols from Σ as constants and @ as the only binary symbol: for an unranked alphabet Σ let $\mathcal{F}^{\Sigma}_{\mathsf{ext}} = \{@(,)\} \cup \{a \mid a \in \Sigma\}$.

The ***extension encoding*** $\mathsf{ext} : T(\Sigma) \to T(\mathcal{F}^{\Sigma}_{\mathsf{ext}})$ is inductively defined as follows:

- For $a \in \Sigma$ let $\mathsf{ext}(a) = a$.
- For $t = a(t_1 \cdots t_n)$ with $n \geq 1$ let $\mathsf{ext}(t) = @(\mathsf{ext}(a(t_1 \cdots t_{n-1})), \mathsf{ext}(t_n))$.

Another way of stating the second rule is: If t is not of height 0, then it is of the form $t = t' \mathbin{@} t''$ and we let $\mathsf{ext}(t) = @(\mathsf{ext}(t'), \mathsf{ext}(t'')))$.

As before, we apply the encoding also to sets and let

$$\mathsf{ext}(L) = \{\mathsf{ext}(t) \mid t \in L\}.$$

Example 8.3.5. Consider the tree $t = a(c(b)cd(bb))$ depicted on the left-hand side of Figure 8.5. The recursive rule for the encoding tells us that $\mathsf{ext}(t) = @(\mathsf{ext}(a(c(b)c)), \mathsf{ext}(d(bb)))$. So the left subtree in the encoding surrounded by the dashed line corresponds to the part of the unranked tree surrounded by the dashed line.

In each further step of the encoding one more subtree is removed until only the symbol at the root is left. So the a at the left-most leaf in the encoding corresponds to the a at the root of the unranked tree.

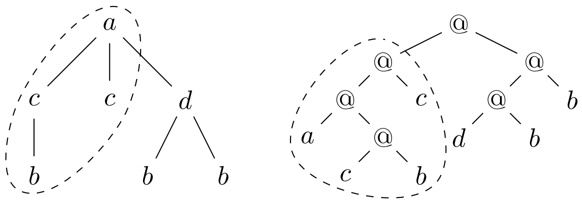


Figure 8.5: An unranked tree and its extension encoding

Although this encoding is less intuitive than the FCNS encoding, it has some advantages. First of all, each tree over the alphabet $\mathcal{F}^{\Sigma}_{\mathsf{ext}}$ corresponds to a tree (and not only to a hedge as for the FCNS encoding).

Proposition 8.3.6. *The extension encoding* $\mathsf{ext} : T(\Sigma) \to T(\mathcal{F}^{\Sigma}_{\mathsf{ext}})$ *is a bijection.*

Similar to the FCNS encoding, recognizability is preserved by the extension encoding in both directions.

Theorem 8.3.7. *A language* $L \subseteq T(\Sigma)$ *is hedge recognizable if, and only if,* $\mathsf{ext}(L)$ *is recognizable.*

We omit the proof because in Subsection 8.6.3 we present the tight relation between deterministic hedge automata and deterministic bottom-up automata on the extension encoding. The same ideas can be used to make the translation for nondeterministic automata. It turns out that one can almost directly transfer the transition relation of the horizontal automata of a hedge automaton to transitions on the extension encoding.

8.3.3 Closure Properties

We have seen that we obtain the same notion of recognizability when using hedge automata or automata on encodings. Therefore, we call languages $L \subseteq T(\Sigma)$ that are recognizable in one of the formalisms (and thus in all) simply ***recognizable***. Using the closure properties that have been shown in Section 1.3 for ranked tree automata, we can easily conclude that recognizable sets of unranked trees have similar properties.

Theorem 8.3.8. *The class of recognizable unranked tree languages is closed under union, under intersection, and under complementation.*

Proof. Let $L_1, L_2 \subseteq T(\Sigma)$ be recognizable. From Theorem 8.3.7 we conclude that $\mathsf{ext}(L_1)$ and $\mathsf{ext}(L_2)$ are recognizable. The class of recognizable languages of ranked trees is closed under union (Theorem 1.3.1) and we have $\mathsf{ext}(L_1) \cup \mathsf{ext}(L_2) = \mathsf{ext}(L_1 \cup L_2)$. Thus, $\mathsf{ext}(L_1 \cup L_2)$ and therefore $L_1 \cup L_2 = \mathsf{ext}^{-1}(\mathsf{ext}(L_1 \cup L_2))$ are recognizable.

The same arguments can be used for intersection and complementation. □

The concept of homomorphisms, introduced in Section 1.4, relies on the fact that every symbol has a fixed rank. We only adapt the notion of alphabetic homomorphism, which we also call projection in this context.

A ***projection*** is mapping $h : \Sigma \to \Sigma'$. This mapping is extended to trees by applying it to each node label, i.e. we can view it as a function $h : T(\Sigma) \to T(\Sigma')$. On the level of sets of trees we look at h and its inverse, defined as

$$h(L) = \{h(t) \mid t \in L\} \text{ and } h^{-1}(L') = \{t \in T(\Sigma) \mid h(t) \in L'\}.$$

From the results in Section 1.4 we can easily derive the following.

Theorem 8.3.9. *The class of recognizable unranked tree languages is closed under projections and inverse projections.*

8.4 Weak Monadic Second Order Logic

Connections between automata and logic are a valuable tool for developing decision procedures for the logics under consideration. Furthermore, a robust logic which is equivalent to automata in expressive power can serve as a yardstick for the expressiveness of query languages.

In Section 3.3 we have seen a logical characterization of recognizable tree languages using the logic WSkS. In this section we present a similar logic for unranked trees. In WSkS one can directly access the ith successor of a node x by the term xi. For unranked trees this would lead to an infinite vocabulary, and as each formula can only use finitely many symbols it would not even be possible to write a single formula stating that two nodes have the same parent.

To avoid this problem we introduce the two binary relations $\mathit{child}(x, y)$ and $\mathit{next\text{-}sibling}(x, y)$ interpreted as "y is a child of x" and "y is the next sibling to the right of x", respectively. The first-order elements are now interpreted as strings over N^* (as opposed to $\{1, \ldots, k\}^*$ for WSkS). The semantics of the relations is the following:

- $\forall p, p' \in N^*$, $\mathit{child}(p, p')$ iff $p' = pi$ for some $i \in N$,
- $\forall p, p' \in N^*$, $\mathit{next\text{-}sibling}(p, p')$ iff there is $p'' \in N^*$ with $p = p''i$, $p' = p''j$ for some $i, j \in N$ with $j = i + 1$.

We now consider standard ***weak monadic second-order logic*** (WMSO) over this signature. We use first-order variables $x, y, \ldots$ interpreted by elements of N^*, and monadic second-order variables $X, Y, \ldots$ interpreted by finite subsets of N^* (similar to WSkS). WMSO-formulas are built from the atomic formulas $x = y$, $x \in X$, $\mathit{child}(x, y)$, and $\mathit{next\text{-}sibling}(x, y)$ by the logical connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, etc., and by quantification over elements and finite sets of elements.

We do not give a formal definition of the semantics because it is defined along the same lines as for WSkS.

Example 8.4.1. The following formula defines the *right-sibling* relation, i.e. the relation stating that x and y are siblings and that y is to the right of x. The formula $\mathit{right\text{-}sibling}(x, y)$ is defined as follows:

$$\forall X.\big(x \in X \wedge \forall z, z'.(z \in X \wedge \mathit{next\text{-}sibling}(z, z') \Rightarrow z' \in X)\big) \Rightarrow y \in X$$

In the same way it is possible to define the *descendant* relation, where $\mathit{descendant}(x, y)$ is true if x is above y in the tree, i.e. x is a prefix of y.

Using these relations it is possible to specify a formula Domain(X) stating that the set X codes a tree domain:

$$\forall x, y.(y \in X \wedge (\mathit{descendant}(x, y) \vee \mathit{right\text{-}sibling}(x, y) \Rightarrow x \in X)$$

Note that we do not have to explicitly require that this set is finite because set variables are interpreted by finite sets by definition.

In the same way as for WSkS we can use WMSO to define sets of unranked trees by using formulas of the form $\varphi(X, X_1, \ldots, X_n)$, where X codes the domain of the tree and $X_1, \ldots, X_n$ the labeling. A set X_i codes the positions in the tree labeled by a_i if the alphabet is $\Sigma = \{a_1, \ldots, a_n\}$. We can describe correct codings of trees by using the formula Domain(X) and by requiring that $X_1, \ldots, X_n$ form a partition of X.

We skip the formal definitions as they are completely analogous to the ones from Section 3.3. We call a language $L \subseteq T(\Sigma)$ ***definable*** if there exists a WMSO-formula defining it.

It turns out that definability and recognizability are the same as in the case of ranked trees. This is not a surprising result as the relations used in WMSO for unranked trees provide a direct link to the FCNS encoding.

Theorem 8.4.2. *A language $L \subseteq T(\Sigma)$ is definable in WMSO if and only if it is recognizable.*

Proof. Assume that L is definable in WMSO. It is straightforward to translate a defining WMSO-formula $\varphi(X, X_1, \ldots, X_n)$ into a formula $\varphi'(X, X_1, \ldots, X_n, X_\#)$ of WS2S such that φ' defines the language $\mathsf{fcns}(L)$. Here, the set $X_\#$ encodes the positions labeled # in the encoding. The formula φ' is obtained by

- requiring that the sets $X, X_1, \ldots, X_n, X_\#$ indeed represent a term (as described in Section 3.3),
- this term codes an unranked tree (not a hedge), and
- replacing atomic formulas in φ according to the following table

$$\begin{array}{lll} x \in X & \rightsquigarrow & x \in X \wedge x \notin X_\# \\ \mathit{child}(x, y) & \rightsquigarrow & y \in x12^* \\ \mathit{next\text{-}sibling}(x, y) & \rightsquigarrow & y = x2 \end{array}$$

  where $x12^*$ is a short notation for the set of elements z satisfying the formula $\mathit{descendant}(x1, z) \wedge \forall x'.(\mathit{descendant}(x, x') \wedge \mathit{descendant}(x'1, z) \Rightarrow x = x')$.

From this we obtain that L being definable implies that L is recognizable: Transform the defining formula into a WS2S-formula defining $\mathsf{fcns}(L)$ as described above. Then $\mathsf{fcns}(L)$ is recognizable according to Theorem 3.3.7 from Section 3.3 and hence L is recognizable (Proposition 8.3.3).

For the other direction, the approach of transferring a WS2S-formula for $\mathsf{fcns}(L)$ into a WMSO-formula for L is more technical because one has to take care of the elements of the form $2\{1, 2\}^*$, which are not used in the FCNS encoding and hence have no counterpart in the unranked tree.

So in fact, it is easier to construct a WMSO-formula that describes the existence of an accepting run of a hedge automaton for L. We omit the details because the construction of the formula is similar to the one from Lemma 3.3.6 from Section 3.3. □

8.5 Decision Problems and Complexity

In this section we study various decision problems for recognizable languages of unranked trees. We have already seen that choosing appropriate encodings by ranked trees allows to reduce many problems to the well-studied ranked setting. This is certainly also true to some extent for decision problems. Consider for example the emptiness problem: "Given a hedge automaton $\mathcal{A}$, is $L(\mathcal{A})$ empty?"

We can use the construction from Theorem 8.3.7 to construct a ranked tree automaton for the extension encoding of $L(\mathcal{A})$. Then we apply the results from Section 1.7 and conclude that the problem is decidable. This approach has the advantage that we can reuse existing algorithms. A disadvantage of this approach is that it does not take into account the representation of the horizontal languages while languages of unranked trees are commonly defined by schemas using horizontal languages. If the representation of the horizontal languages uses a rather complex formalism, then hedge automata cannot be transfered to automata on the encodings as easily as in Theorems 8.3.7 and 8.3.2. So going through encodings might result in higher complexities in these situations.

From this point of view it is worth developing generic algorithms that directly work with hedge automata and using, whenever possible, required decision procedures for the horizontal level as a black box. In this section we first discuss some representations for the horizontal languages and then consider several decision problems.

To estimate the complexity of algorithms we use as size of the automata the size of their representation. This of course depends on how the horizontal languages are specified, but for these representations we use size definitions which are standard as, e.g., the length of regular expressions or the number of states plus number of transitions for NFAs.

8.5.1 Representations of Horizontal Languages

In Section 8.2 we have specified some example hedge automata using regular expressions for the horizontal languages. We also mentioned that w.r.t. expressive power we can use in principle any formalism for defining regular word languages. Here we present some of such representations and introduce some notations that will be used in the later subsections to illustrate the differences between direct algorithms for hedge automata and the approach via encodings.

The most common way of specifying hedge automata is to use regular expressions or NFAs for the horizontal languages. This might result in rather large automata for simple properties because these formalisms do not allow to easily specify conjunctions of properties or the absence of certain patterns.

Consider the following simple example. We are given some trees $t_1, \ldots, t_n \in T(\Sigma)$ and a label $a \in \Sigma$ and want to construct a hedge automaton that recognizes the trees with the property that each node with label a has each tree from

$t_1, \dots, t_n$ as subtree at least once (in any order). A hedge automaton would need states $q_1, \dots, q_n$ where q_i signals that the tree t_i has been read. A standard regular expression that checks that all states $q_1, \dots, q_n$ occur in a word basically has to explicitly list all possible permutations. A way to bypass this problem is to allow in the representation of the horizontal languages intersections of regular expressions. Then the required horizontal language can simply be defined by $\bigcap_{i=1}^{n} Q^* q_i Q^*$ if Q is the state set of the hedge automaton. Note that we do not allow the intersection as an operator in the expressions but only on the outermost level.

We can generalize this by allowing also negations of regular expressions or arbitrary Boolean combinations of regular expressions. An example for a negation of regular expression is $\sim (Q^* q_1 q_2 q_3 Q^*)$ stating that the pattern $q_1 q_2 q_3$ does not appear. Specifying the absence of such patterns with standard regular expressions is possible but much more involved.

Expressions of this kind can be translated into alternating finite automata (AFA) of size linear in the expression: The regular expressions can first be translated into NFAs, and then the Boolean operations are applied introducing alternation in the transition function but without increasing the size of the state set (compare Chapter 7).

Since Boolean combinations of regular expressions are a rather natural and convenient formalism for specifying languages, and as they easily translate to AFAs, we will also consider AFAs as a formalism for representing horizontal languages.

Proposition 8.5.1. *The (uniform) membership problem for AFAs can be solved in polynomial time, and the emptiness problem for AFAs is PSPACE-complete.*

Proof. The membership problem is polynomial even for the more general formalism of alternating tree automata (compare Section 7.5). For the emptiness problem see the bibliographic notes. ☐

Another way of extending regular expressions for making them more succinct is the ***shuffle*** or ***interleaving*** operator $\|$. A shuffle of two words u and v is an arbitrary interleaving of the form $u_1 v_1 \cdots u_n v_n$ where the u_i and v_i are finite words (that may be empty) such that $u = u_1 \cdots u_n$ and $v = v_1 \cdots v_n$.

