Limburgs Universitair Centrum

Faculteit Wetenschappen

Design and Analysis of Query Languages for Structured Documents

A Formal and Logical Approach

Proefschrift voorgelegd tot het behalen van de graad van
doctor in de wetenschappen, richting informatica
aan het Limburgs Universitair Centrum
te verdedigen door

Frank Neven

Promotor:
Prof. Dr. J. Van den Bussche

1999

D/1999/2451/123

Preface

During the last three years, many people have contributed directly as well as indirectly to the development of this dissertation. I would like to express my gratitude to them.

First of all, I am deeply indebted to Jan Van den Bussche, my advisor, for his continuing support and guidance during the evolution of this thesis. This dissertation would not have been possible without his considerable advice and the long discussions we had on almost every detail. Further, I am grateful to Thomas Schwentick. This thesis would have looked a lot different if he hadn't introduced me to the "microcosmos of MSO-types". Chapter 5 discusses our joint work; Proposition 5.22 and the idea of *strong* QAs are entirely due to him. This thesis further benefitted from pleasant discussions with, among others, Geert Jan Bex (XML/XSLT oracle), Véronique Bruyère, Joost Engelfriet, Sebastian Maneth (first half of the XSL Think Tank), Dan Suciu, Wolfgang Thomas, Dirk Van Gucht, Moshe Vardi, Victor Vianu, Gottfried Vossen, and Derick Wood. I thank Joost in particular for bringing Lemma 5.9 to my attention. I thank the members of our research group for creating a stimulating environment. Among them, I especially want to mention Marc Gyssens for his scientific (and touristic) guidance, and my two roommates Floris Geerts and Bart Goethals for their enjoyable (but sometimes quite noisy) company. I further thank Bart, Floris, and Geert Jan, for helping hands with TEXnical and computer related problems, and Floris and Marc for proofreading parts of this thesis. Many thanks to the WNI secretaries: Annemie, Conny, Hilde, Martine, and Viviane. I am also grateful to my employer, the Fund for Scientific Research, Flanders.

Last, but definitely not least, I thank my parents for their support, and my sister and my friends (some of which are already mentioned above) for keeping me connected to the real world.

Diepenbeek, December 17, 1999 *Frank Neven*

Abstract

This dissertation introduces query languages for structured documents and investigates their expressiveness and optimization.

It is natural to model structured documents, like for example XML documents, by labeled trees where the children of a node are ordered. We therefore revisit some of the established formal language theory formalisms for computations on trees but approach them from a query language perspective.

We start with a study of the classical formalism of attribute grammars to query documents modeled as derivation trees of context-free grammars. By restricting the attributes to Booleans and relations, and the semantic rules to propositional logic and first-order logic formulas, respectively, we obtain powerful query languages well suited for expressing unary and relational queries. Further restrictions as well as generalizations lead to a complete picture of the expressiveness of such languages. Interestingly, we show that the above formalisms can be readily implemented on top of a deductive database by exhibiting a translation into datalog with negation.

In the rest of the dissertation, we focus on formalisms expressing unary, also called selection, queries. These are important as, on the one hand, they constitute the most simple and common form of document querying, and, on the other hand, they form the basis of more general query languages transforming documents into other documents.

Specifically, we introduce extensions of attribute grammars (extended AGs) to query documents modeled by extended context-free grammars which correspond more closely to the actual XML document type definitions (DTDs). A fundamental difference is that now derivation trees are no longer ranked, which means that nodes need not have a fixed maximal number of children. This seemingly innocent difference greatly complicates the definition of attribute grammars. We give full account of the expressiveness of the query languages based on this formalism and obtain the exact complexity of various optimization questions.

Next, we abandon attribute grammars and turn to another well-studied computation model for trees: the tree automaton. In particular, we want to understand how such automata, on both ranked and unranked trees, can be used to express unary structured document queries. Concretely, we define a query automaton (QA) as a two-way deterministic finite automaton over trees that can select nodes depending on the state and the label at those nodes. We study the expressiveness of QAs and investigate the exact complexity of various optimization questions.

Finally, we apply the techniques and results obtained in this work. We drastically improve the upper bound on the complexity of the equivalence test of Region Algebra expressions from iterated exponential to EXPTIME by essentially translating the latter into equivalent extended AGs. By employing the techniques used to obtain our expressiveness results, we establish the expressiveness as a pattern language of the actual XML transformation language XSLT. Further, we obtain that our languages are more expressive than most current query languages for structured documents and semi-structured data.

As argued by Suciu [Suc98], dealing with the inherent order of children of nodes in documents is a major research issue in the design of query languages for semi-structured data and XML. An important contribution of this work is that all proposed query languages can take this ordering into account.

Contents

1

Introduction

1.1 Motivation and aim

The increasing popularity of the Internet together with the use of markup languages like HTML or XML [Con], has lead to huge repositories of electronic structured documents. Current database systems, however, are not suited to manage this new type of data. Therefore, the need for new database systems and associated query languages capable of storing and manipulating such structured documents emerges. In this work, we focus on the design and analysis of such query languages.

Perhaps the best known example of a document specification language is the eXtensible Markup Language (XML), already referred to above. This language is the new standard adopted by the World Wide Web Consortium (W3C) for the specification of structured documents. The language quickly became immensely popular. In fact, many software vendors already bet on XML for becoming tomorrow's universal data exchange format and build tools for importing and exporting XML documents. Remarkably, the vigor and elegance of XML stems mainly from its simplicity: the basic component in XML is the *element* which is just a piece of text enclosed by matching tags such as `<author>` and `</author>`. Inside an element we may have 'raw' text, other elements, or a mixture of the two. As an example consider the XML document of Figure 1.1 representing bibliographic information. XML documents are self-describing in the following way: the tags make out the structure of the document, while the raw text determines the content. Usually, however, it does not make sense to allow any string as a tag or allow any possible nesting of elements. To create order out of chaos, XML provides Document Type Definitions (DTDs) to constrain documents. These are essentially context-free grammars that allow regular expressions

```
<bibliography>
    <book>
        <author>
            S. Abiteboul
        </author>
        <author>
            R. Hull
        </author>
        <author>
            V. Vianu
        </author>
        <title>
            Foundations of Databases
        </title>
        <publisher>
            Addison-Wesley
        </publisher>
        <year>
            1995
        </year>
    </book>
    <article>
        <author>
            E. Codd
        </author>
        <title>
            A Relational Model of Data for Large Shared Data Banks
        </title>
        <journal>
            Communications of the ACM
        </journal>
        <year>
            1970
        </year>
    </article>
</bibliography>
```

Figure 1.1: Example of an XML document describing bibliographic information.

```
<!ELEMENT bibliography (book | article)+>

<!ELEMENT article      (author+, title, journal, year)>

<!ELEMENT book         (author+, title, publisher, year)>

<!ELEMENT author       PCDATA>

<!ELEMENT title        PCDATA>

<!ELEMENT journal      PCDATA>

<!ELEMENT year         PCDATA>

<!ELEMENT publisher    PCDATA>
```

Figure 1.2: A DTD for the XML document in Figure 1.1

over non-terminals as right-hand sides of productions. The document in Figure 1.1, for instance, conforms to the DTD of Figure 1.2. This DTD says that a bibliography consists of a non-empty sequence of books and articles, and that an article should consist of a non-empty sequence of authors, followed by one title, one journal, and one year. For a book, the journal is replaced by a publisher. Finally, it specifies author, title, journal, year, and publisher as arbitrary string values.

A more abstract view of XML documents is given by a natural tree representation obtained by omitting end-tags. The tree representation of the document in Figure 1.1, for instance, is depicted in Figure 1.3. In this work, we therefore abstract away from the many bells and whistles, like attributes, references, style information, and so on, provided by XML (or any other document specification language). We naturally model structured documents as labeled trees where the children of a node are ordered, and that, additionally, conform to some grammar. So, a structured document is one such tree. Information retrieval systems [FBY92] usually query a set of structured documents instead of only one document. However, as far as query language design is concerned, a set of documents can be considered as a single long structured document.

This abstraction is in essence the same approach as adopted by the semi-structured data community [ABS99] with this difference that they do consider references and therefore have labeled graphs (as opposed to trees) as the underlying data model. However, they disregard the ordering of nodes which is for example inherent in the document of Figure 1.1: neglecting the order of the authors could seriously affect the information content of the data. Moreover, it is far from obvious how the current query languages for semi-structured data can be adapted to take such ordering into account. In fact, as stressed by Suciu [Suc98], dealing with this inherent ordering is an important research topic. One of the contributions of this work, therefore, is that

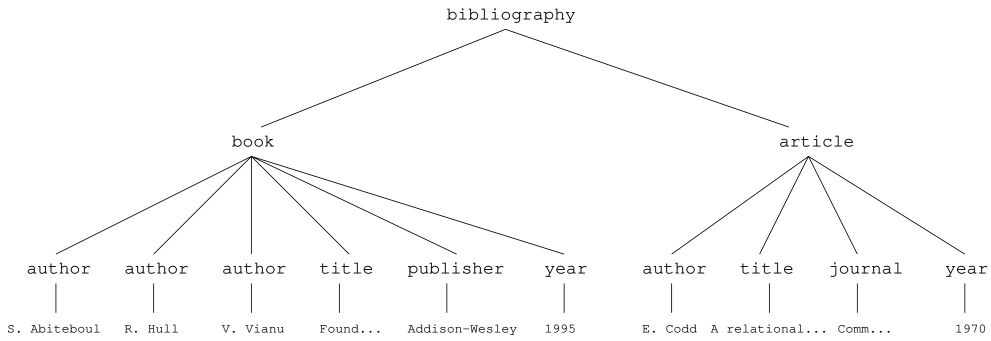

Figure 1.3: Tree representation of the XML document in Figure 1.1.

all studied query languages can deal with the inherent ordering of nodes without any difficulty.

Computations on trees have been studied in depth for the last 20 years by the formal language theory community [RS97] and many formalisms have been proposed. Since trees are natural abstractions of structured documents, and since queries in this framework are computations on trees, it is natural to revisit some of these established formalisms but now approach them from a query language perspective. In fact, the main objective of this dissertation is to examine how such formalisms can be used as a query language. More concretely, we study in this respect the expressiveness and optimization of various kinds of attribute grammars (Chapter 3 and Chapter 4) and tree automata (Chapter 5). Further, we apply the to this end developed techniques to (*i*) drastically improve the known upperbounds of various optimization problems of the Region Algebra [CM98a]; and (*ii*) provide evidence for the robustness of the actual XML transformation language XSLT [Cla99] (Chapter 6).

In this dissertation, we are mostly, but not exclusively, concerned with query languages expressing selection queries. By this we mean the retrieval of certain nodes in the tree corresponding to positions in the document or structural elements of the document. Such queries can also be seen as retrieving those subtrees in a document whose roots satisfy a certain pattern [BYN96, KM93, KM94, Mur98, NS98]. We refer to such queries as unary queries as they map a document to a set of its nodes. The interest in such queries is two-fold:

(*i*) The selection of interesting subtrees occurring in large documents is precisely the simple query facility provided by most information retrieval systems and therefore constitutes the most simple and common form of document querying.

(*ii*) Selection queries form the basis of more general query languages transforming documents into other documents. Indeed, most document query languages operating on trees or graphs, like, e.g., XML-QL, XSLT [DFF$^+$99, Cla99], specifically for XML, and Lorel, StruQL, and UnQL [AQM$^+$97, FFK$^+$98, BDHD96], for the semi-structured data model, have some kind of pattern language at their disposal for identifying the different parts of the document that have to be combined, possibly after some more manipulation, to obtain the output document. In all cases these pattern languages are based on regular path expressions. The query languages we propose are far more powerful than those, as we formally show in Chapter 6.

We next give a detailed overview of the content and the structure of the thesis.

1.2 Detailed overview

We start in **Chapter** 2, by recalling the necessary definitions concerning finite automata and logic. In particular, we reprove Büchi's Theorem stating that a regular string language is definable in monadic second-order logic (MSO) if and only if it is regular. We give the proof of Ladner [Lad77] based on MSO-equivalence types. This technique is fundamental for the technical development of the dissertation as it will be employed (in suitably generalized forms) in all later chapters.

In **Chapter** 3 we focus on documents modeled as derivation trees of (ordinary) context-free grammars (CFGs). This is the approach originally proposed by Gonnet and Tompa [GT87], and, in essence, is also the view taken by XML.

The classical formalism of *attribute grammars*, introduced by Knuth [Knu68], has always been a prominent framework for expressing computations on derivation trees. Attribute grammars provide a mechanism for annotating the nodes of a tree with so-called "attributes", by means of so-called "semantic rules" which can work either bottom-up (for so-called "synthesized" attribute values) or top-down (for so-called "inherited" attribute values). Attribute grammars are applied in such diverse fields of computer science as compiler construction and software engineering (for a survey, see [DJL88]).

Hence, it is natural to consider attribute grammars as a basis for structured document database languages. For instance, this approach was chosen by Abiteboul, Cluet and Milo [ACM98] and Kilpeläinen et al. [KLMN90]. None of them consider expressivity issues, though. Our goal in Chapter 3, therefore, is to understand the expressive power of attribute grammars as a structured document query language.

Note that derivation trees of context-free grammars are ranked. That is, the maximal number of children of a node is bounded by some constant depending on the grammar. We discuss attribute grammars for unranked trees in the next chapter.

We propose to use *Boolean-valued* attribute grammars (BAGs) to express the unary queries discussed in the previous section. BAGs are attribute grammars with Boolean attribute values, and with propositional logic formulas as semantic rules. A BAG indeed expresses a query in a natural way: the result of the query expressed by a BAG consists of those nodes in the tree for which some designated attribute is true.

We show that a unary query is expressible by a BAG if and only if it is definable in monadic second-order logic (MSO). We point out that this result was obtained independently by Bloem and Engelfriet [BE]. The only-if direction is easy to prove.

For the if direction, we show that BAGs can compute the MSO-equivalence type of a tree. As a corollary of the proof, we obtain that every BAG is equivalent to a BAG that consists of one bottom-up pass followed by one top-down pass. Additionally, one can use BAGs also to express Boolean queries. In this more restricted setting the equivalence between BAGs and MSO follows directly from the Doner-Thatcher-Wright Theorem which states that a tree language is regular if and only if it is definable be a tree automaton. From this equivalence then follows a *bottom-up property* for Boolean BAG queries: every Boolean query expressible by a BAG is already expressible by a BAG using synthesized attributes only. This bottom-up property does *not* hold for BAGs expressing unary queries.

Having understood the expressive power of BAGs, we then turn to queries that result in relations, of arbitrary fixed arity, among the nodes of the tree. These queries are for example used when one wants to define "wrappers" that map relevant parts of the document into a relational database [ACM98, MAM$^+$98, PGMW95]. To this end, we introduce *relation-valued* attribute grammars (RAGs), which use first-order logic (FO) formulas as semantic rules. The query expressed by a RAG is naturally defined as the value (a relation) of some designated attribute of the root. We show that the queries expressible by RAGs are precisely those definable by first-order inductions of linear depth. Results by Immerman [Imm89] imply that these are precisely the queries computable in linear time on a parallel random access machine with polynomially many processors.

We also investigate whether the above-mentioned bottom-up property for Boolean BAG queries carries over to Boolean RAG queries; using tools from finite model theory we prove that it does not.

We complete the picture by showing that synthesized RAGs are strictly more powerful than monadic second-order logic, for queries of *any* arity. This implies in particular that even when restricting attention to unary queries, RAGs are more powerful than BAGs. Moreover, it turns out that each query defined by a monadic second-order logic formula can be expressed by a RAG that uses only synthesized attributes.

We conclude Chapter 3 by considering Boolean-valued and relation-valued *relational* attribute grammars. Relational attribute grammars have been introduced by Courcelle and Deransart [CD88]. This concept is a generalization of standard attribute grammars, where the semantic rules do not specify functions, computing attributes in terms of other attributes, but rather specify relations among attributes. In this section we define relational extended AGs. To make a clear distinction between these and the attribute grammars considered before, we refer to the latter as functional attribute grammars. The main difference with functional BAGs and RAGs is that we now associate one propositional formula (in the case of BAGs) and one FO formula (in the case of RAGs) with each production, rather than with each position in a production and for each attribute. For one thing, this means that there is no longer a distinction between inherited and synthesized attributes. Further, attribute values are now defined implicitly. Hence, the result of a relational attribute grammar is no longer uniquely determined on every tree. We will show that relational BAGs and RAGs can express queries in various ways. Although they are seemingly much less restrictive than functional attribute grammars, we prove that they remain just as expressive in the case of Boolean-valued attributes. For RAGs, however, the situation is much less clear; under various semantics, relational RAGs capture complexity classes such as NP, coNP and UP $\cap$ coUP, whose relationship to the linear parallel time complexity class of functional RAGs is unknown.

The results obtained in this chapter are summarized graphically in Figure 1.4. An arrow from a class of queries C to a class of queries C', means $C \subseteq C'$. A negated arrow from C to C', means there is a Boolean query in C that is not in C'.

Extended abstracts containing some of the results in this chapter are published as [NV98] and [Nev98].

$$
\begin{tikzcd}
\exists\text{-RAG} = \mathrm{NP} & \forall\text{-RAG} = \mathrm{coNP} & \mathrm{IRAG} = \mathrm{UP} \cap \mathrm{coUP} \\
& \mathrm{RAG} = \text{PFP-LIN} = \mathrm{CRAM}[n] \arrow[ul] \arrow[u] \arrow[ur] \arrow[d, shift left, "/" marking] & \\
& \text{synthesized RAG} \arrow[u, shift left] \arrow[d, shift left, "/" marking] & \\
& \mathrm{BAG} = \exists\text{-BAG} = \forall\text{-BAG} = \mathrm{IBAG} = \mathrm{MSO} \arrow[u, shift left] \arrow[d, shift left, "/" marking, "\text{non-Boolean queries}"] & \\
& \text{synthesized BAG} \arrow[u, shift left] &
\end{tikzcd}
$$

Figure 1.4: Summary of results on BAGs and RAGs.

In **Chapter 4** we shift attention to documents defined by *extended* context-free grammars rather than context-free grammars. The former correspond more closely to the XML DTDs mentioned in the previous section. More precisely, extended context-free grammars (ECFG) are context-free grammars having regular expressions over grammar symbols on the right-hand side of productions. It is known that ECFGs generate the same class of string languages as CFGs. Hence, from a formal language point of view, ECFGs are nothing but shorthands for CFGs. However, when grammars are used to model documents, i.e., when also the derivation trees are taken into consideration, the difference between CFGs and ECFGs becomes apparent. Indeed, compare Figure 1.5 and Figure 1.6. They both model a list of poems, but the CFG

DB $\rightarrow$ PoemList
PoemList $\rightarrow$ Poem PoemList
PoemList $\rightarrow$ Poem
Poem $\rightarrow$ VerseList
VerseList $\rightarrow$ Verse VerseList
VerseList $\rightarrow$ Verse
Verse $\rightarrow$ WordList
WordList $\rightarrow$ Word WordList
WordList $\rightarrow$ Word
Word $\rightarrow$ LetterList
LetterList $\rightarrow$ Letter LetterList
LetterList $\rightarrow$ Letter
Letter $\rightarrow$ a $|\ \ldots\ |$ z

Figure 1.5: A CFG modeling a list of poems.

DB $\rightarrow$ Poem^{+}
Poem $\rightarrow$ Verse^{+}
Verse $\rightarrow$ Word^{+}
Word $\rightarrow$ $(\mathrm{a} + \cdots + \mathrm{z})^{+}$

Figure 1.6: An ECFG modeling a list of poems.

needs the extra non-terminals PoemList, VerseList, WordList, and LetterList to allow for an arbitrary number of poems, verses, words, and letters. These non-terminals, however, have no meaning at the level of the logical specification of the document.

A fundamental difference between derivation trees of CFGs and derivation trees of ECFGs is that the former are ranked while the latter are not. In other words, nodes in a derivation tree of an ECFG need not have a fixed maximal number of children. While ranked trees have been studied in depth [GS97, Tho97b], unranked trees only recently received new attention in the context of SGML and XML. Based on work of Pair and Quere [PQ68] and Takahashi [Tak75], Murata defined a bottom-up automaton model for unranked trees [Mur95]. This required describing transition functions for an arbitrary number of children. Murata's approach is the following: a node is assigned a state by checking the sequence of states assigned to its children for membership in a regular language. In this way, the "infinite" transition function is represented in a finite way. We will extend this idea to attribute grammars. See the work of Brüggemann-Klein, Murata and Wood for an extensive study of tree automata over unranked trees [BKMW98].

Inspired by the idea of representing transition functions for automata on unranked trees as regular string languages, we introduce extended attribute grammars (extended AGs) working directly over ECFGs rather than over standard CFGs. They express unary queries much in the same way as BAGs do. The main obstacle in defining attribute grammars for ECFGs is that the right-hand sides of productions contain regular expressions that, in general, specify infinite string languages. This gives rise to two problems for the definition of extended AGs that are not present for standard AGs:

(i) in a production, there may be an unbounded number of grammar symbols for which attributes should be defined; and

(ii) the definition of an attribute should take into account that the number of attributes it depends on may be unbounded.

We resolve these problems in the following way. For (i), we only consider unambiguous regular expressions in the right-hand sides of productions.[1] Informally, this means that every child of a node derived by the production $p = X \to r$ corresponds to exactly one position in r. We then define attributes uniformly for every position in r and for the left-hand side of p. For (ii), we only allow a finite set D as the semantic domain of the attributes and we represent semantic rules as regular languages over D much in the same way as tree automata over unranked trees are defined. By carefully tailoring the semantics of inherited attributes, extended AGs can take into account the inherent order of the children of a node in a document.

Chapter 4 is further organized as follows. First, we introduce extended attribute grammars as a query language for structured document databases defined by ECFGs. Queries in this query language can be evaluated in time quadratic in the number of nodes of the tree. We further show that non-circularity, the property that an attribute grammar is well-defined for every tree, is in EXPTIME. Interestingly, the naive reduction of the non-circularity problem of extended AGs to the same problem for standard AGs gives rise to a double exponential algorithm. We obtain an EXPTIME upper bound by reducing the problem to the problem of deciding whether a tree-walking automaton (over unranked trees) cycles. We then show the latter problem to be complete for EXPTIME. The EXPTIME upper bound for the non-circularity test of extended AGs is also a lower bound since deciding non-circularity for standard attribute grammar is already known to be hard for EXPTIME [JOR75].

Next, we generalize our earlier results on BAGs by showing that extended AGs express precisely the unary queries definable in MSO. Like for BAGs, we show that extended AGs can compute the MSO-equivalence type of the input tree. The only complication arises from the fact that trees are now unranked.

Hereafter, we obtain the exact complexity of some relevant optimization problems for extended AGs. In particular, we establish the EXPTIME-completeness of the non-emptiness (given an extended AG, does there exist a tree of which a node is selected by this extended AG?) and of the equivalence problem of extended AGs. Interestingly, in obtaining this result and the previous complexity result, we make use of nondeterministic two-way automata with a pebble to succinctly describe regular

[1] This is no loss of generality, as any regular language can be denoted by an unambiguous regular expression [BEGO71]. SGML is even more restrictive as it allows only *one*-unambiguous regular languages [BKD98, Woo95].

string languages. The crucial property of those, is that they can be transformed into nondeterministic one-way automata with only exponential size increase, as opposed to the expected double exponential size increase. The latter is a result due to Globerman and Harel [GH96]. Moreover, the thus obtained complexity results will be exploited in Chapter 6 to drastically improve the upper bound on the complexity of the equivalence problem of Region Algebra expressions obtained by Consens and Milo [CM98a].

We conclude Chapter 4 by considering *relational* extended AGs which are a generalization of the relational BAGs studied in Chapter 3. Specifically, we show that they remain just as expressive as extended AGs.

An extended abstract containing the results in this chapter is published as [Nev99].

In **Chapter 5**, we abandon attribute grammars and turn to another well-studied computation model for trees: the tree automaton [GS97, Tho97b]. In particular, we want to understand how such automata, on both ranked and unranked trees, can be used to express unary structured document queries. Concretely, we define a *query automaton* (QA) as a two-way deterministic finite automaton over trees that can select nodes depending on the state and the label at those nodes. In fact, a QA can express queries in a natural way: the result of a QA on a tree consists of all those nodes that are selected during the computation of the QA on that tree.

We stress that the query automata we consider are quite different from the tree acceptors studied in formal language theory [GS97]. For one thing, two-way tree automata are equivalent to one-way ones [Mor94], but it is not so difficult to see that query automata are not equivalent to bottom-up ones. Indeed, a bottom-up QA, for example, cannot express the query "select all leaves if the root is labeled with σ", simply because it cannot know the label of the root when it starts at the leaves. More surprising, however, is that in the unranked case various QA formalisms accept the same class of tree languages,[2] while not expressing the same class of queries. This indicates a substantial difference between looking at automata from a formal language point of view (i.e., for defining tree languages) and looking at automata from a database point of view (i.e., for expressing queries).

To warm up, we first consider query automata on strings which are simply two-way deterministic automata extended with a selection function. This approach allows us to introduce some important proof techniques in an easy setting which then later will be generalized to obtain our main results. These techniques can be summarized as follows: (*i*) capturing the behavior of two-way automata by means of behavior functions; and (*ii*) computing MSO-equivalence types relevant for expressing unary queries by automata.

Recall that computation of MSO-equivalence types has been the main technique in obtaining our expressiveness results in the previous chapters. However, at least in the context of unary queries, the computation of MSO-equivalence types by automata is much more involved than in the case of attribute grammars. The main reason for this is that attribute grammars can store the MSO-equivalence types of subtrees

[2] A *tree language* is a set of trees. We say that a QA *accepts a tree* if the underlying tree automaton accepts it.

and envelopes[3] needed to compute the MSO-equivalence type of the whole tree, in the attributes at the relevant nodes. Query automata, on the other hand, basically have to recompute the MSO-equivalence types of each of these components whenever they need it. This recomputation itself poses no problem: the major difficulty is the relocation of the starting point from where each subcomputation originated. To this end, we will make use of a powerful lemma on two-way string automata obtained by Hopcroft and Ullman [HU67].

Next, we consider query automata over ranked and unranked trees. A QA^r(r stands for *ranked*) is a two-way deterministic tree automaton[4] as defined by Moriya [Mor94] extended with a selection function. As hinted upon above, we show that these automata can express exactly the unary queries definable in MSO. Naturally, a first approach to define query automata for unranked trees, is to add a selection function to the two-way deterministic tree automata over unranked trees defined by Brüggemann-Klein, Murata and Wood [BKMW98]. We denote these automata with QA^u (u stands for *unranked*). Surprisingly, although these automata can accept all recognizable tree languages, they cannot even express all unary queries definable in first-order logic. Intuitively, when the automaton makes a down transition at some node $\mathbf{n}$, it assigns a state to every child of $\mathbf{n}$; although every child knows its own state, it cannot know in general which states are assigned to its siblings (as there can be arbitrarily many of them). This means that in the unranked case not enough information can be passed from one sibling to another. To resolve this, we introduce generalized "stay-transitions" where a two-way string-automaton reads the string formed by the states at the children of a certain node, and then outputs a new state for each child. An automaton making at most one stay-transition (or, equivalently, a constant number of stay-transitions) for the children of each node is a *strong* QA^u (SQA^u). We show that these automata compute exactly all MSO-definable queries. Thus, while QA^u and SQA^u recognize the same tree languages, they do *not* compute the same queries. Moreover, the restriction on the number of stay-transitions is necessary: without any such restriction, SQA^us could simulate linear space Turing Machines.

While the general problem of deciding whether two queries are equivalent or the result of a query is always empty is usually undecidable, their language-theoretic counter parts, equivalence and emptiness of automata, are well-known to be decidable. Therefore, we investigate in Section 5.4 the complexity of the following two problems: (i) *Given a QA, does there exist a tree for which there is a node that is selected?* (non-emptiness) (ii) *Given two QAs, do they express the same query?* (equivalence). One cannot hope to do better than EXPTIME for these decision problems, as non-emptiness of two-way deterministic tree automata over ranked trees, i.e., even without selecting nodes, is already complete for EXPTIME. We show that the non-emptiness and the equivalence problem of all query automata studied in this paper are in EXPTIME. Interestingly, like in the previous chapter, we again make use

[3]The envelope of a tree $\mathbf{t}$ at a node $\mathbf{n}$ is the tree obtained from $\mathbf{t}$ by removing all the subtrees rooted at the children of $\mathbf{n}$.