Example 8.5.2. For the words *ab* and *cd* the possible shuffles are *abcd*, *acbd*, *acdb*, *cabd*, *cadb*, *cdab*.

As usual, the shuffle of two word languages consists of all shuffles of words from these languages:

$$R_1 \parallel R_2 = \{w \mid \exists u \in R_1, v \in R_2 \ : \ w \text{ is a shuffle of } u, v \ \}.$$

This operator is used in the language Relax NG that is presented in Section 8.7. It is easy to see that the shuffle of two regular languages is again a regular language. Hence the expressive power of regular expressions does not increase if we add this operator. We denote the class of regular expressions extended with the operator $\|$ by $\mathsf{RE}_{\|}$.

This class of regular expressions has an interesting property. For most formalisms the membership problem is not harder (and in many cases easier) than the emptiness problem. For $\mathsf{RE}_{||}$ the situation is the other way round. The membership problem becomes harder but the complexity of the emptiness problem does not change because one can replace each operator $||$ by the standard concatenation while preserving non-emptiness.

Theorem 8.5.3. *The uniform membership problem for* $\mathsf{RE}_{||}$ *is NP-complete and the emptiness problem for* $\mathsf{RE}_{||}$ *is in PTIME.*

In the following subsections we have to explicitly refer to the representation used for the horizontal languages in hedge automata. For this we use notations like ***NFHA(NFA)***, denoting the class of nondeterministic hedge automata with horizontal languages represented by NFAs. Another example for this notation is DFHA(AFA) for the class of deterministic hedge automata with horizontal languages represented by AFAs.

8.5.2 Determinism and Completeness

To check if a given NFTA is deterministic one can simply compare all left-hand sides of transitions, which can clearly be done in time polynomial in the size of the automaton. For hedge automata the situation is different because we have to compare the intersections of the horizontal languages appearing in the transitions.

Theorem 8.5.4. *Checking whether a given NFHA(NFA) is deterministic can be done in polynomial time and is PSPACE-complete for NFHA(AFA).*

To check determinism can be interesting because complementing a deterministic automaton is easier than complementing a nondeterministic one. But to achieve complementation just by exchanging final and non-final states, the automaton also has to be complete. It is not difficult to see that testing for completeness is at least as hard as testing universality for the formalism used to represent the horizontal languages. Hence, the following result is not surprising.

Theorem 8.5.5. *Checking whether a given NFHA(NFA) or NFHA(AFA) is complete is PSPACE-complete.*

In the following subsections, when we estimate the complexity for (complete) automata from DFHA(NFA) or DFHA(AFA) one should keep in mind that testing whether the given automaton has the desired properties might increase the complexity.

8.5.3 Membership

The membership problem, i.e. the problem of deciding for a fixed hedge automaton $\mathcal{A}$ if a given tree t is accepted by $\mathcal{A}$ (compare Section 1.7), can easily be decided in linear time in the size of t: If $\mathcal{A}$ is fixed we can assume that it is deterministic and that all of its horizontal languages are given by DFAs. The run of $\mathcal{A}$ on t can then be constructed by a single bottom-up pass over the tree.

The problem is more interesting if the automaton is also part of the input. In this case it is called the uniform membership problem. In Figure 8.6 we give

```
Generic Algorithm for Uniform Membership
input: NFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, $t \in T(\Sigma)$
begin
    Set $\rho(p) = \bot$ for all $p \in \mathcal{P}os(t)$
    while ( $\exists p \in \mathcal{P}os(t)$ with $\rho(p) = \bot$ and $\rho(pi) \neq \bot$ for all
            $i \in \{1, \ldots, k\}$ with $k = \max\{i \mid pi \in \mathcal{P}os(t)$ or $i = 0\}$ )
        $M = \emptyset$
        for each $a(R) \to q \in \Delta$ with $a = t(p)$
            if $\exists q_1 \in \rho(p1), \ldots, q_k \in \rho(pk)$ with $q_1 \cdots q_k \in R$ then
                $M = M \cup \{q\}$
            endif
        endfor
        $\rho(p) = M$
    endwhile
    output: “yes” if $\rho(\varepsilon) \cap Q_f \neq \emptyset$, “no” otherwise
end
```

Figure 8.6: A generic algorithm for solving the uniform membership problem for hedge automata

a generic algorithm that constructs for a given NFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ and a tree t a mapping $\rho : \mathcal{P}os(t) \to 2^Q$ labeling each $p \in \mathcal{P}os(t)$ with the set M of those states that are reachable at p in a run of $\mathcal{A}$ on t. For this purpose, the algorithm picks a position p such that all its successor positions are already labeled by ρ and then checks for each transition $a(R) \to q$ with $a = t(p)$ if it can be applied to some string that is obtained by picking for each successor one state. Note that that the algorithm starts at the leaves because in this case $k = 0$.

The only thing that might be non-polynomial in this algorithm is testing the condition in the if-statement inside the foreach-loop. In a naive implementation one would have to test all possible combinations of sequences $q_1 \cdots q_k$ of states, which are exponentially many in general. But for some representations of the horizontal languages there are more efficient ways to check for the existence of a suitable sequence $q_1 \cdots q_k$.

Theorem 8.5.6. *The uniform membership problem for hedge automata has the following complexity, depending on the type of the given automaton:*

(1) For NFHA(NFA) it is in PTIME.
(2) For DFHA(DFA) it is solvable in linear time.
(3) For NFHA(AFA) it is NP-complete.
(4) For DFHA(AFA) it is in PTIME.
(5) For NFHA($\mathsf{RE}_{\|}$) it is NP-complete.

Proof. We use the generic algorithm for showing (1), (2), and (4). As already mentioned, we mainly have to pay attention to the condition tested in the if-statement.

If we are working with DFHA, then for each $p \in \mathcal{P}os(t)$ there is at most one state in $\rho(p)$. This means that there is at most one sequence $q_1 \cdots q_k$ to consider. From this observation we easily obtain (2), and also (4) in combination with the fact that the uniform membership problem for AFAs is solvable in polynomial time.

For (1) we just note that the condition in the if-statement can be decided in polynomial time for NFAs: One starts at the initial state of the NFA and then collects all states of the NFA that are reachable by using elements from $\rho(p1)$ as input, and then continues to compute all states reachable from there with inputs from $\rho(p2)$, and so on. If the last set contains a final state of the NFA, then the condition is satisfied.

To prove (3) we do not use the generic algorithm because we have to provide a nondeterministic one for the upper bound. It is rather easy to see that guessing an accepting run of $\mathcal{A}$ and then verifying it can be done in polynomial time because the uniform membership problem for AFAs is solvable in polynomial time.

We show the NP-hardness even for horizontal languages represented by intersections of regular expressions instead of the more general formalism of alternating automata. We use a reduction from the satisfiability problem for Boolean formulas.

Let $\varphi = c_1 \wedge \cdots \wedge c_m$ be such a formula in CNF using the variables $x_1, \ldots, x_n$, where the c_i are disjunctions of literals, i.e. variables or their negations. An assignment to the variables is coded as a string of length n over the alphabet $\{0, 1\}$, where 1 in position i codes that variable x_i is set to true and 0 that it is set to false. A literal x_j is now translated to a regular expression describing all strings of length n with 1 in position j. Correspondingly, the literal $\neg x_j$ is translated to an expression requiring 0 in position j of the string. A clause c_i corresponds to the regular expression e_i obtained as the union of the expressions for the literals in c_i. The expression e_φ, finally, corresponds to the intersection of all the e_i.

The NFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ uses a singleton alphabet $\Sigma = \{a\}$, three states $Q = \{0, 1, q\}$ with $Q_f = \{q\}$, and the transitions

$$a(\{\varepsilon\}) \to 0, \quad a(\{\varepsilon\}) \to 1, \quad a(e_\varphi) \to q.$$

Now it is rather easy to observe that $\mathcal{A}$ accepts the tree $a(\underbrace{a \cdots a}_{n})$ iff φ is satisfiable because an accepting run has to code a satisfying assignment for φ at the leaves of the tree.

Finally, (5) follows easily from the NP-completeness of the uniform membership problem for $\mathsf{RE}_{\|}$ (see Theorem 8.5.3). □

One should note here that the first two items of Theorem 8.5.6 can easily be obtained by using the extension encoding and the results from Section 1.7 on the uniform membership problem for ranked tree automata.

On the other hand, this approach is not optimal for (3) and (4). One might think that hedge automata with horizontal languages represented by alternating word automata can directly be translated to alternating tree automata working on encodings. But (3) from the above theorem indicates that this is not possible unless P=NP because the uniform membership problem for alternating tree automata is solvable in polynomial time (compare Section 7.5).

```
Generic Algorithm for Emptiness
input: NFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$
begin
    Set $M = \emptyset$
    while $\exists a(R) \to q \in \Delta$ with $q \notin M$ and $R|_M \neq \emptyset$
        $M = M \cup \{q\}$
    endwhile
    output: “yes” if $M \cap Q_f = \emptyset$, “no” otherwise
end
```

Figure 8.7: A generic algorithm for emptiness of hedge automata

Corollary 8.5.7. *There is no polynomial time translation from NFHA(AFA) to alternating tree automata on the FCNS or extension encoding, unless P=NP.*

This suggests that for going through encodings one first has to remove alternation from the horizontal languages, that is one has to convert the alternating finite automata into nondeterministic ones, involving an exponential blow-up. So in general, this procedure requires exponential space opposed to NP and polynomial time for (3) and (4). Similarly, a translation from $\mathsf{RE}_{||}$ expressions to automata involves an exponential blow-up.

Finally, we note that the case of NFHA(DFA) is polynomial because it is subsumed by NFHA(NFA), but it is not linear because of the nondeterminism in the tree automaton.

8.5.4 Emptiness

The emptiness problem is decidable in polynomial time for NFTAs (Section 1.7) and DEXPTIME-complete for alternating tree automata (Section 7.5). Here we analyze the problem for hedge automata and get, similar to Theorem 8.5.6, different results depending on the representation of the horizontal languages.

Figure 8.7 shows a generic algorithm deciding the emptiness for hedge automata using an oracle for the non-emptiness of (restrictions of) the horizontal languages: the condition in the while-loop uses the expression $R|_M \neq \emptyset$, where M is a subset of Q, the alphabet of R, and $R|_M$ simply denotes the restriction of R to the subalphabet M, i.e.

$$R|_M = \{w \in M^* \mid w \in R\},$$

where we use the convention $R|_\emptyset = \{\varepsilon\}$.

Apart from that, the algorithm works in the same way as the reduction algorithm for NFTAs from Section 1.1. It computes the set of all reachable states and then checks if there is a final state among them.

The complexity mainly depends on how fast the test $R|_M \neq \emptyset$ can be realized.

Theorem 8.5.8. *The emptiness problem for hedge automata has the following complexity, depending on the type of the given automaton:*

(1) For NFHA(NFA) it is in PTIME.

(2) For NFHA(AFA) it is PSPACE-complete.

(3) For NFHA($\mathsf{RE}_{\|}$) it is in PTIME.

Proof. The first two claims follow directly from the corresponding complexity for the test $R|_M \neq \emptyset$ in the while loop. For NFAs this can be done in polynomial time and for AFAs it is PSPACE-complete (Proposition 8.5.1).

The last claim follows from the fact that replacing in the $\mathsf{RE}_{\|}$-expressions each $\|$-operator by a standard concatenation results in an automaton that accepts the empty language iff the language of the original automaton is empty. Since standard regular expressions can be translated to NFAs in polynomial time, the claim follows from (1). ☐

One should note again that the first result can easily be obtained using the extension encoding, whereas for (2) this is not the case. First converting the alternating automata for the horizontal languages into nondeterministic can require exponential space.

8.5.5 Inclusion

The inclusion problem, given two hedge automata $\mathcal{A}_1$ and $\mathcal{A}_2$, decide whether $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$, can be solved by a very simple generic algorithm using the emptiness test from Section 8.5.4:

$$L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2) \text{ iff } L(\mathcal{A}_1) \cap (T(\Sigma) \setminus L(\mathcal{A}_2)) = \emptyset \ .$$

To solve the inclusion problem according to this scheme, we have to compute the complement, the intersection, and do the emptiness test.

Theorem 8.5.9. *The inclusion problem for hedge automata has the following complexity, depending on the type of the given automaton:*

(1) For DFHA(DFA) it is in PTIME.

(2) For NFHA(NFA) it is EXPTIME-complete.

(3) For DFHA(AFA) it is PSPACE-complete.

Proof. As a DFHA(DFA) can easily be completed and then complemented by exchanging final and non-final states, we obtain (1). *see errata*

The upper bound for (2) follows from (1) and from the fact that a NFHA(NFA) can be turned into a DFHA(DFA) of exponential size. The lower bound follows from the corresponding one for NFTAs from Section 1.7.

Finally, for (3) a DFHA(AFA) $\mathcal{A}$ can be made complete as follows. Let $a(R_1) \to q_1, \ldots, a(R_n) \to q_n$ be all transitions of $\mathcal{A}$ with label a. An AFA for $R = Q^* \setminus \bigcup_{i=1}^n R_i$ can be constructed in polynomial time from the AFAs for the languages R_i. Then we add a new rejecting sink state $q_\perp$ and the transition $a(R) \to q_\perp$. Proceeding like this for all labels from Σ yields a complete DFHA(AFA) that can be complemented by exchanging final and non-final states. The intersection of two DFHA(AFA) can be built by using a simple product construction. Now the upper bound follows from Theorem 8.5.8.

The lower bound is easily obtained because the emptiness problem for AFAs is PSPACE-complete. ☐

8.6 Minimization

In Section 1.5 we have seen that DFTAs have some good properties with respect to minimization: For each recognizable tree language there is a unique minimal DFTA recognizing it, which can be computed efficiently from a given DFTA for the language.

In this section we discuss the minimization problem for hedge automata. We consider the following three approaches:

1. Use encodings and minimization results for ranked tree languages.
2. Use hedge automata while only looking at the number of states and considering transitions $a(R) \to q$ as atomic units.
3. Use hedge automata with a definition of size that takes into account the representations of the languages R in transitions $a(R) \to q$.

The first approach certainly strongly depends on the chosen encoding. One way to justify the choice of a specific encoding is to relate the automata on this encoding to a natural model which is interpreted directly on the unranked trees. When we treat the third approach in Section 8.6.2 we see that this is indeed possible for the extension encoding. Before that we consider the second approach and just focus on the number of states.