[4]These automata are very different from the (alternating) tree-*walking* automata used in, e.g., [Var89].

of nondeterministic string automata with one pebble to obtain the above result for query automata on unranked trees.

We end the chapter by considering nondeterministic query automata which can express queries in various ways. We focus on two semantics: *existential*, a node is selected if it is selected in *at least one* accepting run of the automaton, and *universal*, a node is selected if it is selected in *all* accepting runs. We show that both semantics for nondeterministic bottom-up (top-down) automata capture precisely the queries definable in MSO. Hence, if nondeterminism is added, the automata need only to move in one direction to express all of MSO.

An extended abstract containing some of the results of this chapter is published as [NS99].

We apply the techniques developed in this dissertation and discuss some related work in **Chapter 6**.

First, we show that Region Algebra expressions (introduced by Consens and Milo [CM98a]) can be simulated by extended AGs. Stated as such, the result is hardly surprising, since the former essentially corresponds to a fragment of first-order logic over trees while the latter corresponds to full MSO. We, however, exhibit an *efficient* translation, which gives rise to a drastic improvement on the complexity of the equivalence problem of Region Algebra expressions. To be precise, the algorithm proposed by Consens and Milo first translates each Region Algebra expression into an equivalent first-order logic formula on trees and then invokes the known algorithm testing decidability of such formulas. Unfortunately, the latter algorithm has non-elementary complexity. That is, the complexity of this algorithm cannot be bounded by an elementary function (i.e., an iterated exponential $2\hat{}(2\hat{} \ldots (2^n))$ where n is the size of the input). This approach therefore conceals the real complexity of the equivalence test of Region Algebra expressions. Our efficient translation of Region Algebra expressions into extended AGs, however, gives an EXPTIME algorithm. The thus obtained upper bound more closely matches the coNP lower bound [CM98a]. This result is published in [Nev99]

Next, we apply the techniques used to obtain our expressiveness results to the actual XML transformation language XSLT [Cla99]. Specifically, we show that XSLT has the ability to issue any MSO pattern at any node in the document. That is, when XSLT arrives at a node it can decide for any unary MSO formula $\varphi(x)$ whether this formula holds at that node and use this information for further processing of the document. Stated as such the result is hardly surprising since full-fledged XSLT allows to call arbitrary Java programs and, therefore, can express all computable document transformations. Our aim, however, is to stress that the navigational mechanism of XSLT together with a restricted use of variables already suffices to capture the expressiveness of MSO. Hereby, on the one hand, we reveal that core XSLT has a very powerful pattern language at its disposal, and, on the other hand, provide evidence for the robustness of the language.

Further, we compare MSO with other query languages for structured documents and semi-structured data. Most of the current query languages, like for example, Lorel, UnQL, and XML-QL [ABS99], use regular path expressions to navigate through the input database. Since our extended AGs and our QAs can only express unary queries, they cannot be compared as such with the above mentioned query languages. Therefore, we use first-order logic with regular path expressions, denoted by $\mathrm{FO}^{\mathrm{reg}}$, as an abstraction for the latter. To be precise, for each regular expression r over some alphabet, we have the binary predicate $r(x, y)$ with the following meaning that x is an ancestor of y and the string consisting of the symbols on the path from x to y belongs to the language defined by r. Such a logic can, hence, only look along paths in trees, and not at the tree as a whole. We confirm this intuition by formally proving that MSO is strictly more expressive than $\mathrm{FO}^{\mathrm{reg}}$ with regular path expressions. In fact, we show that no $\mathrm{FO}^{\mathrm{reg}}$ formula can define the class of trees representing Boolean circuits evaluating to true.

Next, we elaborate on the various ways in which unranked trees can be coded by ranked ones.

We end the chapter by proposing a design to implement the BAGs and RAGs previously studied in Chapter 3. More specifically, we want to show that deductive databases offer a natural platform on top of which such an implementation becomes remarkably straightforward. Since BAGs can be seen as special cases of RAGs, we focus attention to the latter ones.

Concretely, we represent a context-free grammar by a relational schema, so that structured documents can be stored as instances over this schema. We then translate a RAG into a set of deductive rules, which in general contain negation. We prove the somewhat surprising result that the naive bottom-up fixpoint procedure suffices to capture the semantics of the RAG. This procedure (the standard one for deductive programs without negation) is usually not considered in the presence of negation, as it is not even guaranteed to terminate; nevertheless, we show that it does on the programs generated by our translation. In fact, the programs resulting from the translation of RAGs become acyclic after removing the rules inconsistent with the extensional database predicates. That is, the predicates encoding the derivation tree at hand. In particular, this means that these programs are modularly stratified [Ros94] which further implies that we can use any deductive database system supporting the well-founded semantics to implement RAGs. An extended abstract describing only the translation of BAGs to deductive rules is published as [NV97].

We present some concluding remarks in **Chapter** 7.

2

Basics of logic and automata on strings and trees

Many of the formalisms we consider in later chapters will be compared with monadic second-order logic (MSO). This logic is the well-known extension of first-order logic (FO) with set quantification. MSO has been laboriously used to characterize the expressiveness of various concepts in formal language theory like automata on strings, trees, and graphs [Tho97b].

In this chapter we recall some basic facts on MSO and use them to reprove Büchi's Theorem [Büc60] stating that a string language is regular if and only if it is definable in MSO. This approach allows us to introduce various techniques related to MSO and automata in an easy setting which we will generalize in later chapters to obtain expressiveness results for attribute grammars and query automata. Specifically, we recall in this chapter how Ehrenfeucht games facilitate reasoning on MSO-equivalence types. These types constitute the building blocks of all simulations of MSO formulas in later chapters. Along the way, we introduce another important concept: the two-way nondeterministic finite automaton with one pebble. Such an automaton is a powerful tool to define regular languages. We use it to obtain some of the complexity results in Chapter 4 and Chapter 5. Finally, we define bottom-up tree automata and reprove the generalization of Büchi's Theorem to trees obtained by Doner, Thatcher and Wright [Don70, TW68].

Before we start we make the following conventions. We denote by $\mathbb{N}$ the set of positive natural numbers. Further, if S is a set then we denote by $|S|$ its cardinality.

2.1 Monadic second-order logic

A *vocabulary* τ is a finite nonempty set of constant symbols and relation names with associated arities. As usual, a *τ-structure* $\mathcal{A}$ consists of a finite set A, the domain of $\mathcal{A}$, together with

- an interpretation $R^{\mathcal{A}} \subseteq A^r$ for each relation name R in τ; here, r is the arity of R; and
- an interpretation $c^{\mathcal{A}} \in A$ for each constant symbol in τ.

When τ is clear from the context or is not important, we just say structure rather than τ-structure. Sometimes, when the structure $\mathcal{A}$ is understood, we abuse notation and write R for the relation $R^{\mathcal{A}}$.

Example 2.1 Let τ be the vocabulary consisting of a binary relation symbol E and two constants s and t. Let $\mathcal{G}$ be the τ-structure with domain $G = \{1, \ldots, n\}$, $E^{\mathcal{G}} := \{(i,j) \in G \times G \mid i + 1 = j\}$, $s^{\mathcal{G}} = 1$, and $t^{\mathcal{G}} = n$. Then $\mathcal{G}$ represents the graph that is a chain of n elements with the source and the target being the first and last element, respectively. ■

Monadic second-order logic (MSO) allows the use of *set variables* ranging over sets of domain elements, in addition to the individual variables ranging over the domain elements themselves as provided by first-order logic. We will assume some familiarity with this logic and refer the unfamiliar reader to the book of Ebbinghaus and Flum [EF95] or the chapter by Thomas [Tho97b].

Example 2.2 We give an example of an MSO formula. As usual we denote set variables by capital letters and first-order variables by small letters. Let $\varphi(x,y)$ be the following MSO formula over the vocabulary of Example 2.1:

$$(\exists X)(X(x) \wedge X(y) \wedge (\forall z_1)(\forall z_2)((X(z_1) \wedge E(z_1, z_2)) \rightarrow X(z_2))).$$

This formula defines the transitive closure of E. Indeed, for each graph $\mathcal{G}$ with nodes $\mathbf{n}$ and $\mathbf{m}$, we have that $\mathcal{G} \models \varphi[\mathbf{n}, \mathbf{m}]$ iff there exists a set of nodes B of $\mathcal{G}$ containing both $\mathbf{n}$ and $\mathbf{m}$, and which contains every element adjacent to an element of B via the edge relation E. In other words, $\mathcal{G} \models \varphi[\mathbf{n}, \mathbf{m}]$ iff there exists a path in $\mathcal{G}$ from $\mathbf{n}$ to $\mathbf{m}$. Hence, the MSO sentence $\varphi(s,t)$ defines those graphs for which there is a path from the source to the target. ■

In the following we will make use of some basic facts about MSO. For a tuple $\bar{a} = a_1, \ldots, a_n$ of elements in $\mathcal{A}$, we write $(\mathcal{A}, \bar{a})$ to denote the finite structure that consists of $\mathcal{A}$ with $a_1, \ldots, a_n$ as distinguished constants. Let $\mathcal{A}$ and $\mathcal{B}$ be two structures, let $\bar{a}$ and $\bar{b}$ be tuples of elements in $\mathcal{A}$ and $\mathcal{B}$, respectively, and let k be a natural number. Then we write $(\mathcal{A}, \bar{a}) \equiv_k^{\text{MSO}} (\mathcal{B}, \bar{b})$ and say that $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$ are *$\equiv_k^{\text{MSO}}$-equivalent*, if for each MSO sentence φ of quantifier depth at most k it holds

$$(\mathcal{A}, \bar{a}) \models \varphi \quad \Leftrightarrow \quad (\mathcal{B}, \bar{b}) \models \varphi.$$

That is, $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$ cannot be distinguished by MSO sentences of quantifier depth (at most) k. It readily follows from the definition that $\equiv_k^{\mathrm{MSO}}$ is an equivalence relation. Moreover, $\equiv_k^{\mathrm{MSO}}$-equivalence can be nicely characterized by Ehrenfeucht games.

The *k-round MSO game* on two structures $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$, denoted by $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$, is played by two players, the *spoiler* and the *duplicator*, in the following way. In each of the k rounds the spoiler decides to make a *point move* or a *set move*. If the i-th move is a point move, then the spoiler selects an element $c_i \in A$ or $d_i \in B$ and the duplicator answers by selecting one element of the other structure. When the i-th move is a set move, the spoiler chooses a set $P_i \subseteq A$ or $Q_i \subseteq B$ and the duplicator chooses a set in the other structure. After k rounds there are elements $c_1, \ldots, c_\ell$ and $d_1, \ldots, d_\ell$ that were chosen in the point moves in A and B respectively and there are sets $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_n$ that were chosen in the set moves in A and B, respectively. The duplicator now wins this play if the mapping which maps c_i to d_i is a partial isomorphism from $(\mathcal{A}, \bar{a}, P_1, \ldots, P_n)$ to $(\mathcal{B}, \bar{b}, Q_1, \ldots, Q_n)$. That is, for all i and j, $c_i \in P_j$ iff $d_i \in Q_j$, and for every atomic formula $\varphi(\bar{x})$ containing no set variable, $\mathcal{A} \models \varphi[\bar{c}, \bar{a}]$ iff $\mathcal{B} \models \varphi[\bar{d}, \bar{b}]$.

We say that the duplicator has a *winning strategy* in $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$, or shortly that he wins $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$, if he can win each play no matter which choices the spoiler makes.

The following fundamental proposition is well known (see, e.g., [EF95] for a proof).

Proposition 2.3 *The duplicator wins $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$ if and only if*

$$(\mathcal{A}, \bar{a}) \equiv_k^{\mathrm{MSO}} (\mathcal{B}, \bar{b}).$$

It is well known that the relation $\equiv_k^{\mathrm{MSO}}$ has only a finite number of equivalence classes. We denote the set of these classes by Φ_k and refer to the elements of Φ_k by $\equiv_k^{\mathrm{MSO}}$*-types* (here we fix some number of parameters added to structures as fixed constants). We denote by $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ the $\equiv_k^{\mathrm{MSO}}$-type of a structure $\mathcal{A}$ with the elements in $\bar{a}$ as distinguished constants; thus, $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ is the equivalence class of $(\mathcal{A}, \bar{a})$ w.r.t. $\equiv_k^{\mathrm{MSO}}$. By $\tau_k^{\mathrm{MSO}}(\mathcal{A})$ we denote the $\equiv_k^{\mathrm{MSO}}$-type of the structure $\mathcal{A}$ without distinguished elements. It is often useful to think of $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ as the set of MSO-sentences of quantifier depth k that hold in $(\mathcal{A}, \bar{a})$. That is, we also view $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ as the set $\{\varphi \mid (\mathcal{A}, \bar{a}) \models \varphi\}$ of MSO sentences of quantifier depth k. It is well known that, upon logical equivalence, there are only a finite number of MSO sentences of quantifier depth k. Moreover, there exists a normal-form for all those non-equivalent formulas. From now on we tacitly assume that all MSO formulas are in this normal-form. In this way, we can simple say $\varphi \in \tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ instead of 'there is a formula in $\tau_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a})$ which is logically equivalent to φ'.

Equivalence types will be the main tool to simulate MSO formulas by automata. To illustrate their usage, we will recall in the next section how they can be employed to prove Büchi's Theorem [Büc60].