8.6.1 Minimizing the Number of States

In Section 1.5 we have seen that for a recognizable language L of ranked trees the congruence $\equiv_L$ has finite index. The equivalence classes of $\equiv_L$ can be used as states in a canonical automaton, which is also the unique minimal DFTA for this language. Recall the definition of $t \equiv_L t'$ for ranked trees t, t':

$$\forall C \in \mathcal{C}(\mathcal{F}) : \; C[t] \in L \Leftrightarrow C[t'] \in L.$$

This definition can easily be adapted to unranked trees. For this purpose, we use $\mathcal{C}(\Sigma)$ to denote the set of unranked trees with exactly one leaf labeled by a variable. For $C \in \mathcal{C}(\Sigma)$ and $t \in T(\Sigma)$ we denote by $C[t]$ the unranked tree obtained by substituting t for the variable in C (in the same way as for ranked trees).

Given a language $L \subseteq T(\Sigma)$, the definition of $\equiv_L$ is then exactly the same as for ranked trees:

$$t \equiv_L t' \text{ iff } \forall C \in \mathcal{C}(\Sigma) : \; C[t] \in L \Leftrightarrow C[t'] \in L.$$

Using the equivalence classes of $\equiv_L$ we can construct a minimal DFHA. To ensure that it is unique we have to require that it is normalized because otherwise one can always split a transition into two without changing the behavior of the automaton.

For the formulation of the following theorem we say that two DFHAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are the same up to renaming of states if there is a bijection f between the two sets of states that respects final states and transitions: q is a final state of $\mathcal{A}_1$ iff $f(q)$ is a final state of $\mathcal{A}_2$, and $a(R) \to q$ is a transition of $\mathcal{A}_1$ iff $a(f(R)) \to f(q)$ is a transition of $\mathcal{A}_2$.

Theorem 8.6.1. *For each recognizable language $L \subseteq T(\Sigma)$ there is a unique (up to renaming of states) normalized DFHA with a minimal number of states.*

Proof. Let $L \subseteq T(\Sigma)$ be recognizable. For $t \in T(\Sigma)$ we denote the $\equiv_L$-class of t by $[t]$. We define the components of $\mathcal{A}_{min}$ as follows. Let $Q_{min} = \{[t] \mid t \in T(\Sigma)\}$, $Q_{min_f} = \{[t] \mid t \in L\}$. The transition relation Δ_{min} contains the transitions $a(R_{a,[t]}) \rightarrow [t]$ with

$$R_{a,[t]} = \{[t_1] \cdots [t_n] \mid a(t_1 \cdots t_n) \equiv_L t\}.$$

This transition relation is deterministic because $\equiv_L$ is an equivalence relation.

To show that the sets $R_{a,[t]}$ are regular consider a normalized DFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ accepting L with transitions of the form $a(R_{a,q}) \rightarrow q$. For a state $q \in Q$ let

$$[q] = \{t \in T(\Sigma) \mid t \xrightarrow[\mathcal{A}]{*} q\}.$$

From the definition of $\equiv_L$ we obtain that all trees from $[q]$ are $\equiv_L$-equivalent, i.e. for each q there exists a t with $[q] \subseteq [t]$. Then we can write the sets $R_{a,[t]}$ as

$$\begin{aligned} R_{a,[t]} = \{[t_1] \cdots [t_n] \mid\ & \exists q \in Q \text{ with } [q] \subseteq [t] \text{ and } q_1 \cdots q_n \in R_{a,q} \\ & \text{with } [q_1] \subseteq [t_1], \ldots, [q_n] \subseteq [t_n]\}\ . \end{aligned}$$

As the sets $R_{a,q}$ are regular we can derive from this description that also the sets $R_{a,[t]}$ are regular.

For the correctness of the construction one can easily show that $t \xrightarrow[\mathcal{A}_{min}]{*} [t']$ iff $t \equiv_L t'$.

As mentioned above, if q is a state of a DFHA for L, then $[q] \subseteq [t]$ for some $t \in T(\Sigma)$. From this one can easily derive that $\mathcal{A}_{min}$ is indeed unique up to renaming of states. ☐

However, this result is not sufficient for practical purposes because it does not take into account the size for representing the horizontal languages. Furthermore, the complexity of computing the minimal automaton from a given one by merging equivalent states strongly depends on the formalism used to represent the horizontal languages. If the horizontal languages are given by regular expressions, for example, it is easy to code the equivalence problem for regular expressions into the question of deciding whether a given DFHA is minimal. As the former problem is PSPACE-hard, the latter one is also at least PSPACE-hard.

Furthermore, the congruence $\equiv_L$ does not yield a Myhill-Nerode-like theorem as presented in Section 1.5. The following example illustrates this by providing a language L that is not recognizable but for which $\equiv_L$ is of finite index.

Example 8.6.2. Consider the set $L = \{a(b^n c^n) \mid n \in N\}$ of unranked trees. The number of equivalence classes of $\equiv_L$ for this set is 4: the language L forms one class, the trees b and c form classes of size one, respectively, and all the remaining trees the fourth class. But certainly L is not recognizable because $b^n c^n$ is not a regular language.

The example shows that regularity on the horizontal level is not captured by the congruence $\equiv_L$. One might be tempted to think that it is enough to require for each label $a \in \Sigma$, that the set of words occurring as successor word below the label a in some tree of the language is regular. Indeed, this would exclude the above example but it is not difficult to find a non-regular language of trees for which $\equiv_L$ is of finite index and this additional condition is also met (see Exercises).

At the end of Section 8.6.3 we give a refinement of $\equiv_L$ that is sufficient to characterize the recognizable languages.

8.6.2 Problems for Minimizing the Whole Representation

Minimization becomes a more complex task if we also want to consider the size of the representations of the transitions. It might be that adding additional states to the hedge automaton allows splitting transitions such that the representations for the required horizontal languages become smaller.

If we are interested in unique representations while taking into account also the horizontal languages, then we should certainly choose a formalism that allows unique representations for the horizontal languages. Therefore we only consider DFHAs with horizontal languages given by deterministic finite automata in the following.

But even with this assumption it turns out that there are no unique minimal DFHAs for a given recognizable language. An exact statement of this negative result, however, would require a precise definition of the size of DFHAs. There are various reasonable such definitions, and to prove that minimal DFHAs for a given language are not unique in general, it would be necessary to show this for all these definitions.

Here, we restrict ourselves to an explanation of the main reason why DFHAs are problematic with respect to minimization. In Section 8.6.3 we then introduce a model that solves this problem.

Example 8.6.3. We consider the language containing the trees of the form $a(b^n)$ with $n \bmod 2 = 0$ or $n \bmod 3 = 0$, i.e. trees of height 1 with a at the root and b at all the successors, where the number of successors divides 2 or 3. A DFHA recognizing this language can use two states q_a and q_b with rules $b(\{\varepsilon\}) \rightarrow q_b$ and $a(L) \rightarrow q_a$ with $L = \{q_b^n \mid n \bmod 2 = 0 \text{ or } n \bmod 3 = 0\}$. The minimal deterministic automaton (over the singleton alphabet $\{q_b\}$, which is sufficient for the language under consideration) for L has 6 states because it has to count modulo 6 for verifying if one of the condition holds.

It is also possible to split the second transition into two transitions: $a(L_2) \rightarrow q_a$ and $a(L_3) \rightarrow q_a$ with $L_2 = \{q_b^n \mid n \bmod 2 = 0\}$ and $L_3 = \{q_b^n \mid n \bmod 3 = 0\}$. The two automata for L_2 and L_3 need only 2 and 3 states, respectively. But in exchange the tree automaton has one transition more.

If we take as size of a DFHA the number of states and the number of transitions of the tree automaton plus the number of states used in the horizontal languages, then the two DFHAs from above are both minimal for the given language but they are clearly non-isomorphic.

This example illustrates that the proposed model is not fully deterministic. We have chosen the horizontal automata to be deterministic but to know which automaton to apply we have to know the whole successor word.

One should note here that in the example we did not require the automata to be normalized. But even this additional restriction is not sufficient to guarantee unique minimal automata. The interested reader is referred to the bibliographic notes.

8.6.3 Stepwise automata

We have seen that the first model of deterministic automata that we defined is not fully deterministic on the horizontal level because there may be different choices for the horizontal automaton to apply, or we have to know the full successor word to decide which horizontal automaton to use.

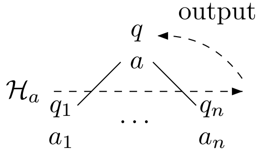
We now present a way to overcome this problem. We start from a rather intuitive model that uses deterministic automata with output at the sequence of successors. By applying simple transformations we finally end up with a model that uses only one sort of states and is tightly connected to automata on encoded trees (see Section 8.3).

We start by explaining the models on an intuitive level and give formal definitions in the end.

We start with a model that works as follows:

- Transitions are represented by deterministic automata with output, one automaton $\mathcal{H}_a$ for each letter a from Σ. We refer to these automata as 'horizontal automata'.
- For a node labeled a, the automaton $\mathcal{H}_a$ reads the sequence of states at the successor nodes.
- After reading this sequence the automaton outputs an element from Q.

This is illustrated in the following picture.



Example 8.6.4. We consider the set of all unranked trees over the alphabet $\Sigma = \{a, b, c, d\}$ such that below each a there exists a subtree containing at least two b, and below each d there exist at least two subtrees containing at least one b.

A hedge automaton for this language can be implemented using states q, q_b, q_{bb} indicating the number of b in the subtree (all of them are final states), and a rejecting sink state $q_\perp$.

If a node is found where one of the conditions is not satisfied, then the automaton moves to $q_\perp$.

Figure 8.8 shows the transition function of the automaton, represented by deterministic automata with output. The arrow pointing to the initial state

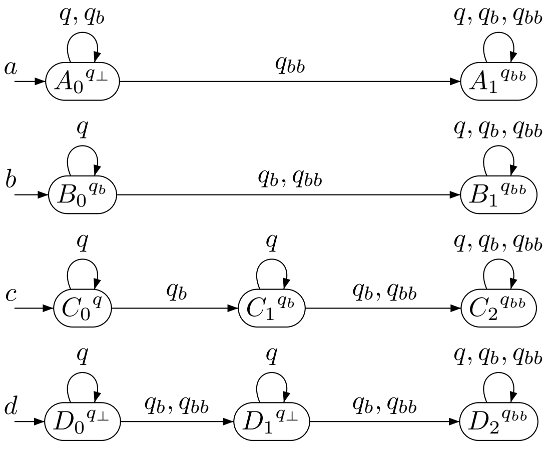


Figure 8.8: Transition function of the automaton from Example 8.6.4 *see errata*

is labeled with a letter from Σ indicating for which nodes in the tree which automaton has to be used. The output at each state is written in the upper right corner. For example, the output at state C_1 is q_b. For better readability we did not specify the transitions for $q_\perp$. In each of the four automata a sink state has to be added to which the automaton moves as soon as $q_\perp$ is read. The output at these sink states is $q_\perp$.

Figure 8.9 shows a run of this automaton on an input tree. Directly above the labels of the tree the state of the tree automaton is written. The states of the horizontal automata are obtained as follows. Consider the bottom left leaf of the tree. It is labeled by b. So we have to use the automaton with the initial arrow marked with b. This automaton now reads the sequence of states below the leaf, i.e. the empty sequence. It starts in its initial state B_0. As there is nothing to read, it has finished. We take the output q_b of B_0 and obtain the state of the hedge automaton at this leaf.

Now consider the node above this leaf, it is labeled by c. So we have to start the horizontal automaton with initial state C_0. It reads the state q_b that we have just computed and moves to C_1. Then it has finished and produces the output q_b. In this way we can complete the whole run.

A first thing to note about this model is that there is no reason to keep the automata $\mathcal{H}_a$ separate. Looking at the example one quickly realizes that the states A_1, B_1, C_2, D_2 are all equivalent in the sense that they produce the same output and behave in the same way for each input. So instead of having four automata we could consider them as one automaton with four initial states. This allows reducing the representation by merging equivalent states.

Next, we can note that this type of automaton uses two different sets of states, those of the hedge automaton and those of the horizontal automaton. But there is a direct relationship between these two different types of states, namely the one given by the output function: each state of the horizontal automaton is mapped to a state of the hedge automaton. If we replace each state q of the hedge automaton by its inverse image under this output function, i.e. by the set

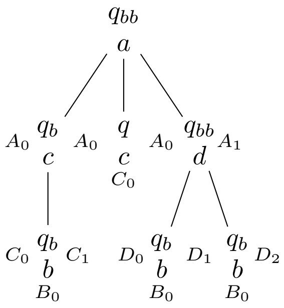


Figure 8.9: A run using the transition function from Figure 8.8

of states of the horizontal automaton that produce q as output, then we obtain a model using only one type of states.

The result of these operations applied to the example is shown in Figure 8.10. The state BB is obtained by merging A_1, B_1, C_2, D_2, and C_1 is merged into B_0. Then the output function is removed. One can think of each state outputting itself. On the transitions, state q is replaced by C_0, q_b by B_0, q_{bb} by BB, and $q_\perp$ by A_0, D_0, D_1. The latter means that the transitions to the sink (not shown in the picture) are now labeled by A_0, D_0, and D_1 instead of just $q_\perp$. The final states are B_0, BB, C_0, i.e. all those states that were mapped to a final state of the hedge automaton before.

On the right-hand side of Figure 8.10 it is shown how this new automaton processes a tree. Consider, for example, the front of the right subtree. Both leaves are labeled by b and hence are assigned state B_0. Now the automaton processes this sequence B_0B_0. As the node above is labeled d, it starts in state D_0. On reading B_0 it moves to state D_1. On reading the next B_0 it moves to BB. This is the state assigned to the node above in the tree.

Another example on how the state sequence of the successors is processed is directly below the root: scanning the sequence B_0C_0BB leads the automaton from A_0 to BB, which is then assigned to the root.

We now give a formal definition of the model we have just derived. It is basically a word automaton that has its own state set as input alphabet and a function assigning to each letter of the tree alphabet an initial state.

A ***deterministic stepwise hedge automaton*** (DSHA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta_0, Q_f, \delta)$, where Q, Σ, and Q_f are as usual, $\delta_0 : \Sigma \to Q$ is a function assigning to each letter of the alphabet an initial state, and $\delta : Q \times Q \to Q$ is the transition function.

To define how such an automaton works on trees we first define how it reads sequences of its own states. For $a \in \Sigma$ let $\delta_a : Q^* \to Q$ be defined inductively by $\delta_a(\varepsilon) = \delta_0(a)$, and $\delta_a(wq) = \delta(\delta_a(w), q)$. This corresponds to the view of $\mathcal{A}$ as a word automaton reading its own states.

For a tree t and a state q of $\mathcal{A}$ we define the relation $t \xrightarrow[\mathcal{A}]{*} q$ as follows. Let $t = a(t_1 \cdots t_n)$ and $t_i \xrightarrow[\mathcal{A}]{*} q_i$ for each $i \in \{1, \ldots, n\}$. Then $t \xrightarrow[\mathcal{A}]{*} q$ for $q = \delta_a(q_1 \cdots q_n)$. For $n = 0$ this means $q = \delta_0(a)$.