First, we mention for later use the *k-round FO game* on two structures $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$, denoted by $G_k^{\mathrm{FO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$, which is the restriction of $G_k^{\mathrm{MSO}}(\mathcal{A}, \bar{a}, \mathcal{B}, \bar{b})$

to point moves only. We write $(\mathcal{A}, \bar{a}) \equiv_k^{\mathrm{FO}} (\mathcal{B}, \bar{b})$ and say that $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$ are $\equiv_k^{\mathrm{FO}}$-*equivalent*, if for each FO sentence φ of quantifier depth at most k we have

$$(\mathcal{A}, \bar{a}) \models \varphi \quad \Leftrightarrow \quad (\mathcal{B}, \bar{b}) \models \varphi.$$

That is, $(\mathcal{A}, \bar{a})$ and $(\mathcal{B}, \bar{b})$ cannot be distinguished by FO sentences of quantifier depth (at most) k. Analogous to Proposition 2.3, we have the following (see, e.g., [EF95], for a proof):

Proposition 2.4 *The duplicator wins $G_k^{\mathrm{FO}}(\mathcal{A}, \bar{a}; \mathcal{B}, \bar{b})$ if and only if*

$$(\mathcal{A}, \bar{a}) \equiv_k^{\mathrm{FO}} (\mathcal{B}, \bar{b}).$$

2.2 Queries

Definition 2.5 Let k be a natural number. A *k-ary query* is a function $\mathcal{Q}$ that maps each structure $\mathcal{A}$ to a k-ary relation over its domain. If $\mathcal{Q}$ is a nullary query, i.e., k is zero, then we also say that $\mathcal{Q}$ is a *Boolean* query.

MSO can be used to define queries in a straightforward way: if $\varphi(x_1, \ldots, x_k)$ is an MSO-formula then $\varphi(x_1, \ldots, x_k)$ defines the k-ary query $\mathcal{Q}$ defined by

$$\mathcal{Q}(\mathcal{A}) = \{(a_1, \ldots, a_k) \mid \mathcal{A} \models \varphi[a_1, \ldots, a_k]\}.$$

2.3 Regular string languages

In the following Σ denotes a finite alphabet. A *string* $w = \sigma_1 \cdots \sigma_n$ over Σ is a sequence of Σ-symbols. We denote the length of w by $|w|$ and for each $i \in \{1, \ldots, |w|\}$, we denote σ_i by w_i. We refer to $\{1, \ldots, |w|\}$ as the set of *positions* of w.

To define sets of strings by MSO formulas, we associate to each string w over Σ, a finite structure with domain $\{1, \ldots, |w|\}$, denoted by $\mathrm{dom}(w)$, over the binary relation symbol $<$, and the unary relation symbols $(O_\sigma)_{\sigma \in \Sigma}$. The interpretation of $<$ is the obvious one, and for each $\sigma \in \Sigma$, O_σ is the set of positions labeled with a σ, i.e., $O_\sigma = \{i \mid w_i = \sigma\}$. In the following, we will make no distinction between the string w and the relational structure that corresponds to it.

A *nondeterministic finite automaton* M (NFA) over Σ is a tuple $(S, \Sigma, \delta, I, F)$ where S is finite set of states, $\delta : S \times \Sigma \to 2^S$ is the transition function, $I \subseteq S$ is the set of initial states, and $F \subseteq S$ is the set of final states. We denote the canonical extension of the transition function to strings by δ^*. A string $w \in \Sigma^*$ is accepted by M if $\delta^*(s_0, w) \in F$ for an $s_0 \in I$. The language accepted by M, denoted by $L(M)$, is defined as the set of all strings accepted by M. The size of M is defined as $|S| + |\Sigma|$. A string language is *regular* if it is accepted by an NFA.

A *state assignment* ρ for a string $w \in \Sigma^*$ is a mapping from $\{1, \ldots, |w|\}$ to S. A state assignment ρ for w is *valid* if there exists an $s_0 \in I$ such that $\rho(1) \in \delta(s_0, w_1)$,

$\rho(|w|) \in F$, and for $i = 1, \dots, |w|-1$, $\rho(i+1) \in \delta(\rho(i), w_{i+1})$. Clearly, w is accepted by M if and only if there exists a valid state assignment for w.

If $|I| = 1$ and $|\delta(s,\sigma)| \leq 1$ for all $s \in S$ and $\sigma \in \Sigma$, then M is a *deterministic* finite automaton (DFA) and we treat δ as a function $S \times \Sigma \to S$. Additionally, we write s_0 in the definition of M when $I = \{s_0\}$.

We will reprove Büchi's Theorem [Büc60] stating that a string language is regular if and only if it can be defined by an MSO sentence. Here, an MSO sentence φ defines the language $\{w \in \Sigma^* \mid w \models \varphi\}$. The simulation of an MSO sentence by a DFA will be based on the computation of equivalence types. To this end we will use the following proposition which in particular says that $\tau_k^{\mathrm{MSO}}(\sigma_1 \cdots \sigma_{n-1}\sigma_n)$ only depends on $\tau_k^{\mathrm{MSO}}(\sigma_1 \cdots \sigma_{n-1})$ and $\tau_k^{\mathrm{MSO}}(\sigma_n)$. This observation will be used in the proof of Theorem 2.7 to define the transition function of the DFA computing the $\equiv_k^{\mathrm{MSO}}$-type of input strings.

By using the above Ehrenfeucht games and Proposition 2.3, we can easily show the following.

Proposition 2.6 *Let $k \geq 0$ and let w, v, w' and v' be strings. If $w \equiv_k^{\mathrm{MSO}} w'$ and $v \equiv_k^{\mathrm{MSO}} v'$, then $w \cdot v \equiv_k^{\mathrm{MSO}} w' \cdot v'$.*

Proof. By Proposition 2.3 it suffices to show that the duplicator wins $G_k^{\mathrm{MSO}}(w \cdot v; w' \cdot v')$. We already know that he wins the subgames $G_k^{\mathrm{MSO}}(w; w')$ and $G_k^{\mathrm{MSO}}(v; v')$. The duplicator, therefore, plays in $G_k^{\mathrm{MSO}}(w \cdot v; w' \cdot v')$ according to his winning strategies in $G_k^{\mathrm{MSO}}(w; w')$ and $G_k^{\mathrm{MSO}}(v; v')$. We make this strategy precise, but only consider moves of the spoiler on the string $w \cdot v$. Responses to moves where the spoiler picks elements in $w' \cdot v'$ can be treated similarly. If the spoiler chooses an element in w (v) then the duplicator answers according to his winning strategy in $G_k^{\mathrm{MSO}}(w; w')$ ($G_k^{\mathrm{MSO}}(v; v')$). If the spoiler makes a set move and chooses $P_1 \cup P_2$ in $w \cdot v$, where P_1 and P_2 contain the elements in w and v, respectively, then the duplicator chooses sets Q_1 and Q_2 in w' and v' according to his winning strategy in $G_k^{\mathrm{MSO}}(w; w')$ and $G_k^{\mathrm{MSO}}(v; v')$, respectively.

This is indeed a winning strategy. Let $c_1, \dots, c_\ell$ and $d_1, \dots, d_\ell$ be the elements chosen in point moves in $w \cdot v$ and $w' \cdot v'$, respectively, and let $P_1, \dots, P_r$ and $Q_1, \dots, Q_r$ be the sets of elements chosen in set moves in $w \cdot v$ and $w' \cdot v'$, respectively. By construction, the mapping $\bar{c} \mapsto \bar{d}$ restricted to the different components (w and w', and v and v') is a partial isomorphism between these corresponding components extended with the sets $\overline{P}$ and $\overline{Q}$. Hence, it only remains to check that the relation $<$ is preserved between elements coming from different components. This is always the case, as all elements of w (w') precede those of v (v'), the duplicator chooses elements in w' (v') whenever the spoiler chooses elements in w (v), and the duplicator chooses elements in w (v) whenever the spoiler chooses elements in w' (v'). ■

We are ready to prove Büchi's Theorem [Büc60]:

Theorem 2.7 *A language $L \subseteq \Sigma^*$ is regular if and only if it is definable in MSO.*

Proof. Suppose L is defined by the DFA $M = (S, \Sigma, \delta, s_0, F)$ with $S = \{0, \dots, n\}$ and $s_0 = 0$. We have to find an MSO sentence expressing for every string $w \in L(M)$ that M accepts w. On w, this sentence defines the run of M on w. Such a run is encoded by pairwise disjoint subsets $Z_0, \dots, Z_n$ of $\{1, \dots, |w|\}$ with the following intended meaning: $i \in Z_j$ iff $\delta^*(0, w_1 \cdots w_i) = j$. We say that Z_j labels position i with state j. Clearly, the run is accepting if $|w|$ is labeled with a final state. The sentence φ is then of the form

$$(\exists Z_0) \dots (\exists Z_n) \left(\psi(Z_1, \dots, Z_n) \wedge (\forall x) \left(\neg(\exists y)(x < y) \rightarrow \bigvee_{i \in F} Z_i(x) \right) \right).$$

Here, ψ is the FO formula that defines $Z_0, \dots, Z_n$ as the encoding of the run of M on the string under consideration. That is, it says that the first position should be labeled with the state $\delta(0, w_1)$ and that the other labelings should be consistent with the transition function. These are all local conditions and can, hence, readily be expressed in FO. The second part of φ expresses that the last element of the input string is labeled with a final state. A more computational view of φ is that, on input w, φ first *guesses* a state assignment and then *verifies*, by means of an FO formula, whether it has guessed correctly. That is, whether its guesses encode an accepting run of the automaton.

For the other direction we make use of types. A similar presentation was given by Ladner [Lad77]. The method presented here is also referred to as the composition method [Tho97a]. Let φ be an MSO sentence of quantifier depth k. Clearly, it suffices to know $\tau_k^{\mathrm{MSO}}(w)$ to determine whether $w \models \varphi$. We will now show that an automaton on input w can in fact compute $\tau_k^{\mathrm{MSO}}(w)$. The set of states is Φ_k, which is finite for every k. Proposition 2.6 says that for a string v and a Σ-symbol σ, $\tau_k^{\mathrm{MSO}}(v\sigma)$ only depends on $\tau_k^{\mathrm{MSO}}(v)$ and $\tau_k^{\mathrm{MSO}}(\sigma)$. Note that $\tau_k^{\mathrm{MSO}}(\sigma)$ only depends on σ. Hence, $\tau_k^{\mathrm{MSO}}(w)$ can be computed from left to right: the initial state is $\tau_k^{\mathrm{MSO}}(\varepsilon)$, that is, the $\equiv_k^{\mathrm{MSO}}$-type of the empty string; and, if the $\equiv_k^{\mathrm{MSO}}$-type of the string seen so far is θ and the next symbol is σ, then the automaton moves to state $\tau_k^{\mathrm{MSO}}(v\sigma)$ for a string v with $\tau_k^{\mathrm{MSO}}(v) = \theta$. By Proposition 2.6, it does not matter which member v of the $\equiv_k^{\mathrm{MSO}}$-equivalence class θ we take. Finally, the automaton accepts if $\varphi \in \theta$ with θ the state obtained after reading the last input symbol.

Formally, the automaton M accepting the language defined by φ is defined as $M = (\Phi_k, \Sigma, \delta, s_0, F)$, where $s_0 = \tau_k^{\mathrm{MSO}}(\varepsilon)$, $F = \{\theta \in \Phi_k \mid \varphi \in \theta\}$, and for all $\theta \in \Phi_k$ and $\sigma \in \Sigma$, $\delta(\theta, \sigma) = \tau_k^{\mathrm{MSO}}(v\sigma)$ for a string v with $\tau_k^{\mathrm{MSO}}(v) = \theta$. ■

In Section 5.1, we will make use of Lemma 2.9. We first provide a suitable generalization of Proposition 2.6.

Proposition 2.8 *Let $k \in \mathbb{N}$, let w and v be strings, let $i \in \{1, \dots, |w|\}$, and let $j \in \{1, \dots, |v|\}$. If $(w_1 \cdots w_i, i) \equiv_k^{\mathrm{MSO}} (v_1 \cdots v_j, j)$ and $(w_i \cdots w_{|w|}, 1) \equiv_k^{\mathrm{MSO}} (v_j \cdots v_{|v|}, 1)$, then $(w, i) \equiv_k^{\mathrm{MSO}} (v, j)$.*

Proof. We just combine the winning strategies in the subgames $G_k^{\mathrm{MSO}}(w_1 \cdots w_i, i; v_1 \cdots v_j, j)$ and $G_k^{\mathrm{MSO}}(w_i \cdots w_{|w|}, 1; v_j \cdots v_{|v|}, 1)$ to obtain a winning strategy in the game $G_k^{\mathrm{MSO}}(w,i;v,j)$, like in the proof of Proposition 2.6. We have to be a bit careful as position i and j in $w_1\cdots w_{|w|}$ and $v_1\cdots v_{|v|}$, respectively, occur in both subgames $G_k^{\mathrm{MSO}}(w_1\cdots w_i,i;v_1\cdots v_j,j)$ and $G_k^{\mathrm{MSO}}(w_i\cdots w_{|w|},1;v_j\cdots v_{|v|},1)$. However, the combined strategy is well defined on these positions: the duplicator picks position i (j) when the spoiler picks position j (i) as he does so in both subgames, simply because the common positions occur as distinguished constants in the subgames.

The following example shows that i and j are really needed as distinguished constants. Consider the string aba and bab. Clearly, $ab \equiv_1^{\mathrm{MSO}} ba$ and $ba \equiv_1^{\mathrm{MSO}} ab$. However, $(aba,2) \not\equiv_1^{\mathrm{MSO}} (bab,2)$ as the distinguished constants do not even carry the same label. ∎

Using the above proposition we obtain the following lemma:

Lemma 2.9 *Let k be a natural number. There exists a DFA $M = (S,\Sigma,s_0,\delta,F)$ such that $\delta^*(w) = \tau_k^{\mathrm{MSO}}(w,|w|)$, for every string w.*

Proof. The automaton M just works like the automaton in the proof of Theorem 2.7. The only difference is that it has to take the distinguished constant into account. Therefore, M has $\Phi_k \cup \{s_0\}$ as set of states where s_0 is the start state and where Φ_k is the set of $\equiv_k^{\mathrm{MSO}}$-types with one distinguished position. By Proposition 2.8, $\tau_k^{\mathrm{MSO}}(w\sigma,|w|+1)$ only depends on $\tau_k^{\mathrm{MSO}}(w,|w|)$ and $\tau_k^{\mathrm{MSO}}(\sigma,1)$. Note that the latter only depends on σ. So, the transition function δ is defined as follows: for each $\sigma\in\Sigma$, $\delta(s_0,\sigma) = \tau_k^{\mathrm{MSO}}(\sigma,1)$ and for each $\theta\in\Phi_k$, $\delta(\theta,\sigma) = \tau_k^{\mathrm{MSO}}(w\sigma,|w|+1)$ where w is a string with $\tau_k^{\mathrm{MSO}}(w,|w|) = \theta$. ∎

We conclude this section by introducing the following important device. A *two-way nondeterministic finite automaton with one pebble* is an NFA that can move in two directions over the input string and that has one pebble which it can lay down on the input string and pick up later. We refrain from giving a formal definition of such automata as we will only use them informally to describe algorithmic computations. Blum and Hewitt [BH67] showed that such automata can only define regular languages. To prove our complexity results in Chapter 4 and Chapter 5, we will need the following stronger result obtained by Globerman and Harel [GH96, Proposition 3.2].

Proposition 2.10 *Every two-way nondeterministic finite automaton M with one pebble is equivalent to an NFA M' whose size is exponential in the size of M. In fact, the size of M' can be uniformly bounded by the function $p(|\Sigma|)\cdot 2^{q(|S|)}$, where p and q are polynomials, Σ is the alphabet, and S is the set of states of M. Additionally, M' can be constructed in time exponential in the size of M.*

2.4 Regular tree languages

In this work we are mainly concerned with trees where the children of a node are ordered and carry a label from some finite alphabet Σ. We refer to such trees as Σ-trees. We introduce some terminology.

Trees will be denoted by the boldface characters $\mathbf{t}$, $\mathbf{s}$, $\mathbf{s}_1, \ldots$, while nodes of trees are denoted by $\mathbf{n}$, $\mathbf{m}$, $\mathbf{n}_1, \ldots$. We use the following convention: if $\mathbf{n}$ is a node of a tree $\mathbf{t}$, then $\mathbf{n}i$ denotes the i-th child of $\mathbf{n}$. We denote the set of nodes of $\mathbf{t}$ by $\mathrm{Nodes}(\mathbf{t})$ and the root of $\mathbf{t}$ by $\mathrm{root}(\mathbf{t})$. Further, the *arity* of a node $\mathbf{n}$ in a tree, denoted by $\mathit{arity}(\mathbf{n})$, is the number of children of $\mathbf{n}$. We say that a tree $\mathbf{t}$ has *rank* m, for $m \in \mathbb{N}$, if $\mathit{arity}(\mathbf{n}) \leq m$ for every $\mathbf{n} \in \mathrm{Nodes}(\mathbf{t})$. For a node $\mathbf{n}$ in $\mathbf{t}$, the set of its children is denoted by $\mathrm{children}(\mathbf{n})$. The subtree of $\mathbf{t}$ rooted at $\mathbf{n}$ is denoted by $\mathbf{t}_\mathbf{n}$; the *envelope* of $\mathbf{t}$ at $\mathbf{n}$, that is, the tree obtained from $\mathbf{t}$ by deleting the subtrees rooted at the children of $\mathbf{n}$ is denoted by $\overline{\mathbf{t}_\mathbf{n}}$;[1] and, for each $\sigma \in \Sigma$, the tree consisting of just one node that is labeled with σ is denoted by $\mathbf{t}(\sigma)$. The *depth* of a node $\mathbf{n}$ is the number of nodes on the path from $\mathbf{n}$ to the root ($\mathbf{n}$ included, root not included). The *height* of $\mathbf{n}$ is the number of nodes on the longest path from $\mathbf{n}$ to a leaf ($\mathbf{n}$ included, leaf not included). Hence, the depth of the root and the height of a leaf are zero. We denote the label of $\mathbf{n}$ in $\mathbf{t}$ by $\mathrm{lab}_\mathbf{t}(\mathbf{n})$.

We end by introducing use the following notation. When σ is a symbol in Σ and $\mathbf{t}_1, \ldots, \mathbf{t}_n$ are Σ-trees, then $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ is the Σ-tree graphically represented by

$$\begin{array}{ccc} & \sigma & \\ \swarrow & \cdots & \searrow \\ \mathbf{t}_1 & & \mathbf{t}_n. \end{array}$$

Note that in the above definitions there is no a priori bound on the number of children that a node may have. In the next chapter, we restrict attention to trees of bounded rank (hereafter simply referred to as *ranked* trees). In the remaining chapters we consider trees without any bound on their rank. To make a clear distinction, we refer to them as *unranked* trees.

A Σ-tree $\mathbf{t}$ can be naturally viewed as a finite structure over the binary relation symbols E and $<$, and the unary relation symbols $(O_\sigma)_{\sigma \in \Sigma}$. The edge relation E is the obvious one. The relation $<$ specifies the ordering on the children for every node $\mathbf{n}$. Finally, for each $\sigma \in \Sigma$, O_σ is the set of nodes that are labeled with a σ.

We now define bottom-up deterministic tree automata and indicate how they can compute the $\equiv_k^{\mathrm{MSO}}$-types of trees. This will allow us to reprove the generalization of Büchi's Theorem for trees. The next definition is for Σ-trees of rank at most m, for some fixed m.

Definition 2.11 A *(bottom-up deterministic) tree automaton* (BDTA) is a triple $B = (Q, \Sigma, \delta, F)$, consisting of a finite set of states Q, a finite alphabet Σ, a set $F \subseteq Q$ of final states, and a transition function $\delta : \bigcup_{i=0}^{m} Q^i \times \Sigma \to Q$. The semantics of B on a tree $\mathbf{t}$, denoted by $\delta^*(\mathbf{t})$, is inductively defined as follows: if $\mathbf{t}$ consists of only one node labeled with σ then $\delta^*(\mathbf{t}) = \delta(\sigma)$; if $\mathbf{t}$ is of the form

$$\begin{array}{ccc} & \sigma & \\ \swarrow & \cdots & \searrow \\ \mathbf{t}_1 & & \mathbf{t}_n, \end{array}$$

[1] Note that $\mathbf{t}_\mathbf{n}$ and $\overline{\mathbf{t}_\mathbf{n}}$ have $\mathbf{n}$ in common.

then $\delta^*(\mathbf{t}) = \delta(\delta^*(\mathbf{t}_1), \ldots, \delta^*(\mathbf{t}_n), \sigma)$. A Σ-tree $\mathbf{t}$ is *accepted* by B if $\delta^*(\mathbf{t}) \in F$. The set of Σ-trees accepted by B is denoted by $L(B)$. A set of Σ-trees $\mathcal{T}$ is *recognizable* if there exists a tree automaton B, such that $\mathcal{T} = L(B)$.

To prove Theorem 2.13, we need a suitable generalization of Proposition 2.6 to trees. The proof of the next proposition is similar to the proof of the latter, but is a bit more subtle due to the presence of the edge relation E.

Proposition 2.12 *Let k be a natural number, $\sigma \in \Sigma$, and let $\mathbf{t}_1, \ldots, \mathbf{t}_n, \mathbf{s}_1, \ldots, \mathbf{s}_n$ be Σ-trees. If $\mathbf{t}_i \equiv_k^{\mathrm{MSO}} \mathbf{s}_i$, for $i = 1, \ldots, n$, then $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n) \equiv_k^{\mathrm{MSO}} \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$.*

Proof. Again, we just combine the winning strategies in the subgames $G_k^{\mathrm{MSO}}(\mathbf{t}_i; \mathbf{s}_i)$ to obtain a winning strategy in $G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n); \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n))$ as explained in the proof of Proposition 2.6. Additionally, we require that the duplicator picks the root in $\sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ whenever the spoiler picks the root of $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$, and vice versa. We now show that this strategy is winning. Suppose that in a play in

$$G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n); \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n))$$

the elements $\bar{c}$ and $\bar{d}$ are chosen in the point moves in $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, respectively, and the sets $\overline{P}$ and $\overline{Q}$, are chosen in set moves in $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, respectively. Clearly, the mapping $\bar{c} \mapsto \bar{d}$ restricted to the different components is a partial isomorphism between these corresponding components extended with the sets $\overline{P}$ and $\overline{Q}$. Hence, it only remains to check that the relations $<$ and E are preserved for elements coming from different components. We restrict attention to elements in $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$. Denote the roots of $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ and $\sigma(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ by $\mathbf{n}$ and $\mathbf{m}$, respectively.

Let c_i and c_j be elements coming from different components $\mathbf{t}_a$ and $\mathbf{t}_b$, with $c_i < c_j$ and $a, b \in \{1, \ldots, n\}$. Consequently, c_i and c_j are children of $\mathbf{n}$ and $a < b$. It, hence, suffices to show that d_i and d_j are the roots of $\mathbf{s}_a$ and $\mathbf{s}_b$, respectively.

First note the following. If the spoiler picks the root of $\mathbf{t}_a$ in his l-th move, with $l < k$, in $G_k^{\mathrm{MSO}}(\mathbf{t}_a; \mathbf{s}_a)$, then the duplicator is forced to answer with the root of $\mathbf{s}_a$. Indeed, if he does not do so and picks another node, say $\mathbf{e}$, then in the next round the spoiler just picks the parent of $\mathbf{e}$ to which the duplicator has no answer.

Since c_i and c_j come from different components, the duplicator and the spoiler never play k rounds in the subgames $G_k^{\mathrm{MSO}}(\mathbf{t}_a; \mathbf{s}_a)$ and $G_k^{\mathrm{MSO}}(\mathbf{t}_b; \mathbf{s}_b)$. That is, in the subgames $G_k^{\mathrm{MSO}}(\mathbf{t}_a; \mathbf{s}_a)$ and $G_k^{\mathrm{MSO}}(\mathbf{t}_b; \mathbf{s}_b)$, the elements c_i and c_j are chosen before the k-th round. By the above argument, this means that d_i and d_j have to be the roots of $\mathbf{s}_a$ and $\mathbf{s}_b$, respectively. Therefore, $d_i < d_j$ as required.

Concerning E, we only have to consider the case where $c_i = \mathbf{n}$ and c_j is a child of $\mathbf{n}$. By a similar argument as before it follows that d_j has to be a child of $\mathbf{m}$. ∎

By the previous proposition, the $\equiv_k^{\mathrm{MSO}}$-type of a tree only depends on the $\equiv_k^{\mathrm{MSO}}$-types of the subtrees rooted at the children of the root. This suggests a mechanism to compute $\equiv_k^{\mathrm{MSO}}$-types of trees in a bottom-up way. Indeed, the set of states is Φ_k, and the transition function is defined as follows: for every $\sigma \in \Sigma$, $\delta(\sigma) = \tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma))$ and for $\theta_1, \ldots, \theta_n \in \Phi_k$, $\delta(\theta_1, \ldots, \theta_n, \sigma) = \tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n))$ whenever there exist trees $\mathbf{t}_1, \ldots, \mathbf{t}_n$ such that $\tau_k^{\mathrm{MSO}}(\mathbf{t}_1) = \theta_1, \ldots, \tau_k^{\mathrm{MSO}}(\mathbf{t}_n) = \theta_n$. By Proposition 2.12, it does not matter which members $\mathbf{t}_1, \ldots, \mathbf{t}_n$ of the $\equiv_k^{\mathrm{MSO}}$-equivalence classes $\theta_1, \ldots, \theta_n$ we take. A tree automaton, in turn, can again be defined in MSO by guessing states and then verifying in FO the consistency with the transition function. This leads to the following theorem obtained by Doner, Thatcher and Wright [Don70, TW68].

Theorem 2.13 *A tree language is recognizable if and only if it is definable in MSO.*

For later use, we show that a bottom-up tree automaton also can compute the type $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$ of each input tree $\mathbf{t}$. Therefore, we need the following proposition. Actually, we only need the second item of Proposition 2.14, the other items will be used in Section 3.2.

Proposition 2.14 *Let k be a natural number, $\mathbf{t}$ and $\mathbf{s}$ be two trees, $\mathbf{n}$ be a node of $\mathbf{t}$ and $\mathbf{m}$ be a node of $\mathbf{s}$ both of arity n.*

1. *If $(\overline{\mathbf{t}_\mathbf{n}}, \mathbf{n}) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_\mathbf{m}}, \mathbf{m})$ and $(\mathbf{t}_\mathbf{n}, \mathbf{n}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_\mathbf{m}, \mathbf{m})$ then $(\mathbf{t}, \mathbf{n}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}, \mathbf{m})$.*
2. *If $\mathrm{lab}_\mathbf{t}(\mathbf{n}) = \mathrm{lab}_\mathbf{s}(\mathbf{m})$ and $(\mathbf{t}_{\mathbf{n}i}, \mathbf{n}i) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_{\mathbf{m}i}, \mathbf{m}i)$ for $i = 1, \ldots, n$, then $(\mathbf{t}_\mathbf{n}, \mathbf{n}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_\mathbf{m}, \mathbf{m})$.*
3. *Let $i \in \{1, \ldots, n\}$. If*
   - $(\overline{\mathbf{t}_\mathbf{n}}, \mathbf{n}) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_\mathbf{m}}, \mathbf{m})$,
   - $\mathrm{lab}_\mathbf{t}(\mathbf{n}i) = \mathrm{lab}_\mathbf{t}(\mathbf{m}i)$, *and*
   - $(\mathbf{t}_{\mathbf{n}j}, \mathbf{n}j) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_{\mathbf{m}j}, \mathbf{m}j)$, *for* $j \in \{1, \ldots, n\} - \{i\}$,

   then $(\overline{\mathbf{t}_{\mathbf{n}i}}, \mathbf{n}i) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_{\mathbf{m}i}}, \mathbf{m}i)$.

Proof. The proofs of all three cases are very similar. The basic idea is to combine the winning strategies of the duplicator on the respective subtrees into a winning strategy on the whole structures like in the case of strings in Proposition 2.6. We focus on the third case where there are altogether $n + 1$ subgames including the trivial game in which one structure consists only of $\mathbf{n}i$ and the other of $\mathbf{m}i$. The winning strategy in the game on $(\overline{\mathbf{t}_{\mathbf{n}i}}, \mathbf{n}i)$ and $(\overline{\mathbf{s}_{\mathbf{m}i}}, \mathbf{m}i)$ just combines the winning strategies in those $n+1$ subgames. At the end of the game, the selected vertices define partial isomorphisms for all pairs of respective substructures. To ensure that they also define a partial isomorphism between the entire structures one only has to check the relations $<$ and E between the chosen elements, and $\mathbf{n}i$ and $\mathbf{m}i$. The preservation of $<$ and E between chosen elements only can be verified as in the proof of Proposition 2.12. Additionally, we have to check that for every corresponding pair of chosen nodes c and d: $c < \mathbf{n}i$ iff $d < \mathbf{m}i$, $\mathbf{n}i < c$ iff $\mathbf{m}i < d$, and $E(c, \mathbf{n}i)$ iff $E(d, \mathbf{m}i)$. This follows immediately, as all siblings and the parents of $\mathbf{n}i$ and $\mathbf{m}i$ are distinguished constants, and only elements of corresponding substructures are chosen. ■

We will use the following lemma in Section 5.2.