We have given an intuitive explanation on how to obtain a stepwise automaton for a recognizable language. To formally prove that there indeed is a

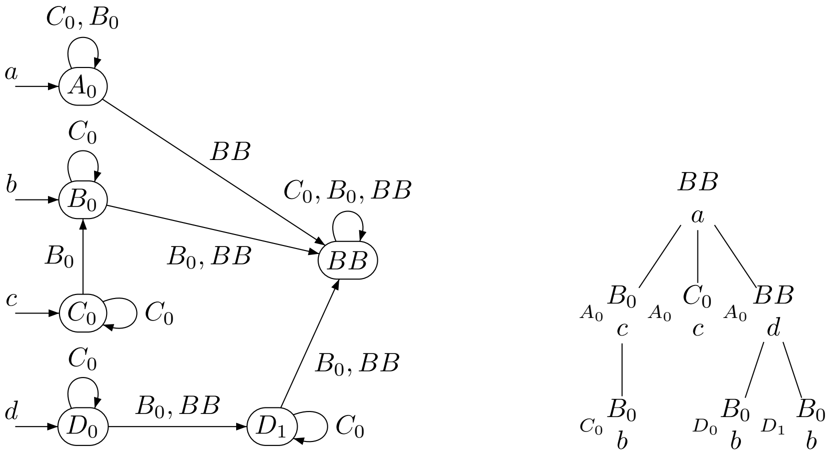


Figure 8.10: Model using only one sort of states

see errata

stepwise automaton for each recognizable language we go through the extension encoding from Subsection 8.3.2.

For this purpose we first analyze how the transitions of a DSHA behave with respect to the extension operator. Our aim is to establish the following simple rules to switch between DSHAs and DFTAs on the extension encodings:

ranked	stepwise
$a \to q$	$\delta_0(a) = q$
$@(q_1, q_2) \to q$	$\delta(q_1, q_2) = q$

The key lemma needed for this is the following.

Lemma 8.6.5. *Let* $\mathcal{A} = (Q, \Sigma, \delta_0, Q_f, \delta)$ *be a DSHA and* $t, t' \in T(\Sigma)$ *with* $t \xrightarrow[\mathcal{A}]{*} q$ *and* $t' \xrightarrow[\mathcal{A}]{*} q'$ *for* $q, q' \in Q$. *Then* $t \mathbin{@} t' \xrightarrow[\mathcal{A}]{*} \delta(q, q')$.

Proof. Let $t = a(t_1 \cdots t_n)$ with $t_i \xrightarrow[\mathcal{A}]{*} q_i$. Then $t \xrightarrow[\mathcal{A}]{*} q$ means that $\delta_a(q_1 \cdots q_n) = q$. Further, we have $t \mathbin{@} t' = a(t_1 \cdots t_n t')$ and

$$\delta_a(q_1 \cdots q_n q') = \delta(\delta_a(q_1 \cdots q_n), q') = \delta(q, q').$$

Hence, we obtain $t \mathbin{@} t' \xrightarrow[\mathcal{A}]{*} \delta(q, q')$. □

This allows us to view a stepwise automaton as an automaton on the extension encodings and vice versa. If $\mathcal{A} = (Q, \Sigma, \delta_0, Q_f, \delta)$ is a DSHA, then we refer to the corresponding DFTA on the extension encoding as $\mathsf{ext}(\mathcal{A})$, formally defined as $\mathsf{ext}(\mathcal{A}) = (Q, \mathcal{F}^{\Sigma}_{\mathsf{ext}}, Q_f, \Delta)$ with Δ defined according to the above table by

- $a \to q$ in Δ if $\delta_0(a) = q$, and
- $@(q_1, q_2) \to q$ in Δ if $\delta(q_1, q_2) = q$.

Using Lemma 8.6.5 it is not difficult to show that this construction indeed transfers the language in the desired way.

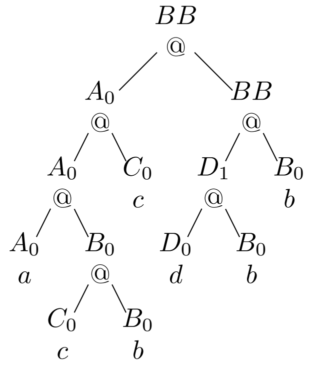


Figure 8.11: A run on the extension encoding corresponding to the run in Figure 8.10

Theorem 8.6.6. *Let $\mathcal{A}$ be a DSHA. Then* $\mathsf{ext}(L(\mathcal{A})) = L(\mathsf{ext}(\mathcal{A}))$.

Proof. For simplicity, we denote $\mathsf{ext}(\mathcal{A})$ by $\mathcal{B}$. We show by induction on the number of nodes of $t \in T(\Sigma)$ that

$$t \xrightarrow[\mathcal{A}]{*} q \Leftrightarrow \mathsf{ext}(t) \xrightarrow[\mathcal{B}]{*} q.$$

For trees with only one node this is obvious from the definition: We have $a \xrightarrow[\mathcal{A}]{*} q$ iff $\delta_0(a) = q$ iff $a \rightarrow q$ is a rule of $\mathcal{B}$ iff $a \xrightarrow[\mathcal{B}]{*} q$.

If t has more than one node, then $t = t' \mathbin{@} t''$ for some t', t''. The definition of ext yields that $\mathsf{ext}(t) = @(\mathsf{ext}(t'), \mathsf{ext}(t''))$.

Applying the induction hypothesis we obtain that $t' \xrightarrow[\mathcal{A}]{*} q' \Leftrightarrow \mathsf{ext}(t') \xrightarrow[\mathcal{B}]{*} q'$ and $t'' \xrightarrow[\mathcal{A}]{*} q'' \Leftrightarrow \mathsf{ext}(t'') \xrightarrow[\mathcal{B}]{*} q''$.

If $\mathsf{ext}(t') \xrightarrow[\mathcal{B}]{*} q'$ and $\mathsf{ext}(t'') \xrightarrow[\mathcal{B}]{*} q''$, then $\mathsf{ext}(t) \xrightarrow[\mathcal{B}]{*} q$ means that $@(q', q'') \rightarrow q$ is a rule of $\mathcal{B}$. So in $\mathcal{A}$ we have $\delta(q', q'') = q$ and we conclude $t \xrightarrow[\mathcal{A}]{*} q$ from Lemma 8.6.5. ☐

Example 8.6.7. Figure 8.11 shows the run on the extension encoding corresponding to the right-hand side of Figure 8.10.

A simple consequence is that DSHAs are sufficient to accept all recognizable unranked tree languages. But we can furthermore use the tight connection between DFTAs working on extension encodings and stepwise automata that is established in Theorem 8.6.6 to transfer the results on minimization from Section 1.5 to stepwise automata.

Theorem 8.6.8. *For each recognizable language $L \subseteq T(\Sigma)$ there is a unique (up to renaming of states) minimal DSHA accepting L.*

Proof. Starting from a stepwise automaton $\mathcal{A}$ for L we consider the DFTA $\mathcal{B} := \mathsf{ext}(\mathcal{A})$. Using the results from Section 1.5 we know that there is a unique minimal DFTA $\mathcal{B}_{min}$ equivalent to $\mathcal{B}$. We then define $\mathcal{A}_{min} := \mathsf{ext}^{-1}(\mathcal{B}_{min})$. From Theorem 8.6.6 we conclude that $\mathcal{A}_{min}$ is the unique minimal DSHA equivalent to $\mathcal{A}$. ☐

In Section 8.6.1 we have introduced the congruence $\equiv_L$ to characterize the state-minimal DFHA for the language L. We have also seen that there are non-recognizable languages for which this congruence has finite index.

In the following we show that there is another congruence characterizing recognizable sets of unranked trees as those sets for which this congruence has finite index.

To achieve this we consider congruences w.r.t. the extension operator from Section 8.3. We say that an equivalence relation $\equiv$ on $T(\Sigma)$ is an @-congruence if the following holds:

$$t_1 \equiv t_2 \text{ and } t_1' \equiv t_2' \Rightarrow t_1 \mathbin{@} t_2 \equiv t_1' \mathbin{@} t_2' \; .$$

It is easy to see that an @-congruence on $T(\Sigma)$ corresponds to a congruence on $T(\mathcal{F}^{\Sigma}_{\mathsf{ext}})$ via the extension encoding and vice versa. Thus, we directly obtain the following result.

Theorem 8.6.9. *A language $L \subseteq T(\Sigma)$ is recognizable if and only if it is the union of equivalence classes of a finite @-congruence.*

For a recognizable language $L \subseteq T(\Sigma)$, the congruence $\equiv_{\mathsf{ext}(L)}$ can be used to characterize the minimal DFTA for $\mathsf{ext}(L)$. By the tight connection between DSHAs and DFTAs on the extension encoding, we obtain that the minimal DSHA can be characterized by the @-congruence $\equiv^{@}_{L}$ defined as

$$t_1 \equiv^{@}_{L} t_2 \text{ iff } \mathsf{ext}(t_1) \equiv_{\mathsf{ext}(L)} \mathsf{ext}(t_2) \; .$$

For a direct definition of $\equiv^{@}_{L}$ see Exercise 8.9.

8.7 XML Schema Languages

The nested structure of XML documents can be represented by trees. Assume, for example, that an organizer of a conference would like to store the scientific program of the conference as an XML document to make it available on the web. In Figure 8.12 a possible shape of such a document is shown and in Figure 8.13 the corresponding tree (the tree only reflects the structure without the actual data from the document).

Some requirements that one might want impose on the structure of the description of a conference program are:

- The conference might be split into several tracks.
- Each track (or the conference itself if it is not spilt into tracks) is divided in sessions, each consisting of one or more talks.
- For each session there is a session chair announcing the talks and coordinating the discussion.

```
<conference>
  <track>
    <session>
       <chair> F. Angorn </chair>
       <talk>
         <title> The Pushdown Hierarchy </title>
         <speaker> D.J. Gaugal </speaker>
       </talk>
       <talk>
         <title> Trees Everywhere </title>
         <authors> B. Aum, T. Rees  </authors>
       </talk>
    </session>
    <break> Coffee </break>
    <session>
       ....
    </session>
  </track>
  <track>
    ....
  </track>
</conference>
```

Figure 8.12: Possible shape of an XML document for a conference program

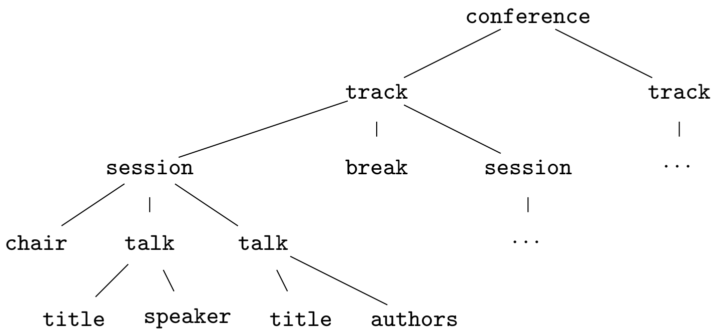

conference
track
track
session
break
session
...
chair
talk
talk
...
title
speaker
title
authors

Figure 8.13: The tree describing the structure of the document from Figure 8.12

- A talk might be contributed, in which case the title and the authors of the contribution are listed, or invited, in which case the title and the speaker are listed.

- Between sessions there might be breaks.

These requirements describe a tree language over the alphabet consisting of the tags like `conference`, `track`, etc. used in the document. In these considerations the data is usually ignored and only the structure of the document is taken into account. Such a description of a tree language is also called a schema. A document is said to be valid w.r.t. a schema if it (seen as a tree) belongs to the tree language defined by the schema.

There are several languages, called XML schema languages, that allow to describe such requirements on XML documents, and in the following we discuss some of them. This should not be considered as a reference for the syntax and semantics of these schema languages but we rather focus on the theoretical aspects and the connections to automata theory. In particular, we are not considering anything connected to the actual data represented in the documents, as e.g. data types, because we are, as already mentioned, only interested in the structure of the documents.

In particular, we are interested in the expressive power of these languages. Furthermore, we consider the decision problems from Section 8.5 because they correspond to natural questions on XML documents.

- The membership problem corresponds to check whether a given document is valid w.r.t. to a fixed schema. For example, checking whether an HTML document conforms to the specification is an instance of such a problem.

- The uniform membership problem corresponds to validation of a document against some given schema. The schema might for example be specified in the header of the document.

- The emptiness problem corresponds to the question whether for a schema there is any tree valid w.r.t. this schema. This can one the one hand be used for sanity checks on schema descriptions but also as subproblem for solving other questions like the inclusion problem.

- The inclusion problem is the question whether all documents that are valid w.r.t. one schema are also valid w.r.t. another schema. This is, e.g., of interest when merging archives of documents that are valid w.r.t. different schemas.

The practical formalisms that we present in the following all have a syntax that is closer to tree grammars than to hedge automata (compare Chapter 2 for ranked tree grammars). But it is convenient to use hedge automata as a reference model for the expressiveness, and to use the results from Section 8.5 to obtain algorithms and complexity results for the different formalisms in a uniform framework. Furthermore, the knowledge on hedge automata (and word automata) has certainly influenced the development of the formalisms presented here.

8.7.1 Document Type Definition (DTD)

A Document Type Definition (DTD) basically is a context-free (CF) grammar with regular expressions on the right-hand sides of the rules. The tree language defined by the DTD consists of all the derivation trees of this grammar. In Chapter 2 it is shown that for a standard CF grammar the set of derivation trees forms a regular tree language. This easily extends to the unranked setting.

An attempt to formalize by a DTD the requirements on documents describing conference programs that are listed in the introduction to this section could look as follows

```
<!DOCTYPE CONFERENCE [
    <!ELEMENT conference      (track+|(session,break?)+)>
    <!ELEMENT track           (session,break?)+>
    <!ELEMENT session         (chair,talk+)>
    <!ELEMENT talk            ((title,authors)|(title,speaker))>
    <!ELEMENT chair           (#PCDATA)>
    <!ELEMENT break           (#PCDATA)>
    <!ELEMENT title           (#PCDATA)>
]>
```

The symbols used in the expressions on the right-hand side of the rules are interpreted as follows:

$$\begin{array}{lcl@{\qquad}lcl} | & = & \text{choice} & + & = & \text{one or more occurrences} \\ , & = & \text{sequence} & ? & = & \text{one or zero occurrences} \end{array}$$

All the other element declarations that are missing (for authors, speaker, etc.) are assumed to be `#PCDATA` ("parsed character data"), which for our purpose simply denotes arbitrary character sequences coding the data inside the document.

The corresponding grammar looks as follows:

$$\begin{array}{lcl} \text{conference} & \rightarrow & \text{track}^{+} + (\text{session } (\text{break} + \varepsilon))^{+} \\ \text{track} & \rightarrow & (\text{session } (\text{break} + \varepsilon))^{+} \\ \text{session} & \rightarrow & \text{chair talk}^{+} \\ \text{talk} & \rightarrow & (\text{title authors}) + (\text{title speaker}) \\ \text{chair} & \rightarrow & \text{DATA} \\ \text{break} & \rightarrow & \text{DATA} \\ \text{title} & \rightarrow & \text{DATA} \end{array}$$

In the terminology of XML the right-hand side of a rule is also called the ***content model***.