Lemma 2.15 *Let k be a natural number. There exists a DBTA $B = (Q, \Sigma, \delta, F)$ such that $\delta^*(\mathbf{t}) = \tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$, for every tree $\mathbf{t}$.*

Proof. We apply the same bottom-up technique as in the proof of Theorem 2.13. Only now we make use of Proposition 2.14(2) rather than Proposition 2.12. Define Q as Φ_k. Here we take Φ_k as the set of $\equiv_k^{\mathrm{MSO}}$-types of trees with one distinguished node. Further, define the transition function as follows: for every $\sigma \in \Sigma$, $\delta(\sigma) = \tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma), \mathrm{root}(\mathbf{t}(\sigma)))$ and for $\theta, \theta_1, \ldots, \theta_n \in \Phi_k$, $\delta(\theta_1, \ldots, \theta_n, \sigma) = \theta$ iff there exists a tree $\mathbf{t}$ with a node $\mathbf{n}$ of arity n such that $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}}, \mathbf{n}) = \theta$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}i}, \mathbf{n}i) = \theta_i$, for $i = 1, \ldots, n$. By Proposition 2.14(2), it does not matter which members $\mathbf{t}_{\mathbf{n}1}, \ldots,$ $\mathbf{t}_{\mathbf{n}n}$ of the $\equiv_k^{\mathrm{MSO}}$-equivalence classes $\theta_1, \ldots, \theta_n$ we take. ■

3

Expressiveness of structured document query languages based on attribute grammars

In this chapter, we focus on structured documents described by context-free grammars. As mentioned before, the context-free grammar models the *schema* of the database, while a *database instance* is simply a derivation tree of this grammar. The latter approach was originally proposed by Gonnet and Tompa [GT87]. In this respect, we study query languages based on the standard attribute grammar formalism as introduced by Knuth [Knu68].

Concretely, we study the expressiveness of Boolean-valued (BAGs) and relation-valued attribute grammars (RAGs). BAGs are an abstraction of the query facility provided by information retrieval systems and therefore express unary queries. RAGs, on the other hand, express relational queries and can be seen as abstractions of wrappers.

Specifically, we link BAGs with monadic second-order logic, and RAGs with first-order inductions of linear depth, or, equivalently, the queries computable in linear time on a parallel machine with polynomially processors. Further, we show that RAGs that only use synthesized attributes are strictly weaker than RAGs that use both synthesized and inherited attributes and obtain that RAGs are more expressive than monadic second-order logic for queries of any arity. Finally, we discuss relational attribute grammars in the context of BAGs and RAGs. Specifically, we show that in the case of BAGs this does not increase the expressive power, while different semantics for relational RAGs capture the complexity classes NP, coNP and UP $\cap$ coUP.

3.1 Attribute grammars as query languages

For all what follows in this chapter, we fix a *context-free grammar* $G = (N, T, P, U)$, where N is the set of non-terminals, T is the set of terminals, P is the set of productions, and U is the start symbol. We make the harmless technical assumption that the start symbol U does not appear on the right-hand side of any production. A *derivation tree of* G is defined in the standard way (see, e.g., [HU79]).

Let $\mathbf{t}$ be a derivation tree of G and let $\mathbf{n}_0, \mathbf{n}_1, \ldots, \mathbf{n}_n$ be nodes of $\mathbf{t}$ such that $\mathbf{n}_0$ has exactly the n children $\mathbf{n}_1, \ldots, \mathbf{n}_n$:

$$\begin{array}{ccc} & \mathbf{n}_0 & \\ \swarrow & \cdots & \searrow \\ \mathbf{n}_1 & & \mathbf{n}_n\,. \end{array}$$

Let $p = X_0 \to X_1 \ldots X_n$ be a production. If the label of $\mathbf{n}_0$ is X_0, and $\mathbf{n}_i$ is labeled by X_i for $i = 1, \ldots, n$, then we say that $\mathbf{n}_0$ is *derived* by p.

3.1.1 Attribute grammar formalism

We define the concepts common to both Boolean-valued and relation-valued attribute grammars.

Definition 3.1 An *attribute grammar vocabulary* has the form

$$(A, \mathrm{Syn}, \mathrm{Inh}, \mathrm{Att}),$$

where

- A is a finite set of symbols called *attributes*;
- Syn, Inh, and Att are functions from $N \cup T$ to the powerset of A such that for every $X \in N$, $\mathrm{Syn}(X) \cap \mathrm{Inh}(X) = \emptyset$; for every $X \in T$, $\mathrm{Syn}(X) = \emptyset$; and $\mathrm{Inh}(U) = \emptyset$.
- for every X, $\mathrm{Att}(X) = \mathrm{Syn}(X) \cup \mathrm{Inh}(X)$.

If $a \in \mathrm{Syn}(X)$, we say that a is a *synthesized attribute of* X. If $a \in \mathrm{Inh}(X)$, we say that a is an *inherited attribute of* X. The above conditions express that an attribute cannot be a synthesized and an inherited attribute of the same symbol, that terminal symbols do not have synthesized attributes, and that the start symbol does not have inherited attributes.

From now on we fix some attribute grammar vocabulary.

Definition 3.2 Let $p = X_0 \to X_1 \ldots X_n$ be a production in P, and a an attribute of X_i for some $i \in \{0, \ldots, n\}$. Then the triple (p, a, i) is called a *context* if $a \in \mathrm{Syn}(X_i)$ implies $i = 0$, and $a \in \mathrm{Inh}(X_i)$ implies $i > 0$.

$$
\begin{array}{ll}
U \to S & x\_before(1) := \text{false} \\
S \to BS & x\_before(2) := is\_x(1) \vee x\_before(0) \\
 & even(0) := \neg even(2) \\
 & result(0) := even(0) \wedge x\_before(0) \\
S \to B & even(0) := \text{false} \\
 & result(0) := \text{false} \\
B \to x & is\_x(0) := \text{true} \\
B \to y & is\_x(0) := \text{false}
\end{array}
$$

Figure 3.1: Example of a BAG.

3.1.2 Boolean-valued attribute grammars

Definition 3.3 A BAG-*rule in the context* (p, a, i), with $p = X_1 \dots X_n$, is an expression of the form

$$a(i) := \varphi,$$

where φ is a propositional logic formula over the set of proposition symbols

$$\{b(j) \mid j \in \{0, \dots, n\} \text{ and } b \in \mathrm{Att}(X_j)\}.$$

A BAG is then defined as follows:

Definition 3.4 A *Boolean-valued attribute grammar (BAG)* $\mathcal{B}$ consists of an attribute grammar vocabulary, together with a mapping assigning to each context a BAG-rule in that context.

Example 3.5 In Figure 3.1 a simple example of a grammar and a BAG over this grammar are depicted. We have $\mathrm{Syn}(S) = \{result, even\}$, $\mathrm{Inh}(S) = \{x\_before\}$, $\mathrm{Syn}(B) = \{is\_x\}$, and $\mathrm{Att}(U) = \mathrm{Att}(x) = \mathrm{Att}(y) = \mathrm{Inh}(B) = \emptyset$. The semantics of this BAG will be explained below. ∎

The semantics of a BAG is that it defines Boolean attributes of the nodes of derivation trees of the underlying grammar G. This is formalized next.

Definition 3.6 Let $\mathbf{t}$ be a derivation tree of G. A *valuation of* $\mathbf{t}$ is a function that maps pairs $(\mathbf{n}, a)$, where $\mathbf{n}$ is a node in $\mathbf{t}$ and a is an attribute of the label of $\mathbf{n}$, to truth values (0 or 1).

In the sequel, for a pair $(\mathbf{n}, a)$ as above we will use the more intuitive notation $a(\mathbf{n})$.

Definition 3.7 Let $\mathcal{B}$ be a BAG, and let $\mathbf{t}$ be a derivation tree. Let $a(i) := \varphi$ be the BAG-rule in context (p, a, i). Let $\mathbf{n}$ be a node of arity n derived by p. Then the formula obtained from φ by replacing each occurrence of a propositional symbol of the form $b(j)$ by the new propositional symbol $b(\mathbf{n}j)$, is denoted by $\Delta(\mathcal{B}, \mathbf{t}, a, \mathbf{n}i)$.

Definition 3.8 Let $\mathcal{B}$ be a BAG and $\mathbf{t}$ a derivation tree. We define a sequence $(\mathcal{B}_l(\mathbf{t}))_{l\geq 0}$ of partial valuations as follows:

- $\mathcal{B}_0(\mathbf{t})$ is the empty valuation ($\mathcal{B}_0(\mathbf{t})$ is nowhere defined).
- $\mathcal{B}_l(\mathbf{t})$, for $l > 0$, is defined as the following extension of $\mathcal{B}_{l-1}(\mathbf{t})$. For every $a(\mathbf{n})$, if $\mathcal{B}_{l-1}(\mathbf{t})$ is defined on all propositional symbols that occur in $\Delta(\mathcal{B}, \mathbf{t}, a, \mathbf{n})$, then $\mathcal{B}_l(\mathbf{t})$ is defined on $a(\mathbf{n})$ and gets the truth value taken by $\Delta(\mathcal{B}, \mathbf{t}, a, \mathbf{n})$ under the valuation $\mathcal{B}_{l-1}(\mathbf{t})$.

If for every $\mathbf{t}$ there is an l such that $\mathcal{B}_l(\mathbf{t})$ is a totally defined valuation of $\mathbf{t}$ (this implies that $\mathcal{B}_{l+1} = \mathcal{B}_l$), then we say that $\mathcal{B}$ is *non-circular*. From now on, we will only consider BAGs that are non-circular. (Non-circularity is well known to be decidable [Knu68].) The valuation $\mathcal{B}(\mathbf{t})$ is then defined as $\mathcal{B}_l(\mathbf{t})$.

It is well known that the evaluation of an attribute grammar takes linear time when counting the evaluation of a semantic rule as one unit of time (see, e.g., [DJL88]). This is simply because only a constant number of attributes should be defined for every node. Since a fixed propositional formula can indeed be evaluated in constant time, the valuation $\mathcal{B}(\mathbf{t})$ of a BAG $\mathcal{B}$ on a tree $\mathbf{t}$ can thus be computed in time linear in the size of $\mathbf{t}$.

An arbitrary total valuation v of $\mathbf{t}$ is said to *satisfy* $\mathcal{B}$ if $v(a(\mathbf{n}))$ equals the truth value taken by $\Delta(\mathcal{B}, \mathbf{t}, a, \mathbf{n})$ under v, for each attribute a and node $\mathbf{n}$ of $\mathbf{t}$ such that a is an attribute of the label of $\mathbf{n}$.

We shall make use of the following lemma:

Lemma 3.9 *For each BAG $\mathcal{B}$ and tree $\mathbf{t}$, $\mathcal{B}(\mathbf{t})$ is the only valuation that satisfies $\mathcal{B}$.*

Proof. It follows immediately from the definitions that $\mathcal{B}(\mathbf{t})$ satisfies $\mathcal{B}$.

Suppose that v satisfies $\mathcal{B}$. We now show by induction on l that if $a(\mathbf{n})$ is defined in $\mathcal{B}_l(\mathbf{t})$ then $\mathcal{B}_l(\mathbf{t})(a(\mathbf{n})) = v(a(\mathbf{n}))$. This clearly holds for $l = 0$. Suppose $l > 0$ and $a(\mathbf{n})$ is defined in $\mathcal{B}_l(\mathbf{t})$. If $a(\mathbf{n})$ is already defined in $\mathcal{B}_{l-1}(\mathbf{t})$ then the claim holds by the inductive hypothesis. If $a(\mathbf{n})$ is not defined in $\mathcal{B}_{l-1}(\mathbf{t})$, then, by definition, the value $\mathcal{B}_l(\mathbf{t})(a(\mathbf{n}))$ equals the truth value of $\Delta(\mathcal{B}, \mathbf{t}, a, \mathbf{n})$ under the valuation $\mathcal{B}_{l-1}(\mathbf{t})$. By assumption $v(a(\mathbf{n}))$ equals the truth value of $\Delta(\mathcal{B}, \mathbf{t}, a, \mathbf{n})$ under the valuation v. By the inductive hypothesis we have that $\mathcal{B}_{l-1}(\mathbf{t})(b(\mathbf{m})) = v(b(\mathbf{m}))$, for all $b(\mathbf{m})$ that are defined in $\mathcal{B}_{l-1}(\mathbf{t})$. Hence, $\mathcal{B}_j(\mathbf{t})(a(\mathbf{n})) = v(a(\mathbf{n}))$.

By definition of $\mathcal{B}(\mathbf{t})$ the lemma now holds. ■

A BAG $\mathcal{B}$ can be used in a simple way to express unary (i.e., 1-ary) queries. Among the attributes in the vocabulary of $\mathcal{B}$, we designate some attribute *result*, and define:

Definition 3.10 A BAG $\mathcal{B}$ expresses the unary query $\mathcal{Q}$ defined by

$$\mathcal{Q}(\mathbf{t}) = \{\mathbf{n} \mid \mathcal{B}(\mathbf{t})(\mathit{result}(\mathbf{n})) = 1\},$$

for every tree $\mathbf{t}$.

$$
\begin{array}{ccccccccccccccc}
 & & \begin{array}{c} \mathit{even} = 0 \\ \mathit{x\_before} = 0 \\ \mathit{result} = 0 \end{array} & & \begin{array}{c} \mathit{even} = 1 \\ \mathit{x\_before} = 0 \\ \mathit{result} = 0 \end{array} & & \begin{array}{c} \mathit{even} = 0 \\ \mathit{x\_before} = 0 \\ \mathit{result} = 0 \end{array} & & \begin{array}{c} \mathit{even} = 1 \\ \mathit{x\_before} = 1 \\ \mathit{result} = 1 \end{array} & & \begin{array}{c} \mathit{even} = 0 \\ \mathit{x\_before} = 1 \\ \mathit{result} = 0 \end{array} & \\
U & \longrightarrow & S & \longrightarrow & S & \longrightarrow & S & \longrightarrow & S & \longrightarrow & S & \\
 & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
 & & B & \mathit{is\_x} = 0 & B & \mathit{is\_x} = 0 & B & \mathit{is\_x} = 1 & B & \mathit{is\_x} = 0 & B & \mathit{is\_x} = 1 \\
 & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
 & & y & & y & & x & & y & & x &
\end{array}
$$

Figure 3.2: A derivation tree and its valuation defined by the BAG of Figure 3.1.