In this setting a ***DTD*** is tuple $D = (\Sigma, s, \delta)$ with a start symbol $s \in \Sigma$ and a mapping δ assigning to each symbol from Σ a regular expression over Σ.

The easiest way to define the language generated by a DTD is to interpret it as a hedge automaton with the set of states being the same as the alphabet: Let $\mathcal{A}_D = (Q_D, \Sigma, Q_{Df}, \Delta_D)$ be defined by $Q = \Sigma$, $Q_{Df} = \{s\}$, and

$$\Delta_D = \{a(\delta(a)) \rightarrow a \mid a \in \Sigma\}.$$

It is obvious that $\mathcal{A}_D$ is a DFHA. We define the language $L(D)$ defined by D to be the language $L(\mathcal{A}_D)$.

The languages defined by DTDs are also called local languages because for a tree t the membership of t in $L(D)$ can be decided just by looking all subpatterns of height 1 (compare also Exercise 2.5 in Chapter 2 for local languages of ranked trees).

It is not difficult to see that these local languages form a proper subclass of the recognizable languages. Even the finite language containing the two trees $a(b(cd))$ and $a'(b(dc))$ cannot be defined by a DTD because every DTD D such that $L(D)$ contains these two trees would also contain the tree $a(b(dc))$.

Remark 8.7.1. There are recognizable languages of unranked trees that cannot be defined by DTDs.

This illustrates the weakness of DTDs compared to hedge automata. But in fact, DTDs are even more restrictive.

Deterministic Content Models

The following quote is taken from the web site of the W3C recommendation for XML:

> For Compatibility, it is an error if the content model allows an element to match more than one occurrence of an element type in the content model.

Informally, this means that, for a given word, we can uniquely match each symbol in this word to an occurrence of the same symbol in the content model by a single left to right pass without looking ahead in the input. This rather vague description is formalized using the notion of deterministic regular expression.

Consider the line

```
<!ELEMENT talk          ((title,authors)|(title,speaker))>
```

from the example. The content model is a regular expression of the type $ab+ac$, which is not deterministic because when reading the first a it is not clear to which a in the regular expression it corresponds. This depends on whether the next letter is b or c.

To formally define deterministic regular expressions (also called 1-unambiguous in the literature) we use markings of regular expressions by adding unique subscripts to each occurrence of letters in the expression: the ith occurrence (from left to right) of a in the expression is replaced by a_i.

Example 8.7.2. The marking of the expression $(a+b)^*b(ab)^*$ is $(a_1+b_1)^*b_2(a_2b_3)^*$.

For a regular expression e over the alphabet Γ we denote by e' the corresponding marked expression, and the marked alphabet by Γ'. As usual, $L(e)$ and $L(e')$ are the word languages defined by these expressions. Note that $L(e')$ defines a language over Γ'.

A regular expression e over some alphabet Γ is called ***deterministic*** if for all words u, v, w over Γ' and all symbols $x_i, y_j \in \Gamma'$ with $ux_iv, uy_jw \in L(e')$ we have that $x_i \neq y_j$ implies $x \neq y$.

Example 8.7.3. The expression $e = ab + ac$ is not deterministic because a_1b_1 and a_2c_1 are in $L(e')$ and $a_1 \neq a_2$ but $a = a$. On the other hand, the equivalent expression $a(b + c)$ is deterministic.

We call a DTD with deterministic content models a ***deterministic DTD***.

In the following we mention some results on deterministic regular expressions without giving proofs. The corresponding references can be found in the bibliographic notes.

The main interest in having deterministic regular expressions is that they can easily be translated to DFAs.

Theorem 8.7.4. *For each deterministic regular expression e one can construct in polynomial time a DFA accepting the language $L(e)$.*

This implies that we can construct in polynomial time a DFHA(DFA) accepting the language defined by the DTD. Hence, all the decision problems considered in Section 8.5 can be solved in polynomial time.

Proposition 8.7.5. *The uniform membership problem, the emptiness problem, and the inclusion problem for deterministic DTDs are solvable in polynomial time.*

This raises the question whether deterministic DTDs have the same expressive power as DTDs without this restriction. The following results imply that this is not the case. But it is decidable whether a DTD is equivalent to a deterministic one.

Theorem 8.7.6.

1. *Not every regular language can be defined by a deterministic regular expression.*
2. *Given a DFA one can decide in polynomial time if there is a deterministic regular expression denoting the same language. If such an expression exists it can be constructed in exponential time.*

Extended DTDs

We have seen that the main restriction for DTDs is that they can only define local languages. Extended DTDs use types to avoid that problem. For this we consider the alphabet $\widehat{\Sigma} = \{a^{(n)} \mid a \in \Sigma, n \in N\}$, which is called alphabet of types.

An ***extended DTD*** (EDTD) over Σ is a DTD over (a finite subset of) $\widehat{\Sigma}$, the alphabet of types. A tree $t \in T(\Sigma)$ satisfies an EDTD D if there is an assignment of types to the labels of t such that the resulting tree over $\widehat{\Sigma}$ satisfies D viewed as DTD over $\widehat{\Sigma}$.

Example 8.7.7. We modify the conference example from the beginning of this section. We replace the session tag with two new tags, one for invited sessions and one for contributed sessions, and require that invited sessions have exactly one invited talk and contributed sessions have a sequence of contributed talks. But we do not introduce different tags for the different types of talks. An invited talk is still characterized by the fact that it consists of a title and

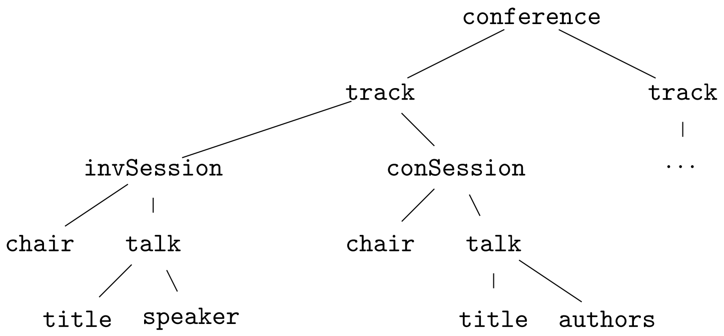


Figure 8.14: An example for a document structure satisfying the XML schema for the extended example

a speaker, where a contributed talk consists of a title and the authors. An example document with the new tags is shown in Figure 8.14.

The new requirement cannot be expressed by a DTD anymore because this would require to distinguish two rules (talk → title speaker) and (talk → title authors), depending on whether the talk is below a tag for an invited session or a contributed session. To resolve this problem one would also have to introduce two new tags for distinguishing invited talks and contributed talks.

To capture the modification that invited sessions only contain a single invited talk and that contributed sessions do not contain invited talks, we can define an EDTD using two types for the talks:

$$\begin{array}{lcl}
\text{conference} & \rightarrow & \text{track}^{+}+((\text{invSession}+\text{conSession})\ (\text{break}+\varepsilon))^{+} \\
\text{track} & \rightarrow & ((\text{invSession}+\text{conSession})\ (\text{break}+\varepsilon))^{+} \\
\text{invSession} & \rightarrow & \text{chair talk}^{(1)} \\
\text{conSession} & \rightarrow & \text{chair } (\text{talk}^{(2)})^{+} \\
\text{talk}^{(1)} & \rightarrow & \text{title speaker} \\
\text{talk}^{(2)} & \rightarrow & \text{title authors} \\
 & \cdots &
\end{array}$$

We have omitted the types for the other tags because only one type is used for them

It is not difficult to see that the types can be used to code the states of a hedge automaton. The proof of this statement is left as an exercise.

Proposition 8.7.8. *A language $L \subseteq T(\Sigma)$ with the property that all trees in L have the same root label is recognizable if and only if it can be defined by an EDTD over Σ.*

The conversion from EDTDs to NFHA(NFA) is polynomial and hence we obtain the following.

Proposition 8.7.9. *The uniform membership and the emptiness problem for EDTDs are solvable in polynomial time.*

In the next subsection we use EDTDs to describe the expressive power of XML Schema.

8.7.2 XML Schema

XML Schema is a formalism close to EDTDs. We present here only small code fragments to show how the extension of the conference example by invited sessions and contributed sessions from Example 8.7.7 can be formalized. The main point is that in XML schema it is possible to define types and that elements (that is, the tree labels) are allowed to have the same name but a different type in different contexts, as for EDTDs.

Lets come back to Example 8.7.7. With the EDTD from the example in mind, the syntax should be self explanatory. We directly start with the definition of tracks, the definition for the conference is similar.

```
<xsd:complexType name="track">
  <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:choice>
      <xsd:element name="invSession" type="invSession"
                   minOccurs="1" maxOccurs="1"/>
      <xsd:element name="conSession" type="conSession"
                   minOccurs="1" maxOccurs="1"/>
    </xsd:choice>
    <xsd:element name="break" type="xsd:string"
                 minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

The two types of sessions can be defined as follows.

```
<xsd:complexType name="invSession">
  <xsd:sequence>
    <xsd:element name="chair" type="xsd:string"
                 minOccurs="1" maxOccurs="1"/>
    <xsd:element name="talk" type="invTalk"
                 minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="conSession">
  <xsd:sequence>
    <xsd:element name="chair" type="xsd:string"
                 minOccurs="1" maxOccurs="1"/>
    <xsd:element name="talk" type="conTalk"
                 minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Note that in both definitions we use the name `talk` for the element but we assign different types. The types `invTalk` and `conTalk` correspond to the types talk$^{(1)}$ and talk$^{(2)}$ in the EDTD from Example 8.7.7. A contributed session can contain more than one talk (`maxOccurs = "unbounded"`) but invited sessions can have only one talk.

What remains is the definition of the two types of talks:

```
<xsd:complexType name="invTalk">
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"
                 minOccurs="1" maxOccurs="1"/>
    <xsd:element name="speaker" type="xsd:string"
                 minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="conTalk">
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"
                 minOccurs="1" maxOccurs="1"/>
    <xsd:element name="authors" type="xsd:string"
                 minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

Without any further restrictions this typing mechanism would give the full power of hedge automata (see Proposition 8.7.8). But the XML schema specification imposes a constraint called "Element Declarations Consistent", which basically requires that elements with the same name that appear in the same content model have to have the same type.

Assume, for example, that we do not want to distinguish the invited and contributed sessions by two different names. But we keep the restriction that a session consists of a single invited talk or a sequence of contributed talks, as expressed with the following code:

```
<xsd:complexType name="track">
  <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:choice>
      <xsd:element name="session" type="invSession"
                   minOccurs="1" maxOccurs="1"/>
      <xsd:element name="session" type="conSession"
                   minOccurs="1" maxOccurs="1"/>
    </xsd:choice>
    <xsd:element name="break" type="xsd:string"
                 minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

This code is not valid in XML schema because it does not satisfy the "Element Declarations Consistent" constraint: In the choice construct two elements with the same name `session` occur but they have different types.

Furthermore, in XML schema the content models also have to be deterministic (called "Unique Particle Attribution") as for DTDs, which is not the case in the above code fragment: There is a choice between two elements with name `session`, corresponding to a regular expression of the type $a + a$, which is not deterministic.

In the following we focus on the "Element Declarations Consistent" constraint and do not consider the restriction to deterministic content models because this subject has already been discussed in connection with DTDs.

To capture the restriction on different types in a content model we introduce single-type EDTDs. An EDTD is called ***single-type*** if there is no regular expression on the right-hand side of a rule using two different types $a^{(i)}$ and $a^{(j)}$ of the same letter.

We have already seen that uniform membership and emptiness are polynomial for unrestricted EDTDs. If we consider single-type EDTDs that are deterministic, then even the inclusion problem becomes tractable because such EDTDs can be complemented in polynomial time.

Theorem 8.7.10. *Let D be a deterministic single-type EDTD. One can construct in polynomial time an NFHA accepting $T(\Sigma) \setminus L(D)$.*

Proof. Let $D = (\Sigma', s^{(1)}, \delta)$ be a deterministic single-type EDTD.

For those trees that do not have the correct symbol s at the root we can easily construct an automaton. So we only consider those trees with s at the root.

We only sketch the idea of the proof. The technical details are left as exercise.

The single-type property of D allows a unique top-down assignment of types to the nodes of a tree $t \in T(\Sigma)$. One starts at the root with the type of the start symbol. For all other nodes the type is determined by the type of their parents: Let $p \in \mathcal{P}os(t)$ such that the parent of p is of type $a^{(i)}$. There is a rule $a^{(i)} \to e_{a,i}$ with a deterministic regular expression $e_{a,i}$ that uses at most one type per letter in D. Let $b = t(p)$. If some type of b appears in $e_{a,i}$, then this defines the type of p. Otherwise, we assign type $b^{(k)}$ to p, where k is bigger than all the types for b used in D.

Note that this assignment of types also works if t is not in $L(D)$.

One can now construct an NFHA that assigns types to the nodes and accepts only if the assignment is correct according to the above definition, and if there is a node of type $a^{(i)}$ such that the sequence of types at its successors is not in the language of $e_{a,i}$. To check this latter condition, one has to construct automata for the complement of the languages of the $e_{a,i}$. As the expressions are deterministic, this is possible in polynomial time. □

As discussed in Section 8.5 the complementation step is the most difficult one in the inclusion algorithm.

Theorem 8.7.11. *The inclusion problem for deterministic single-type EDTDs is solvable in polynomial time.*

Given an EDTD it is very easy to check if it is single-type. In case it is not, one can decide if it can be transformed into a single-type EDTD for the same language. See the bibliographic notes for references.

Theorem 8.7.12. *Deciding whether an EDTD is equivalent to a single-type EDTD is EXPTIME-complete.*

8.7.3 Relax NG

Relax NG is the most expressive of the tree formalisms presented here. It is very close to regular tree grammars (as presented in Chapter 2) adapted to the setting of unranked trees by allowing regular expressions in the rules for capturing the horizontal languages.

Consider again the conference example with the last modification discussed in Section 8.7.2: we have invited and contributed sessions but we do not distinguish them with different labels. The following regular tree grammar defines the corresponding language of trees, where nonterminals are written in small capitals like the start symbol CONFERENCE below. We do not give the rules for those nonterminals that directly derive to data:

$$\begin{array}{rcl}
\textsc{Conference} & \rightarrow & \text{conference}(\textsc{Track}^{+} + \textsc{Sessions}) \\
\textsc{Track} & \rightarrow & \text{track}(\textsc{Sessions}) \\
\textsc{Sessions} & \rightarrow & ((\textsc{InvSession}+\textsc{ConSession})\ (\textsc{Break}\ +\varepsilon))^{+} \\
\textsc{InvSession} & \rightarrow & \text{session}(\textsc{Chair}\ \textsc{InvTalk}) \\
\textsc{ConSession} & \rightarrow & \text{session}(\textsc{Chair}\ \textsc{ConTalk}^{+}) \\
\textsc{InvTalk} & \rightarrow & \text{talk}(\textsc{Title}\ \textsc{Speaker}) \\
\textsc{ConTalk} & \rightarrow & \text{talk}(\textsc{Title}\ \textsc{Authors})
\end{array}$$

We do not precisely define the semantics of such a grammar because it is the straightforward adaption of the semantics for regular tree grammars from Chapter 2.

Such grammars can directly be expressed in Relax NG. In Figure 8.15 the above grammar is presented in Relax NG syntax. The `start`-block defines the start symbol of the grammar, and the `define`-blocks the rules of the grammar, where

- the name in the `define`-tag corresponds to the nonterminal on the left-hand side of the rule, and
- the block enclosed by `define`-tags gives the right-hand side of the rule, where
- names in element tags correspond to terminals.

In contrast to DTDs and XML schema, it is not required that the content models are deterministic. This of course increases the complexity of algorithms dealing with Relax NG schemas. Furthermore, Relax NG allows an operator `<interleave>` in the definition of the content models that corresponds to the shuffle $\|$ introduced in Section 8.5. This raises to complexity of checking whether a given document satisfies a given schema to NP (compare Theorem 8.5.6). But without the use of the interleaving operator this problem is still solvable in polynomial time (linear in the size of the document).

Checking emptiness for Relax NG schemas remains in polynomial time, even with interleaving (compare Theorem 8.5.8).

8.8 Exercises

Exercise 8.1. As for ranked trees we define for an unranked tree t the word obtained by reading the front from left to right, the yield of t, inductively by $\mathit{Yield}(a) = a$ and

```
<start>
  <ref name="Conference"/>
</start>

<define name="Conference">
  <element name="conference">
    <choice>
      <oneOrMore>
         <ref name="Track"/>
      </oneOrMore>
      <ref name="Sessions"/>
    </choice>
  </element>
</define>

<define name="Track">
  <element name="track">
    <ref name="Sessions"/>
  </element>
</define>

<define name="Sessions">
  <oneOrMore>
    <choice>
      <ref name="InvSession"/>
      <ref name="ConSession"/>
    </choice>
    <optional>
      <ref name="Break"/>
    </optional>
  </oneOrMore>
</define>
```

```
<define name="InvSession">
  <element name="session">
    <ref name="Chair"/>
    <ref name="InvTalk"/>
  </element>
</define>

<define name="ConSession">
  <element name="session">
    <ref name="Chair"/>
    <oneOrMore>
      <ref name="ConTalk"/>
    </oneOrMore>
  </element>
</define>

<define name="InvTalk">
  <element name="talk">
    <ref name="Title"/>
    <ref name="Speaker"/>
  </element>
</define>

<define name="ConTalk">
  <element name="talk">
    <ref name="Title"/>
    <ref name="Authors"/>
  </element>
</define>

<define name="Chair">
  <element name="chair">
    <text/>
  </element>
</define>
...
```

Figure 8.15: Relax NG grammar for the conference example with invited and contributed sessions

$\mathit{Yield}(a(t_1 \cdots t_n)) = \mathit{Yield}(t_1) \cdots \mathit{Yield}(t_n)$. Let $R \subseteq \Sigma^*$ be a regular word language. Construct an NFHA accepting the language $\{t \in T(\Sigma) \mid \mathit{Yield}(t) \in R\}$.

Exercise 8.2. An ϵ-NFHA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ has the same components as an NFHA except that Δ may contain ϵ-transitions of the form (q, q'). For $t \in T(\Sigma)$ we define inductively the relation $t \xrightarrow[\mathcal{A}]{*} q$ as follows. If $t = a(t_1 \cdots t_n) \in \Sigma$, then $t \xrightarrow[\mathcal{A}]{*} q$ if

- there is a rule $a(R) \to q$ in Δ with $q_1 \cdots q_n \in R$ where $t_i \xrightarrow[\mathcal{A}]{*} q_i$ for each $i \in \{1, \ldots, n\}$ (for $n = 0$ this means that $\epsilon \in R$), or
- $t \xrightarrow[\mathcal{A}]{*} q'$ and $(q', q) \in \Delta$.

We say that an ϵ-NFHA accepts t if $t \xrightarrow[\mathcal{A}]{*} q$ for some $q \in Q_f$.

Give a construction that transforms an ϵ-NFHA into an NFHA over the same set of states that accepts the same language.

Exercise 8.3. Give direct constructions to show that the class of recognizable unranked tree languages is closed under union and intersection.

Exercise 8.4. Consider a ranked alphabet $\mathcal{F}$ and assume that f is a binary associative symbol. The following recursively defined functions $A : T(\mathcal{F}) \to T(\Sigma)$ and $A_f : T(\mathcal{F}) \to H(\Sigma)$ turn a term over $\mathcal{F}$ into an unranked tree over Σ by gathering all the arguments of nested applications of f below a single node labeled f, where Σ is an unranked alphabet containing the same symbols as $\mathcal{F}$:

$$\begin{aligned}
&A(g(t_1, \ldots, t_n)) = g(A(t_1) \cdots A(t_n)) \text{ for } g \neq f\\
&A(f(t_1, t_2)) = f(A_f(t_1)\ A_f(t_2))\\
&A_f(f(t_1, t_2)) = A_f(t_1)\ A_f(t_2)\\
&A_f(g(t_1, \ldots, t_n)) = g(A(t_1) \cdots A(t_n)) \text{ for } g \neq f
\end{aligned}$$

Show the following:

(a) There is a recognizable set $L \subseteq T(\mathcal{F})$ such that $\{A(t) \in T(\Sigma) \mid t \in L\}$ needs not to be a recognizable set of unranked trees.

(b) If $L \subseteq T(\mathcal{F})$ is recognizable and closed under the congruence defined by the equation $f(x, f(y, z)) = f(f(x, y), z)$, then $\{A(t) \in T(\Sigma) \mid t \in L\}$ is recognizable.

Exercise 8.5. Let $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ be a normalized NFHA over Σ. We call $\mathcal{A}$ top-down determinsitic, i.e. a top-down DFHA, if $|Q_f| = 1$ and for all transitions $a(R) \to q$ there is for each n at most one word of length n in R.

(a) Give a recognizable language $L \subseteq T(\Sigma)$ that cannot be recognized by a top-down DFHA.

(b) Give a language $L \subseteq T(\Sigma)$ that is recognizable by a top-down DFHA such that the two languages $\mathsf{fcns}(L)$ and $\mathsf{ext}(L)$ cannot be recognized by a top-down DFTA.

Exercise 8.6. Give the proof of Proposition 8.3.3 (page 209).

Exercise 8.7. Let $L \subseteq T(\Sigma)$ be definable in WMSO and let $\varphi(X, X_1, \ldots, X_n)$ be a defining formula. Construct a formula $\varphi'(X, X_1, \ldots, X_n, X_@)$ of WS2S that defines $\mathsf{ext}(L)$ (similar to the proof of Theorem 8.4.2 on page 213 where such a formula for $\mathsf{fcns}(L)$ is constructed).

Exercise 8.8. Show that the finiteness problem for NFHA(NFA) is solvable in polynomial time. The finiteness problem for ranked tree automata is described in Section 1.7.

Exercise 8.9. Find a non-recognizable language $L \subseteq T(\Sigma)$ such that $\equiv_L$ is of finite index and the horizontal language R_a for each label $a \in \Sigma$, defined as

$$R_a = \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} \exists t \in L,\, p \in \mathcal{P}os(t) \text{ such that } t(p) = a \text{ and} \\ w = t(p1) \cdots t(pk) \text{ for } k = \max\{i \mid pi \in \mathcal{P}os(t)\} \end{array} \right\}$$

is regular.

Exercise 8.10. Consider the congruence $\equiv_L^@$ defined on page 229. Our aim is to provide a direct definition: For an unranked alphabet Σ the set of special trees is defined to be the set of trees over $\Sigma \cup \{\circ\}$ that have exactly one node labeled with $\circ$. A special tree s is always of the form $s = C[\circ(s_1 \cdots s_n)]$, where $C \in \mathcal{C}(\Sigma)$ and $s_1, \ldots, s_n \in T(\Sigma)$. For $t \in T(\Sigma)$ we define $s{\cdot}t = C[t@s_1@\cdots@s_n]$, i.e. if $t = a(t_1 \cdots t_m)$, then $s \cdot t = C[a(t_1 \cdots t_m s_1 \cdots s_n)]$. Show that $t_1 \equiv_L^@ t_2$ iff $s \cdot t_1 \in L \Leftrightarrow s \cdot t_2 \in L$ for all special trees s.

Exercise 8.11. Give the proof that recognizable languages of unranked trees with the property that every tree in the language has the same root symbol are exactly those languages definable by EDTDs (compare Proposition 8.7.8 on page 235).

Exercise 8.12. Work out the details for the construction in Theorem 8.7.10 on page 238.

8.9 Bibliographic Notes

The first papers considering automata for unranked trees and hedges appeared around the same time as those for ranked trees. In [Tha67] several basic results are stated and the connection to derivation trees of context-free grammars with regular expressions in the rules is mentioned. The proofs are not given in the paper but the reader is pointed to technical reports. Approximately at the same time, in [PQ68] languages of hedges ("bilangages") are studied from an algebraic point of view. A few years later [Tak75] considers tree languages over alphabets where symbols can have several ranks (even infinitely many). The case of unranked trees is called homogeneous rank. Besides derivation trees of context-free grammars with regular expressions also systems involving horizontal recursion are considered for generating unranked trees.

Later, the theory of tree automata has focused on ranked trees. Unranked trees became interesting again as a formal model for XML documents and the corresponding automata theory was revived by Brüggemman-Klein, Murata, and Wood [Mur00, BKMW01]. Before that in [Bar91] languages of unranked trees have been studied but using automata without regular languages on the horizontal level, i.e. with only finitely many transitions.

The term hedge was coined by Courcelle in [Cou78, Cou89] who studied algebraic notions of recognizability. This notion is not standard, in the older papers one finds the notions, ramification [PQ68], and forest [Tak75]. Nowadays, the notion hedge is used very often but the term forest also appears.

Encodings of unranked trees by ranked ones have already been considered independently of automata theory: The first-child-next-sibling encoding can be found in the textbook [Knu97]. The extension encoding was presented in [CNT04] but the extension operator for unranked trees was already used in [Tak75]. There are several other encodings that have been used in the literature. In [MSV03] transformations of unranked trees are studied using an encoding similar to the FCNS encoding. In [Cou89] an encoding based on operators to build unranked trees is presented, which is similar in spirit to the extension encoding.

We have only briefly presented the connection between logic and hedge automata. For an extensive survey on various kinds of logics for unranked trees we refer the reader to the survey [Lib06].

Many of the results concerning complexities of decision problems can be considered folklore as they are easily obtained using encodings and corresponding results for ranked tree automata. Representations using alternating automata (even two-way) for the horizontal languages are used in [MN03] to obtain complexity bounds for transformations of XML documents. For complexities of decision problems for alternating word automata we refer to [Var96] and references therein. The membership problem for regular expressions with shuffle are treated in [MS94].

The minimization problem for hedge automata is studied in [MN05] and [CLT05]. The latter article considers the model introduced at the beginning of Section 8.6.3 and gives a minimization procedure directly working on the unranked tree automaton. The results presented in Section 8.6.3 are from [MN05]. There one can also find more results on the problems discussed in Section 8.6.2. The characterization of minimal stepwise automata by congruences is also discussed in [MN05] but Theorem 8.6.9 already goes back to [Tak75].

There is a lot of literature on XML and the connection to automata and we cannot give a comprehensive overview of this very rich field of research. Our presentation is oriented along the lines of [MLMK05], where the correspondence between DTDs and local languages, XML Schema and single-type EDTDs, and Relax NG and regular tree grammars is illustrated. The results on deterministic regular expressions presented in Section 8.7.1 are from [BKW98]. Even stronger forms of unambiguity of regular expressions have been studied in [KS03] in the context of transforming streaming XML documents. The inclusion problem for DTDs with other fragments of regular expressions is considered in detail in [MNS04].

The results on single-type EDTDs (Theorems 8.7.11 and 8.7.12) are from [MNS05] (where EDTDs are called specialized DTDs). Single-type EDTDs are close to top-down deterministic automata as defined in [CLT05], where it is shown that it is decidable whether a given NFHA is equivalent to one that is top-down deterministic.

Something that has not been discussed in this chapter are problems for streaming XML documents. In this setting the documents are not accessible in their tree shape but arrive as a stream and can only be parsed once. Problems in this framework have for example been addressed in [SV02] and [MNS05, MNSB06].

Another aspect that we did not take into account concerns the data values in the XML documents. In this case one has to deal with automata over infinite alphabets. This is addressed, e.g., in [NSV04] and [BMS$^+$06, BDM$^+$06].

Finally, one should certainly mention the survey [Nev02], which gives a nice overview of several topics related to XML and automata for unranked trees.

Bibliography

[AD82] A. Arnold and M. Dauchet. Morphismes et bimorphismes d'arbres. *Theorical Computer Science*, 20:33–93, 1982.

[AG68] M. A. Arbib and Y. Give'on. Algebra automata I: Parallel programming as a prolegomena to the categorical approach. *Information and Control*, 12(4):331–345, April 1968.

[AKVW93] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proceedings of Computer Science Logic*, volume 832 of *Lecture Notes in Computer Science*, pages 1–17, 1993. Techn. Report 93-1352, Cornell University.

[AKW95] A. Aiken, D. Kozen, and E.L. Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44, October 1995.

[AM78] M.A. Arbib and E.G. Manes. Tree transformations and semantics of loop-free programs. *Acta Cybernetica*, 4:11–17, 1978.

[AM91] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *Proceedings of the ACM conf. on Functional Programming Languages and Computer Architecture*, pages 427–447, 1991.

[AU71] A. V. Aho and J. D. Ullmann. Translations on a context-free grammar. *Information and Control*, 19:439–475, 1971.

[AW92] A. Aiken and E.L. Wimmers. Solving Systems of Set Constraints. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science* [IEE92], pages 329–340.

[Bak78] B.S. Baker. Generalized syntax directed translation, tree transducers, and linear space. *Journal of Comput. and Syst. Sci.*, 7:876–891, 1978.

[Bar91] A. Barrero. Unranked tree languages. *Pattern Recognition*, 24(1):9–18, 1991.

[BDM$^+$06] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and xml reasoning. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2006*, pages 10–19. ACM, 2006.

[BGG97] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem.* Perspectives of Mathematical Logic. Springer Verlag, 1997.

[BGW93] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 75–83. IEEE Computer Society Press, 19–23 June 1993.

[BJ97] A. Bouhoula and J.-P. Jouannaud. Automata-driven automated induction. In *Proceedings, 12^{th} Annual IEEE Symposium on Logic in Computer Science* [IEE97].