Example 3.11 Recall the BAG of Figure 3.1. A derivation tree of the underlying grammar can be viewed naturally as a string over the alphabet $\{x, y\}$. Every node labeled S in the tree represents a position in the string. Now consider the semantic rules defining the synthesized attribute *even*. They can be evaluated bottom-up; for any node $\mathbf{n}$, $\mathit{even}(\mathbf{n})$ is true iff $\mathbf{n}$ is even-numbered when counting up from the bottom. The semantic rules defining the inherited attribute *x_before* can be evaluated top-down; $\mathit{x\_before}(\mathbf{n})$ is true iff the letter x occurs in the string somewhere before position $\mathbf{n}$. Finally, the semantic rules for the attribute *result* simply define $\mathit{result}(\mathbf{n})$ as $\mathit{x\_before}(\mathbf{n}) \wedge \mathit{even}(\mathbf{n})$. Hence, the BAG expresses the query retrieving those even-numbered positions that come after an x in the string. An illustration is given in Figure 3.2. ■

Example 3.12 A BAG can also be used to query content rather than just structure. The context-free grammar in Figure 3.3 models a list of authors where each author is a sequence of letters. Suppose we want to select all authors named John. For the non-terminal Letter, we use the synthesized attributes is-a, ..., is-z. These attributes indicate the symbol that expands this non-terminal: for each production Letter $\rightarrow x$ and each attribute is-y, the semantic rule for is-y is defined as true if $x = y$ and as false otherwise. Further, for the non-terminal LetterList, we have the attributes is-john, is-ohn, is-hn, and is-n. Their meaning is the obvious one. For instance, is-hn is true for a LetterList-labeled node when its first child is expanded with h and when the first child of its second child is expanded with n. These attributes are defined as follows:

LetterList $\rightarrow$ Letter LetterList	is-john(0) := is-j(1) $\wedge$ is-ohn(2)
	is-ohn(0) := is-o(1) $\wedge$ is-hn(2)
	is-hn(0) := is-h(1) $\wedge$ is-n(2)
	is-n(0) := is-n(1)
LetterList $\rightarrow$ Letter	is-john(0) := false
	is-ohn(0) := false
	is-hn(0) := false
	is-n(0) := is-n(1).

$$
\begin{array}{l}
\text{AuthorList} \to \text{Author AuthorList} \\
\text{AuthorList} \to \text{Author} \\
\text{Author} \to \text{LetterList} \\
\text{LetterList} \to \text{Letter LetterList} \\
\text{LetterList} \to \text{Letter} \\
\text{Letter} \to \text{a} \\
\qquad\vdots \\
\text{Letter} \to \text{z}
\end{array}
$$

Figure 3.3: A context-free grammar modeling a list of authors.

Finally, we define the *result* attribute:

$$\text{Author} \to \text{LetterList} \quad \mathit{result}(0) := \text{is-john}(1).$$

From this simple example it should be clear how text searching can be done for derivation trees of more involved grammars. ■

A BAG can also be used to express Boolean (i.e., nullary) queries. Among the attributes of the start symbol, we designate some attribute *result*, and define:

Definition 3.13 A BAG $\mathcal{B}$ expresses the Boolean query $\mathcal{Q}$ defined by

$$\mathcal{Q}(\mathbf{t}) = \begin{cases} \text{true} & \text{if } \mathcal{B}(\mathbf{t})(\mathit{result}(\mathbf{r})) = 1; \\ \text{false} & \text{otherwise,} \end{cases}$$

for every tree $\mathbf{t}$. Here $\mathbf{r}$ denotes the root of $\mathbf{t}$.

3.1.3 Relation-valued attribute grammars

In this section, we generalize BAGs to *relation*-valued attribute grammars (RAGs). We start by giving an example.

Example 3.14 As a first example, consider the RAG shown in Figure 3.4. A derivation tree of the underlying grammar models a set (S) of documents (D). Each document is a list (L) of paragraphs (p). The synthesized attribute *result* of U and S is relation-valued; on any tree, the value of *result* at the root will be the ternary relation consisting of all triples (d, f, l) such that, intuitively, d is a document, f is the first paragraph of d, and l is the last paragraph of d. More precisely, d, f, and l are not actual parts of the derivation tree, but are just nodes corresponding to documents and paragraphs. The *result* relation is computed using the synthesized attributes *first* and *last* of D and L; for every document node $\mathbf{n}$, *first*$(\mathbf{n})$ contains the first paragraph of that document, and *last*$(\mathbf{n})$ contains the last. These attributes are computed in turn using the inherited attribute *begin* and the synthesized attribute *end* of L, which are Boolean-valued; for any L-node $\mathbf{n}$, *begin*$(\mathbf{n})$ is true if $\mathbf{n}$ marks the beginning of a

$$\begin{array}{ll}
U \to S & \mathit{result}(0) := \mathit{result}(1) \\
S \to DS & \mathit{result}(0) := (\{(\mathbf{1})\} \times \mathit{first}(1) \times \mathit{last}(1)) \cup \mathit{result}(2) \\
S \to & \mathit{result}(0) := \emptyset \\
D \to L & \mathit{first}(0) := \mathit{first}(1) \\
 & \mathit{last}(0) := \mathit{last}(1) \\
 & \mathit{begin}(1) := \mathrm{true} \\
L \to pL & \mathit{first}(0) := \mathbf{if}\ \mathit{begin}(0)\ \mathbf{then}\ \{(\mathbf{1})\}\ \mathbf{else}\ \emptyset \\
 & \mathit{last}(0) := \mathbf{if}\ \mathit{end}(2)\ \mathbf{then}\ \{(\mathbf{1})\}\ \mathbf{else}\ \mathit{last}(2) \\
 & \mathit{begin}(2) := \mathrm{false} \\
 & \mathit{end}(0) := \mathrm{false} \\
L \to & \mathit{end}(0) := \mathrm{true} \\
 & \mathit{first}(0) := \emptyset \\
 & \mathit{last}(0) := \emptyset
\end{array}$$

Figure 3.4: Example of a RAG.

document, and $\mathit{end}(\mathbf{n})$ is true if $\mathbf{n}$ marks the end. Note that we now use first-order expressions, rather than propositional ones, to define the values of the attributes. ■

Let us indicate the differences between BAGs and RAGs more formally.

Definition 3.15 To each attribute a we associate an arity r_a (a natural number). A RAG-rule in the context (p, a, i), with $p = X_0 \to X_1 \dots X_n$ is an expression of the form

$$a(i) := \varphi(x_1, \dots, x_{r_a}),$$

where φ is a first-order logic formula over the vocabulary

$$\bigcup_{j=0}^{n} \{b(j) \mid b \in \mathrm{Att}(X_j)\} \cup \{\mathbf{0}, \mathbf{1}, \dots, \mathbf{n}\},$$

where for each $j = 0, \dots, n$, $b(j)$ is a relation symbol of arity r_b, and $\mathbf{j}$ is a constant symbol. A valuation of a derivation tree $\mathbf{t}$ is a function that maps each pair $(\mathbf{n}, a)$, where $\mathbf{n}$ is a node labeled X and a is an attribute of X, to an r_a-ary relation over the nodes of $\mathbf{t}$. A RAG $\mathcal{R}$ consists of an attribute grammar vocabulary together with a mapping assigning to each context a RAG-rule in that context.

Definition 3.16 Let $\mathcal{R}$ be a RAG, and let $\mathbf{t}$ be a derivation tree. Let $a(i) := \varphi$ be the RAG-rule in the context (p, a, i). Let $\mathbf{n}$ be a node of arity n derived by p. Then the formula obtained from φ by replacing each occurrence of a relation symbol $b(j)$ by the relation symbol $b(\mathbf{n}j)$, and by replacing each constant symbol $\mathbf{j}$ by the node $\mathbf{n}j$, is denoted by $\Delta(\mathcal{R}, \mathbf{t}, a, \mathbf{n}i)$.

Definition 3.17 Let $\mathcal{R}$ be a RAG and $\mathbf{t}$ a derivation tree. We define a sequence $(\mathcal{R}_l(\mathbf{t}))_{l \geq 0}$ of partial valuations as follows:

$$\begin{array}{lll}
U \to N & \mathit{result}(0) := & \mathit{order}(1) \cup \{(\mathbf{1},\mathbf{0})\} \cup \{(\mathbf{0},\mathbf{0})\} \\
& & \cup\, \mathit{descendants}(1) \times \{(\mathbf{0})\} \\
N \to NN & \mathit{descendants}(0) := & \{(\mathbf{0})\} \cup \mathit{descendants}(1) \cup \mathit{descendants}(2) \\
& \mathit{order}(0) := & \mathit{descendants}(0) \times \{(0)\} \\
& & \cup\, (\mathit{descendants}(1) \times \mathit{descendants}(2)) \\
& & \cup\, \mathit{order}(1) \cup \mathit{order}(2) \\
N \to x & \mathit{descendants}(0) := & \{(\mathbf{0}), (\mathbf{1})\} \\
& \mathit{order}(0) := & \{(\mathbf{1},\mathbf{0}), (\mathbf{0},\mathbf{0}), (\mathbf{1},\mathbf{1})\}
\end{array}$$

Figure 3.5: Computing a linear order on the nodes using a RAG.

- $\mathcal{R}_0(\mathbf{t})$ is the empty valuation ($\mathcal{R}_0(\mathbf{t})$ is nowhere defined).
- $\mathcal{R}_l(\mathbf{t})$, for $l > 0$, is defined as the following extension of $\mathcal{R}_{l-1}(\mathbf{t})$. For every $a(\mathbf{n})$, if $\mathcal{R}_{l-1}$ is defined on all relational symbols that occur in $\Delta(\mathcal{R}, \mathbf{t}, a, \mathbf{n})$, then $\mathcal{R}_l(\mathbf{t})$ is defined on $a(\mathbf{n})$ as the relation obtained by evaluating the FO-formula $\Delta(\mathcal{R}, \mathbf{t}, a, \mathbf{n})$ over the whole tree where each relation symbol $b(\mathbf{m})$ in $\Delta(\mathcal{R}, \mathbf{t}, a, \mathbf{n})$ is interpreted by $\mathcal{R}_{l-1}(b(\mathbf{m}))$.

The valuation $\mathcal{R}(\mathbf{t})$ is then defined as $\mathcal{R}_l(\mathbf{t})$, where l is such that $\mathcal{R}_l$ is a total valuation.

An arbitrary total valuation v of $\mathbf{t}$ is said to *satisfy* $\mathcal{R}$ if $v(a(\mathbf{n}))$ equals the relation defined by the FO-formula $\Delta(\mathcal{R}, \mathbf{t}, a, \mathbf{n})$, where each relation symbol $b(\mathbf{n}_j)$ is interpreted by $v(b(\mathbf{n}_j))$. Analogous to Lemma 3.9, one can prove the following lemma:

Lemma 3.18 *For each RAG $\mathcal{R}$ and tree $\mathbf{t}$, $\mathcal{R}(\mathbf{t})$ is the only valuation that satisfies $\mathcal{R}$.*

A RAG can be used to express k-ary queries in a simple way. Among the attributes of the start symbol U we designate some k-ary attribute *result*, and define:

Definition 3.19 A RAG $\mathcal{R}$ expresses the query $\mathcal{Q}$ defined as follows: for any tree $\mathbf{t}$, $\mathcal{Q}(\mathbf{t})$ equals the value of *result*(root($\mathbf{t}$)) in $\mathcal{R}(\mathbf{t})$.

Example 3.20 Another example of a RAG is depicted in Figure 3.5. The binary (i.e., 2-ary) query expressed by the RAG results on each tree in a linear order on its nodes, corresponding to a postorder traversal [Knu82] of the tree. This example can easily be generalized to arbitrary grammars. ■

As mentioned in the introduction RAGs can be seen as an abstract model for wrappers. These are tools that map relevant parts of the document at hand into, for instance, a relational database [ACM98, MAM$^+$98, PGMW95]. We give an example to illustrate this.

Example 3.21 The grammar in Figure 3.6 models a list of publications. Each publication consists of a list of authors and a title. We now want a wrapper generating a binary relation consisting of all pairs (a, t) such that a is the author of a publication

PubList → Pub PubList	$result(0) := result(1) \cup result(2)$
PubList → Pub	$result(0) := result(1)$
Pub → AuthorList Title	$result(0) := b(1) \times \{\mathbf{2}\}$
AuthorList → Author AuthorList	$b(0) := \{\mathbf{1}\} \cup b(2)$
AuthorList → Author	$b(0) := \{\mathbf{1}\}$

Figure 3.6: RAGs as an abstraction of wrappers.

with title t. The RAG in Figure 3.6 expresses this transformation. Here, for every AuthorList node $\mathbf{n}$, $b(\mathbf{n})$ contains the set of authors in the author list associated to $\mathbf{n}$. Further, for every Pub node $\mathbf{n}$, $result(\mathbf{n})$ contains all pairs (a, t) where a is an author and t is the title of the publication represented by $\mathbf{n}$.

Of course, the binary relation created by a real wrapper, as opposed to the abstraction of it by RAGs, would contain the actual string content (that is, actual names of authors and titles) rather than just the nodes in the document corresponding to them. ■

3.2 Expressive power of BAGs

In this section we characterize the expressive power of BAGs in terms MSO. As a corollary we obtain a bottom-up property for Boolean BAG queries.