[BKMW01] A. Brüggemann-Klein, M.Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKTUST-TCSC-2001-05, HKUST Theoretical Computer Science Center Research, 2001.

[BKW98] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[BMS$^+$06] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 7–16. IEEE Computer Society, 2006.

[Boz99] S. Bozapalidis. Equational elements in additive algebras. *Theory of Computing Systems*, 32(1):1–33, 1999.

[Boz01] S. Bozapalidis. Context-free series on trees. *ICOMP*, 169(2):186–229, 2001.

[BR82] Jean Berstel and Christophe Reutenauer. Recognizable formal power series on trees. *TCS*, 18:115–148, 1982.

[Bra68] W. S. Brainerd. The minimalization of tree automata. *Information and Control*, 13(5):484–491, November 1968.

[Bra69] W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14(2):217–231, February 1969.

[BT92] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In A. Finkel and M. Jantzen, editors, *9^th Annual Symposium on Theoretical Aspects of Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 161–171, 1992.

[Büc60] J. R. Büchi. On a decision method in a restricted second order arithmetic. In Stanford Univ. Press., editor, *Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science*, pages 1–11, 1960.

[CCC+94] A.-C. Caron, H. Comon, J.-L. Coquidé, M. Dauchet, and F. Jacquemard. Pumping, cleaning and symbolic constraints solving. In *Proceedings, International Colloquium Automata Languages and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 436–449, 1994.

[CD94] H. Comon and C. Delor. Equational formulae with membership constraints. *Information and Computation*, 112(2):167–216, August 1994.

[CDGV94] J.-L. Coquide, M. Dauchet, R. Gilleron, and S. Vagvolgyi. Bottom-up tree pushdown automata : Classification and connection with rewrite systems. *Theorical Computer Science*, 127:69–98, 1994.

[CG90] J.-L. Coquidé and R. Gilleron. Proofs and reachability problem for ground rewrite systems. In *Proc. IMYCS'90*, Smolenice Castle, Czechoslovakia, November 1990.

[CGL+03] J. Carme, R. Gilleron, A. Lemay, A. Terlutte, and M. Tommasi. Residual finite tree automata. In *7th International Conference on Developments in Language Theory*, number 2710 in Lecture Notes in Computer Science, pages 171 – 182. Springer Verlag, July 2003.

[Chu62] A. Church. Logic, arithmetic, automata. In *Proc. International Mathematical Congress*, 1962.

[CJ97a] H. Comon and F. Jacquemard. Ground reducibility is EXPTIME-complete. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science* [IEE97], pages 26–34.

[CJ97b] H. Comon and Y. Jurski. Higher-order matching and tree automata. In M. Nielsen and W. Thomas, editors, *Proc. Conf. on Computer Science Logic*, volume 1414 of *LNCS*, pages 157–176, Aarhus, August 1997. Springer-Verlag.

[CK96] A. Cheng and D. Kozen. A complete Gentzen-style axiomatization for set constraints. In *Proceedings, International Colloquium Automata Languages and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 134–145, 1996.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[CLT05] J. Cristau, C. Löding, and W. Thomas. Deterministic automata on unranked trees. In *Proceedings of the 15th International Symposium on Fundamentals of Computation Theory, FCT 2005*, volume 3623 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2005.

[CNT04] J. Carme, J. Niehren, and M. Tommasi. Querying unranked trees with stepwise tree automata. In *International Conference on Rewriting Techniques and Applications, Aachen*, volume 3091 of *Lecture Notes in Computer Science*, pages 105–118. Springer, 2004.

[Com89] H. Comon. Inductive proofs by specification transformations. In *Proceedings, Third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 76–91, 1989.

[Com95] H. Comon. Sequentiality, second-order monadic logic and tree automata. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 26–29 June 1995.

[Com98a] H. Comon. Completion of rewrite systems with membership constraints. Part I: deduction rules. *Journal of Symbolic Computation*, 25:397–419, 1998. This is a first part of a paper whose abstract appeared in Proc. ICALP 92, Vienna.

[Com98b] H. Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *Journal of Symbolic Computation*, 25:421–453, 1998. This is the second part of a paper whose abstract appeared in Proc. ICALP 92, Vienna.

[Cou78] B. Courcelle. A representation of trees by languages I. *Theorical Computer Science*, 6, 1978.

[Cou86] B. Courcelle. Equivalences and transformations of regular systems–applications to recursive program schemes and grammars. *Theorical Computer Science*, 42, 1986.

[Cou89] B. Courcelle. *On Recognizable Sets and Tree Automata*, chapter Resolution of Equations in Algebraic Structures. Academic Press, m. Nivat and Ait-Kaci edition, 1989.

[CP94a] W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 128–136. IEEE Computer Society Press, 4–7 July 1994.

[CP94b] W. Charatonik and L. Pacholski. Set constraints with projections are in NEXPTIME. In *Proceedings of the* 35^{th} *Symp. Foundations of Computer Science*, pages 642–653, 1994.

[CP97] W. Charatonik and A. Podelski. Set Constraints with Intersection. In *Proceedings,* 12^{th} *Annual IEEE Symposium on Logic in Computer Science* [IEE97].

[Dau94] M. Dauchet. Rewriting and tree automata. In H. Comon and J.-P. Jouannaud, editors, *Proc. Spring School on Theoretical Computer Science: Rewriting*, Lecture Notes in Computer Science, Odeillo, France, 1994. Springer Verlag.

[DCC95] M. Dauchet, A.-C. Caron, and J.-L. Coquidé. Reduction properties and automata with constraints. *Journal of Symbolic Computation*, 20:215–233, 1995.

[DGN$^+$98] A. Degtyarev, Y. Gurevich, P. Narendran, M. Veanes, and A. Voronkov. The decidability of simultaneous rigid e-unification with one variable. In T. Nipkow, editor, *$9^t h$ International Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, 1998.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter Rewrite Systems, pages 243–320. Elsevier, 1990.

[DM97] I. Durand and A. Middeldorp. Decidable call by need computations in term rewriting. In W. McCune, editor, *Proc. 14th Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 4–18. Springer Verlag, 1997.

[Don65] J. E. Doner. Decidability of the weak second-order theory of two successors. *Notices Amer. Math. Soc.*, 12:365–468, March 1965.

[Don70] J. E. Doner. Tree acceptors and some of their applications. *Journal of Comput. and Syst. Sci.*, 4:406–451, 1970.

[DT90] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 242–248. IEEE Computer Society Press, 4–7 June 1990.

[DT92] M. Dauchet and S. Tison. Structural complexity of classes of tree languages. In M. Nivat and A. Podelski, editors, *Tree Automata and Languages*, pages 327–353. Elsevier Science, 1992.

[DTHL87] M. Dauchet, S. Tison, T. Heuillard, and P. Lescanne. Decidability of the confluence of ground term rewriting systems. In *Proceedings, Symposium on Logic in Computer Science*, pages 353–359. The Computer Society of the IEEE, 22–25 June 1987.

[DTT97] P. Devienne, J.-M. Talbot, and S. Tison. Solving classes of set constraints with tree automata. In G. Smolka, editor, *Proceedings of the 3^{th} International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 62–76, oct 1997.

[Eng75] J. Engelfriet. Bottom-up and top-down tree transformations. a comparision. *Mathematical System Theory*, 9:198–231, 1975.

[Eng77] J. Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical System Theory*, 10:198–231, 1977.

[Eng78] J. Engelfriet. A hierarchy of tree transducers. In *Proceedings of the third Les Arbres en Algèbre et en Programmation*, pages 103–106, Lille, 1978.

[Eng82] J. Engelfriet. Three hierarchies of transducers. *Mathematical System Theory*, 15:95–125, 1982.

[ES78] J. Engelfriet and E.M. Schmidt. IO and OI II. *Journal of Comput. and Syst. Sci.*, 16:67–99, 1978.

[Esi83] Z. Esik. Decidability results concerning tree transducers. *Acta Cybernetica*, 5:303–314, 1983.

[EV91] J. Engelfriet and H. Vogler. Modular tree transducers. *Theorical Computer Science*, 78:267–303, 1991.

[EW67] S. Eilenberg and J. B. Wright. Automata in general algebras. *Information and Control*, 11(4):452–470, 1967.

[FSVY91] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 300–309, 1991.

[FV88] Z. Fülöp and S. Vágvölgyi. A characterization of irreducible sets modulo left-linear term rewiting systems by tree automata. Un type rr ??, Research Group on Theory of Automata, Hungarian Academy of Sciences, H-6720 Szeged, Somogyi u. 7. Hungary, 1988.

[FV89] Z. Fülöp and S. Vágvölgyi. Congruential tree languages are the same as recognizable tree languages–A proof for a theorem of D. kozen. *Bulletin of the European Association of Theoretical Computer Science*, 39, 1989.

[FV98] Z. Fülöp and H. Vögler. *Formal Models Based on Tree Transducers.* Monographs in Theoretical Computer Science. Springer Verlag, 1998.

[GB85] J. H. Gallier and R. V. Book. Reductions in tree replacement systems. *Theorical Computer Science*, 37(2):123–150, 1985.

[Gen97] T. Genet. Decidable approximations of sets of descendants and sets of normal forms - extended version. Technical Report RR-3325, Inria, Institut National de Recherche en Informatique et en Automatique, 1997.

[GJV98] H. Ganzinger, F. Jacquemard, and M. Veanes. Rigid reachability. In *Proc. ASIAN'98*, volume 1538 of *Lecture Notes in Computer Science*, pages 4–??, Berlin, 1998. Springer-Verlag.

[GMW97] H. Ganzinger, C. Meyer, and C. Weidenbach. Soft typing for ordered resolution. In W. McCune, editor, *Proc. 14th Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 1997.

[Gou00] Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *Proc. 15 IPDPS 2000 Workshops, Cancun, Mexico, May 2000*, volume 1800 of *Lecture Notes in Computer Science*, pages 977–984. Springer Verlag, 2000.

[GRS87] J. Gallier, S. Raatz, and W. Snyder. Theorem proving using rigid E-unification: Equational matings. In *Proc. 2nd IEEE Symp. Logic in Computer Science, Ithaca, NY*, June 1987.

[GS84] F. Gécseg and M. Steinby. *Tree Automata.* Akademiai Kiado, 1984.

[GS96] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer Verlag, 1996.

[GT95] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24:157–176, 1995.

[GTT93] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In *Proceedings of the 34th Symp. on Foundations of Computer Science*, pages 372–380, 1993. Full version in the LIFL Tech. Rep. IT-247.

[GTT99] R. Gilleron, S. Tison, and M. Tommasi. Set constraints and automata. *Information and Control*, 149:1 – 41, 1999.

[Gue83] I. Guessarian. Pushdowm tree automata. *Mathematical System Theory*, 16:237–264, 1983.

[Hei92] N. Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[HJ90a] N. Heintze and J. Jaffar. A Decision Procedure for a Class of Set Constraints. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51. IEEE Computer Society Press, 4–7 June 1990.

[HJ90b] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th ACM Symp. on Principles of Programming Languages*, pages 197–209, 1990. Full version in the IBM tech. rep. RC 16089 (#71415).

[HJ92] N. Heintze and J. Jaffar. An engine for logic program analysis. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science* [IEE92], pages 318–328.

[HL91] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems I. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–414. MIT Press, 1991. This paper was written in 1979.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[IEE92] IEEE Computer Society Press. *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, 22–25 June 1992.

[IEE97] IEEE Computer Society Press. *Proceedings, 12*$^{\text{th}}$ *Annual IEEE Symposium on Logic in Computer Science*, 1997.

[Jac96] F. Jacquemard. Decidable approximations of term rewriting systems. In H. Ganzinger, editor, *Proceedings. Seventh International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, 1996.

[JM79] N. D. Jones and S. S. Muchnick. Flow Analysis and Optimization of LISP-like Structures. In *Proceedings of the 6^{th} ACM Symposium on Principles of Programming Languages*, pages 244–246, 1979.

[Jon87] N. Jones. *Abstract interpretation of declarative languages*, chapter Flow analysis of lazy higher-order functional programs, pages 103–122. Ellis Horwood Ltd, 1987.

[Jr.76] William H. Joyner Jr. Resolution strategies as decision procedures. *Journal of the ACM*, 23(3):398–417, 1976.

[KFK97] Y. Kaji, T. Fujiwara, and T. Kasami. Solving a unification problem under constrained substitutions using tree automata. *Journal of Symbolic Computation*, 23(1):79–118, January 1997.

[Knu97] D.E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms.* Addison-Wesley, 3 edition, 1997.

[Koz92] D. Kozen. On the Myhill-Nerode theorem for trees. *Bulletin of the European Association of Theoretical Computer Science*, 47:170–173, June 1992.

[Koz93] D. Kozen. Logical aspects of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proceedings of Computer Science Logic*, volume 832 of *Lecture Notes in Computer Science*, pages 175–188, 1993.

[Koz95] D. Kozen. Rational spaces and set constraints. In *Proceedings of the 6^{th} International Joint Conference on Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 42–61, 1995.

[Koz98] D. Kozen. Set constraints and logic programming. *Information and Computation*, 142(1):2–25, 1998.

[KS03] C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. In *Database Programming Languages, 9th International Workshop, DBPL 2003*, volume 2921 of *Lecture Notes in Computer Science*, pages 233–256. Springer, 2003.

[Kuc91] G. A. Kucherov. On relationship between term rewriting systems and regular tree languages. In R. Book, editor, *Proceedings. Fourth International Conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 299–311, April 1991.

[Kui99] W. Kuich. Full abstract families of tree series i. In Juhani Karhumäki, Hermann A. Maurer, and Gheorghe Paun andy Grzegorz Rozenberg, editors, *Jewels are Forever*, pages 145–156. SV, 1999.

[Kui01] W. Kuich. Pushdown tree automata, algebraic tree systems, and algebraic tree series. *Information and Computation*, 165(1):69–99, 2001.

[KVW00] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching time model-checking. *Journal of the ACM*, 47(2):312–360, 2000.

[Lib06] Leonid Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

[LM87] J.-L. Lassez and K. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):301–318, September 1987.

[LM93] D. Lugiez and J.-L. Moysset. Complement problems and tree automata in AC-like theories. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *10^{t}h Annual Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 515–524, Würzburg, 25–27 February 1993.

[LM94] Denis Lugiez and Jean-Luc Moysset. Tree automata help one to solve equational formulae in ac-theories. *Journal of Symbolic Computation*, 18(4):297–318, 1994.

[Loh01] M. Lohrey. On the parallel complexity of tree automata. In *Proceedings of the 12th Conference on Rewriting and Applications*, pages 201–216, 2001.

[MGKW96] D. McAllester, R. Givan, D. Kozen, and C. Witty. Tarskian set constraints. In *Proceedings, 11*$^{\text{th}}$ *Annual IEEE Symposium on Logic in Computer Science*, pages 138–141. IEEE Computer Society Press, 27–30 July 1996.

[Mis84] P. Mishra. Towards a Theory of Types in PROLOG. In *Proceedings of the 1*st *IEEE Symposium on Logic Programming*, pages 456–461, Atlantic City, 1984.

[MLMK05] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.

[MN03] W. Martens and F. Neven. Typechecking top-down uniform unranked tree transducers. In *Database Theory - ICDT 2003, 9th International Conference, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2003.

[MN05] W. Martens and J. Niehren. Minimizing tree automata for unranked trees. In *Database Programming Languages, 10th International Symposium, DBPL 2005*, volume 3774 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 2005.

[MNS04] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Mathematical Foundations of Computer Science 2004, 29th International Symposium, MFCS 2004, Proceedings*, volume 3153 of *Lecture Notes in Computer Science*, pages 889–900. Springer, 2004.

[MNS05] W. Martens, F. Neven, and T. Schwentick. Which XML schemas admit 1-pass preorder typing? In *Database Theory - ICDT 2005, 10th International Conference, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2005.

[MNSB06] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML schema. *ACM Transactions on Database Systems*, 31(3):770–813, 2006.

[Mon81] J. Mongy. *Transformation de noyaux reconnaissables d'arbres. Forêts RATEG.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Villeneuve d'Ascq, France, 1981.

[MS94] A. J. Mayer and L. J. Stockmeyer. Word problems – this time with interleaving. *Information and Computation*, 115(2):293–311, 1994.

[MS96] A. Mateescu and A. Salomaa. Aspects of classical language theory. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 175–246. Springer Verlag, 1996.

[MSV03] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *Journal of Comput. and Syst. Sci.*, 66(1):66–97, 2003.

[Mur00] M. Murata. "Hedge Automata: a Formal Model for XML Schemata". Web page, 2000.

[MW67] J. Mezei and J. B. Wright. Algebraic automata and context-free sets. *Information and Control*, 11:3–29, 1967.

[Nev02] F. Neven. Automata, logic, and XML. In *Computer Science Logic, 16th International Workshop, CSL 2002, Proceedings*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2002.

[Niv68] M. Nivat. *Transductions des langages de Chomsky.* Thèse d'etat, Paris, 1968.

[NP89] M. Nivat and A. Podelski. *Resolution of Equations in Algebraic Structures*, volume 1, chapter Tree monoids and recognizable sets of finite trees, pages 351–367. Academic Press, New York, 1989.

[NP97] M. Nivat and A. Podelski. Minimal ascending and descending tree automata. *SIAM Journal on Computing*, 26(1):39–58, February 1997.

[NSV04] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5(3):403–435, 2004.

[NT99] T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. In M. Rusinowitch F. Narendran, editor, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, pages 256–270, Trento, Italy, 1999. Springer Verlag.

[Oya93] M. Oyamaguchi. NV-sequentiality: a decidable condition for call-by-need computations in term rewriting systems. *SIAM Journal on Computing*, 22(1):114–135, 1993.

[Pel97] N. Peltier. Tree automata and automated model building. *Fundamenta Informaticae*, 30(1):59–81, 1997.

[Pla85] D. A. Plaisted. Semantic confluence tests and completion method. *Information and Control*, 65:182–215, 1985.

[Pod92] A. Podelski. A monoid approach to tree automata. In Nivat and Podelski, editors, *Tree Automata and Languages, Studies in Computer Science and Artificial Intelligence 10*. North-Holland, 1992.

[PQ68] C. Pair and A. Quere. Définition et étude des bilangages réguliers. *Information and Control*, 13(6):565–593, 1968.

[Rab69] M. O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Rab77] M. O. Rabin. *Handbook of Mathematical Logic*, chapter Decidable theories, pages 595–627. North Holland, 1977.

[Rao92] J.-C. Raoult. A survey of tree transductions. In M. Nivat and A. Podelski, editors, *Tree Automata and Languages*, pages 311–325. Elsevier Science, 1992.

[Rey69] J. C. Reynolds. Automatic Computation of Data Set Definition. *Information Processing*, 68:456–461, 1969.

[Sal73] A. Salomaa. *Formal Languages*. Academic Press, New York, 1973.

[Sal88] K. Salomaa. Deterministic tree pushdown automata and monadic tree rewriting systems. *Journal of Comput. and Syst. Sci.*, 37:367–394, 1988.

[Sal94] K. Salomaa. Synchronized tree automata. *Theorical Computer Science*, 127:25–51, 1994.

[Sei89] H. Seidl. Deciding equivalence of finite tree automata. In *Annual Symposium on Theoretical Aspects of Computer Science*, 1989.

[Sei90] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19, 1990.

[Sei92] H. Seidl. Single-valuedness of tree transducers is decidable in polynomial time. *Theorical Computer Science*, 106:135–181, 1992.

[Sei94a] H. Seidl. Equivalence of finite-valued tree transducers is decidable. *Mathematical System Theory*, 27:285–346, 1994.

[Sei94b] H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

[Sén97] G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 671–681, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[Sey94] F. Seynhaeve. Contraintes ensemblistes. Master's thesis, LIFL, 1994.

[Slu85] G. Slutzki. Alternating tree automata. *Theorical Computer Science*, 41:305–318, 1985.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proc. 5th ACM Symp. on Theory of Computing*, pages 1–9, 1973.

[Ste94] K. Stefansson. Systems of set constraints with negative constraints are nexptime-complete. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 137–141. IEEE Computer Society Press, 4–7 July 1994.

[SV95] G. Slutzki and S. Vagvolgyi. Deterministic top-down tree transducers with iterated look-ahead. *Theorical Computer Science*, 143:285–308, 1995.

[SV02] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proceedings of the 21st Symposium on Principles of Database Systems, PODS'02*, pages 53–64. ACM, 2002.

[Tak75] M. Takahashi. Generalizations of regular sets and their application to a study of context-free languages. *Information and Control*, 27:1–36, 1975.

[Tha67] J. W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Comput. and Syst. Sci.*, 1:317–322, 1967.

[Tha70] J. W. Thatcher. Generalized sequential machines. *Journal of Comput. and Syst. Sci.*, 4:339–367, 1970.

[Tha73] J. W. Thatcher. Tree automata: an informal survey. In A.V. Aho, editor, *Currents in the theory of computing*, pages 143–178. Prentice Hall, 1973.

[Tho90] W. Thomas. *Handbook of Theoretical Computer Science*, volume B, chapter Automata on Infinite Objects, pages 134–191. Elsevier, 1990.

[Tho97] W. Thomas. Languages, automata and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–456. Springer Verlag, 1997.

[Tis89] S. Tison. Fair termination is decidable for ground systems. In *Proceedings, Third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 462–476, 1989.

[Tiu92] J. Tiuryn. Subtype inequalities. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science* [IEE92], pages 308–317.

[Tom92] M. Tommasi. Automates d'arbres avec tests d'égalité entre cousins germains. Mémoire de DEA, Univ. Lille I, 1992.

[Tom94] M. Tommasi. *Automates et contraintes ensemblistes.* PhD thesis, LIFL, 1994.

[Tra95] B. Trakhtenbrot. Origins and metamorphoses of the trinity: Logic, nets, automata. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 26–29 June 1995.

[Tre96] R. Treinen. The first-order theory of one-step rewriting is undecidable. In H. Ganzinger, editor, *Proceedings. Seventh International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 276–286, 1996.

[TW65] J. W. Thatcher and J. B. Wright. Generalized finite automata. *Notices Amer. Math. Soc.*, 820, 1965. Abstract No 65T-649.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.

[Uri92] T. E. Uribe. Sorted Unification Using Set Constraints. In D. Kapur, editor, *Proceedings of the 11^{th} International Conference on Automated Deduction*, New York, 1992.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266, 1996.

[Vea97a] M. Veanes. On computational complexity of basic decision problems of finite tree automata. Technical report, Uppsala Computing Science Department, 1997.

[Vea97b] M. Veanes. *On simultaneous rigid E-unification.* PhD thesis, Computing Science Department, Uppsala University, Uppsala, Sweden, 1997.

[Zac79] Z. Zachar. The solvability of the equivalence problem for deterministic frontier-to-root tree transducers. *Acta Cybernetica*, 4:167–177, 1979.

Index

α-equivalence, 104
β-reduction, 104
ϵ-free, 34
ϵ-rules, 23
  for NFHA, 205
η-long form, 104
$\models$, 88, 114
Rec, 76
$\mathit{Rec}_\times$, 75

acceptance
  by an automaton, 115
accepted
  term, 21
  unranked tree, 202
accepts, 115
accessible, 24
alphabetic, 35
alternating
  tree automaton, 183
  word automaton, 183
alternating automaton
  weak, 196
arity, 15
automaton
  $\Box$-automaton, 104
  generalized reduction automaton, 133
  pushdown, 196
  reduction automaton, 126
  stepwise, 226
  with constraints between brothers, 122
  with equality and disequality constraints, 114
AWCBB, 122
AWEDC, 114
axiom, 51

bimorphism
  word, 166
close equalities, 128
closed, 16
closure, 56
closure properties
  for $\mathit{Rec}_\times$,Rec, 80
  for GTTs, 82
  for hedge automata, 211
closure property, 29
  complementation, 30
  intersection, 30
  union, 30
complete, 35, 117
  NFHA, 204
complete specification
  of an automaton with constraints, 117
concatenation, 56
congruence, 35
  finite index, 35
  for unranked trees, 221, 229
constraint
  disequality constraint, 114
  equality constraint, 114
content model, 232
context, 17
context-free
  tree grammar, 66
  tree languages, 67
  word grammar, 64
cryptographic protocols, 195
cylindrification, 80

definable
  language of unranked trees, 213
  set, 90
definite set constraints, 190
delabeling, 35
derivation relation, 52
derivation trees, 64
determinacy

    - of an automaton with constraints, 117
- deterministic, 117
    - hedge automaton, 205
    - regular expression, 233
    - tree automaton, 23
- determinization, 25, 117
- DFHA, 205
- DFTA, *see* tree automaton
- disequality constraint, 114
- domain, 17
- DSHA, 226
- DTD, 232
    - deterministic, 234
- DUTT, *see* tree transducer

- E-unification, 102
- EDTD, 234
    - single-type, 238
- encoding
    - FCNS, 207
- encompassment, 97
- equality constraint, 114
- equivalent, 21, 52
    - NFHA, 203
- extended DTD, 234
- extension encoding, 210
- extension operator, 210

- FCNS encoding, 207
- final states, 114
- first order logic
    - monadic fragment, 189
- first-child-next-sibling encoding, 207
- free variables, 103
- frontier position, 16
- FTA, *see* tree automaton

- generalized reduction automata, 133
- generalized tree set, 142
    - accepted, 143
    - regular, 146
- generalized tree set automaton, 142, *see* GTSA
- ground reducibility, 98, 125, 133
- ground rewriting
    - theory of, 100
- ground substitution, 17
- ground terms, 15
- Ground Tree Transducer, 76
- GTS, *see* generalized tree set
- GTSA
    - complete, 143
    - deterministic, 143
    - run, 142
    - simple, 143
    - strongly deterministic, 143
    - sucessful run, 142
- GTT, 76

- hedge, 201
- hedge automaton, 202
- height, 16
    - of an unranked tree, 201
- horizontal language, 202
- horizontal languages, 202

- index, 101
- interleaving, 215
- IO, 67

- Löwenheim class, 190, 197
- language
    - accepted by an automaton with constraints, 115
    - of a hedge automaton, 202
    - recognizable, 21
    - recognized, 21
- language accepted, 115
- language generated, 52
- linear, 15, 32
- local, 69, 233

- matching problem, 104
- monadic class, 190
- monotonic
    - predicate, 100
- move relation
    - for NFTA, 20
    - for rational transducers, 164
- Myhill-Nerode Theorem, 35

- NDTT, *see* tree transducer
- NFHA, 202
- NFHA(NFA), 216
- NFTA, *see* tree automaton
- non-terminal, 51
- normalized, 53
- normalized NFHA, 205
- NUTT, *see* tree transducer

OI, 67
order, 103
order-sorted signatures, 96
overlapping constraints, 134

path-closed, 39
pop clause, 191
position, 16
Presburger's arithmetic, 73
production rules, 52
productive, 52
program analysis, 138
projection, 80, 212
pumping lemma, 29
push clause, 191
pushdown automaton, 195

Rabin
  automaton, 74, 96
  theorem, 96
ranked alphabet, 15
RATEG, 113
reachable, 52
recognition
  by an automaton, 115
recognizable
  by a hedge automaton, 204
  language of unranked trees, 211
recognized, 115
recognizes, 115
reduced, 52
  NFHA, 204
reducibility theory, 98, 125, 133
reduction automaton, 126
regular equation systems, 61
regular expression
  deterministic, 233
regular tree expressions, 57
regular tree grammars, 52
regular tree language, 52
relation
  rational relation, 106
  of bounded delay, 106
remote equalities, 128
root symbol, 16
rules
  ϵ-rules, 23
run, 22, 115
  of an alternating tree automaton, 185
  of an alternating word automaton, 184
  of an automaton, 115
  of hedge automaton, 202
  successful, 22

sequential calculus, 73
sequentiality, 101
set constraints, 137
  definite, 190
shuffle, 215
size, 16, 116
  of a constraint, 116
  of an automaton with constraints, 116
SkS, 88
solution, 104
sort
  constraint, 96
  expression, 96
  symbol, 96
state
  accessible, 24
  dead, 24
stepwise automaton, 226
substitution, 17
subterm, 16
subterm ordering, 16
subtree
  of an unranked tree, 201
success node, 184
symbol to symbol, 34
synchronization states, 76

target state, 114
temporal logic
  propositional linear time temporal logic, 108
term
  in the simply typed lambda calculus, 103
  well-formed, 96
terminal, 51
terms, 15, 103
theory
  of reducibility, 98
transducer
  ε-free, 164
  rational, 164
transition rules, 114

tree, 15
  unranked, 201
tree automaton, 20
  flat tree automaton, 47
  alternating, 183
  complete, 24
  deterministic, 23
  dual, 188
  generalized, *see* GTSA
  reduced, 24
  reduction automaton, 114
  top down, 38
  two-way, 191
  weak alternating, 196
  with ϵ-rules, 22
  with constraints between brothers, 113
tree homomorphism
  alphabetic, 35
  complete, 35
  delabeling, 35
  linear, 32
  symbol to symbol, 34
tree substitution, 55
tree transducer
  ε-free, 173
  bottom-up, 173
  complete, 173
  deterministic, 173
  linear, 173
  non-erasing, 173
  top-down, 174
tree grammar, 51
tree homomorphism, 31
tree automaton
  alternating, 185
  product, 30
tree homomorphism
  ϵ-free, 34
two-way tree automaton, 191
type
  in the simply typed lambda calculus, 103
type inference, 138

unranked tree, 201

variable position, 16
variables, 15

weak monadic second-order logic, 212
Weak Second-order monadic logic with K successors, 95
WMSO, 212
WS1S, 73
WSkS, 73, 87, 95

Yield, 63