A derivation tree of G can be seen as a Σ-tree with $\Sigma = N \cup T$. We therefore use for such derivation trees the vocabulary associated to $(N \cup T)$-trees as defined in Section 2.4. Further, let m be the maximum number of symbols occurring on the right hand side of a production of G. All derivation trees then are of rank m and we will make use of the shorthand $S_i(x, y)$ indicating that y is the i-th child of x. Clearly, $S_i(x, y)$ is FO definable from $<$ and E for each i.

Proviso 3.22 In the following, unless explicitly specified otherwise, if we say 'tree' we always mean 'derivation tree of G'.

3.2.1 Main Theorem

We first show that every query expressible by a BAG is also definable in MSO.

Lemma 3.23 *Every unary query expressible by a BAG is definable in MSO.*

Proof. Let $\mathcal{B}$ be a BAG. We know from Lemma 3.9 that for each tree there exists only one valuation that satisfies $\mathcal{B}$. In MSO we can easily define this valuation. For each attribute a we have a set variable Z_a. This variable will contain all the nodes for which the attribute a is true in $\mathcal{B}(\mathbf{t})$. To this end, we associate a formula to each semantic rule in the following way. Consider a rule $a(i) := \varphi$ of $\mathcal{B}$ in the context

(p, a, i) for some production $p = X_0 \to X_1 \dots X_n$ of G. Define the formula $\rho_{p,a,i}(z_a)$ as

$$\rho_{p,a,i}(z_a, (Z_b)_{b\in A}) :=$$
$$(\exists z_0)(\exists z_1)\dots(\exists z_n)\left(\underbrace{\bigwedge_{j=0}^{n} O_{X_j}(z_j) \wedge \bigwedge_{j=1}^{n} S_j(z_0, z_j)}_{(*)} \wedge z_a = z_i \ \wedge \widehat{\varphi}\right),$$

where $\widehat{\varphi}$ is obtained from φ by replacing each propositional symbol $b(j)$ occurring in φ by $Z_b(z_j)$. Intuitively, formula $(*)$ states that

$$\begin{array}{ccc} & z_0 & \\ \swarrow & \cdots & \searrow \\ z_1 & & z_n \end{array}$$

is derived by the production p. Formula $\hat{\varphi}$ states that φ holds for $z_0, z_1, \dots, z_n$, i.e., that $a(z_i)$ is true. We now define φ_a as the following disjunction over all rules defining the attribute a:

$$\varphi_a(z_a, (Z_b)_{b\in A}) := \bigvee\{\rho_{p,a,i}(z_a, (Z_b)_{b\in A}) \mid (p, a, i) \text{ is a context}\}.$$

Define $\xi((Z_a)_{a\in A})$ as the formula

$$\bigwedge_{a\in A} (\forall z)(Z_a(z) \leftrightarrow \varphi_a(z, (Z_b)_{b\in A})).$$

Let $\mathbf{t}$ be a tree and for each $a \in A$ let $\mathbf{s}_a$ be a set of nodes of $\mathbf{t}$ such that $\mathbf{t} \models \xi[(\mathbf{s}_a)_{a\in A}]$. Then define the valuation v as follows:

$$v(a(\mathbf{n})) := \begin{cases} 1 & \text{if } \mathbf{n} \in \mathbf{s}_a, \\ 0 & \text{otherwise.} \end{cases}$$

If follows from the definition of ξ that v satisfies $\mathcal{B}$. Since, according to Lemma 3.9, there exists only one valuation that satisfies $\mathcal{B}$, it follows that for each $\mathbf{t}$ there exists only one sequence of sets $(\mathbf{s}_a)_{a\in A}$ such that $\mathbf{t} \models \xi[(\mathbf{s}_a)_{a\in A}]$. Hence, the following formula defines the query expressed by $\mathcal{B}$:

$$\sigma(z) := (\exists Z_a)_{a\in A} \left(\xi((Z_a)_{a\in A}) \wedge Z_{result}(z)\right).$$

■

The heart of the proof of the other direction consists of showing that a BAG can compute MSO-equivalence types. To this end we, make use of Proposition 2.14(1) which says that it suffices to compute $\tau_k^{\text{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$ and $\tau_k^{\text{MSO}}(\mathbf{t_n}, \mathbf{n})$, where k is the quantifier depth of φ, to decide whether $\mathbf{t} \models \varphi[\mathbf{n}]$. Furthermore, it follows from Proposition 2.14(2,3) that the types $\tau_k^{\text{MSO}}(\mathbf{t_n}, \mathbf{n})$ can be computed in a bottom-up fashion for each node $\mathbf{n}$, while the types $\tau_k^{\text{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$ can be computed in a top-down fashion when the $\equiv_k^{\text{MSO}}$-types of the subtrees rooted at the children of $\mathbf{n}$ are already known. We use these ideas in the proof of our first main result.

Theorem 3.24 *A unary query is expressible by a BAG if and only if it is definable in MSO.*

Proof. The only-if direction is given by Lemma 3.23.

For the other direction, consider the unary query defined by the MSO-formula $\varphi(z)$ with one free individual variable z. Furthermore, suppose the quantifier depth of φ is at most k.

We construct a BAG $\mathcal{B}$ with attribute set $A = \{env_\theta, sub_\theta \mid \theta \in \Phi_k\} \cup \{result\}$, where each env_θ is inherited for all grammar symbols except for the start symbol for which it is synthesized, and each sub_θ and *result* are synthesized for all non-terminals and inherited for all terminals. The intended meaning is the following: for each tree $\mathbf{t}$, each node $\mathbf{n}$ of $\mathbf{t}$, and each $\theta \in \Phi_k$,

- $\mathcal{B}(\mathbf{t})(env_\theta(\mathbf{n}))$ is true iff $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \theta$;
- $\mathcal{B}(\mathbf{t})(sub_\theta(\mathbf{n}))$ is true iff $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n}) = \theta$; and
- $\mathcal{B}(\mathbf{t})(result(\mathbf{n}))$ is true iff $\mathbf{t} \models \varphi[\mathbf{n}]$.

By Proposition 2.14(1), $\mathbf{t} \models \varphi[\mathbf{n}]$ only depends on $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n})$. Hence, $\mathcal{B}(\mathbf{t})(result(\mathbf{n}))$ only depends on the values

$$(\mathcal{B}(\mathbf{t})(env_\theta(\mathbf{n})))_{\theta\in\Phi_k} \quad \text{and} \quad (\mathcal{B}(\mathbf{t})(sub_\theta(\mathbf{n})))_{\theta\in\Phi_k}.$$

The BAG $\mathcal{B}$ works in two passes. In the first bottom-up pass all the sub_θ attributes are computed, while in the subsequent top-down pass all the env_θ attributes are computed. To initiate the top-down pass we use our convention, mentioned at the beginning of Section 3.1, that the start symbol cannot appear in the left-hand side of a production.[1] During this second pass, there is enough information at each node $\mathbf{n}$ to decide whether $\mathbf{t} \models \varphi[\mathbf{n}]$.

We now define the semantic rules of $\mathcal{B}$. To this end, we introduce the following functions. For convenience, we sometimes write $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root})$ for $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$. Define for each grammar symbol X the function $\xi_X : \Phi_k^* \mapsto \Phi_k$ mapping each sequence $\theta_1 \cdots \theta_n$ to θ if there are trees $\mathbf{t}_1, \dots, \mathbf{t}_n$, with $\tau_k^{\mathrm{MSO}}(X(\mathbf{t}_1, \dots, \mathbf{t}_n), \mathrm{root}) = \theta$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_i, \mathrm{root}) = \theta_i$, for $i = 1, \dots, n$. Further, define for each natural number i and grammar symbol X the function $\chi_X^i : \Phi_k \times \Phi_k^* \mapsto \Phi_k$ mapping each pair $(\theta_0, \theta_1 \cdots \theta_n)$, with $n \geq i$, to θ if there exists a tree $\mathbf{t}$ with a node $\mathbf{n}$ with n children such that

- $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}i) = X$;
- $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \theta_0$;
- for each $j \in \{1, \dots, n\} - \{i\}$, $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}j}, \mathbf{n}j) = \theta_j$; and

[1] This technicality can be dispensed with by adding so-called root rules to the attribute grammar formalism (see, e.g., Giegerich for a definition of attribute grammars with root rules [Gie88]). The formalism of attribute grammars becomes much less elegant then, however. Hence our harmless technical assumption concerning the start symbol.

- $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}i}}, \mathbf{n}i) = \theta$.

Note that the above definition does not depend on θ_i. By Proposition 2.14(2,3) the above functions are well defined. To each production $p = X_0 \to X_1 \ldots X_n$ we assign the following semantic rules (as usual the empty disjunction is false):

- For each $\theta \in \Phi_k$ and for every j such that X_j is a terminal, add the rule

$$sub_\theta(j) \ := \ \begin{cases} \text{true} & \text{if } \tau_k^{\mathrm{MSO}}(\mathbf{t}(X_j), \mathrm{root}) = \theta, \\ \text{false} & \text{otherwise.} \end{cases}$$

  Further, add for each $\theta \in \Phi_k$ the rule

$$sub_\theta(0) \ := \ \bigvee\{\bigwedge_{i=1}^{n} sub_{\theta_i}(i) \mid \theta_1, \ldots, \theta_n \in \Phi_k \text{ and } \xi_{X_0}(\theta_1 \cdots \theta_n) = \theta\}.$$

- If X_0 is the start symbol U, then for each $\theta \in \Phi_k$, add the semantic rule

$$env_\theta(0) \ := \ \begin{cases} \text{true} & \text{if } \tau_k^{\mathrm{MSO}}(\mathbf{t}(U), \mathrm{root}) = \theta, \\ \text{false} & \text{otherwise.} \end{cases}$$

  From our assumption that the start symbol U does not occur on the right-hand side of any production, we know that there is only one occurrence of U in the tree and this is at the root. So the second, top-down, pass will start at the root. For $j = 1, \ldots, n$, and $\theta \in \Phi_k$, add the rule

$$\begin{aligned} env_\theta(j) \ := \ & \bigvee\{env_{\theta_0} \wedge \bigwedge_{i=1}^{n} sub_{\theta_i}(i) \mid \\ & \qquad \theta_0, \theta_1, \ldots, \theta_n \in \Phi_k \text{ and } \chi^j_{X_j}(\theta_0, \theta_1 \cdots \theta_n) = \theta\}. \end{aligned}$$

- Finally, for $j = 0, \ldots, n$, add the rule

$$\begin{aligned} \mathit{result}(j) \ := \ & \bigvee\{env_{\theta_e}(j) \wedge sub_{\theta_s}(j) \mid \theta_e, \theta_s \in \Phi_k \text{ and} \\ & \text{there exists a tree } \mathbf{t} \text{ with a node } \mathbf{n} \text{ such that } \tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n}) = \theta_e, \\ & \tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}}, \mathbf{n}) = \theta_s, \text{ and } \mathbf{t} \models \varphi[\mathbf{n}]\}. \end{aligned}$$ ■

As a corollary of our proof of Theorem 3.24 we obtain a normal form for BAGs. The BAG described in the proof is special in two ways. First, it needs only positive formulas (involving only the connectives $\vee$ and $\wedge$, without $\neg$) in its semantic rules. Second, it can be evaluated on any tree by one bottom-up pass followed by one top-down pass. So we have the following:

$$\begin{array}{ll}
U \to S & \mathit{x_before}(1) := \text{false} \\
S \to BS & \mathit{x_before}(2) := \mathit{is_x}(1) \vee \mathit{x_before}(0) \\
 & \mathit{even}(0) := \mathit{odd}(2) \\
 & \mathit{odd}(0) := \mathit{even}(2) \\
 & \mathit{result}(0) := \mathit{even}(0) \wedge \mathit{x_before}(0) \\
S \to B & \mathit{even}(0) := \text{false} \\
 & \mathit{odd}(0) := \text{true} \\
 & \mathit{result}(0) := \text{false} \\
B \to x & \mathit{is_x}(0) := \text{true} \\
B \to y & \mathit{is_x}(0) := \text{false}
\end{array}$$

Figure 3.7: Example of a BAG without negation.

Corollary 3.25 *Every BAG is equivalent to one which uses only positive formulas in its semantic rules, and moreover which can be evaluated in two passes (more precisely, which is simply-2-pass* [DJL88]*).*

Actually, part of the above corollary, that one can always find an equivalent BAG which uses only positive rules, can also quite easily be seen directly. Let $\mathcal{B}$ be BAG over the attribute grammar vocabulary $(A, \text{Syn}, \text{Inh}, \text{Att})$. We construct an equivalent BAG $\mathcal{B}'$ over the attribute grammar vocabulary $(A', \text{Syn}', \text{Inh}', \text{Att}')$ that does not use negation in its semantic rules in the following way. For each attribute a we add an attribute Na that becomes true if the attribute a is false. Formally, $A' = A \cup \{Na \mid a \in A\}$, for each grammar symbol X,

$$\text{Syn}'(X) = \text{Syn}(X) \cup \{Na \mid a \in \text{Syn}(X)\},$$

and

$$\text{Inh}'(X) = \text{Inh}(X) \cup \{Na \mid a \in \text{Inh}(X)\}.$$

For each rule $a(i) := \varphi$ of $\mathcal{B}$ in context (p, a, i), add the rule $a(i) := \tilde{\varphi}$ in context (p, a, i) and the rule $Na(i) := \widetilde{\neg\varphi}$ in context (p, Na, i) to $\mathcal{B}'$. The formula $\tilde{\psi}$, where $\psi = \varphi$ or $\psi = \neg\varphi$, is obtained from ψ by transforming it into disjunctive normal form and then replacing each literal $\neg b(j)$ by $Nb(j)$.

Example 3.26 The BAG in Example 3.5 uses negation to select all nodes on an even numbered position. In Figure 3.7 a BAG is depicted that retrieves those nodes without using negation. For clarity we replaced the attribute *Neven* by *odd*. ■

3.2.2 Bottom-up property for Boolean BAG queries

Another view of a BAG is that of a two-way version of finite bottom-up tree automata, alternative to the more classical two-way generalization of tree automata provided by Moriya [Mor94]. The two-way generalization is provided by the two different types of attributes in a BAG: intuitively, synthesized attributes provide the bottom-up direction, and inherited attributes provide the top-down direction.

The following proposition relates BAGs and tree automata.

Proposition 3.27 *For each deterministic bottom-up tree automaton B there exists a BAG $\mathcal{B}$ such that for every derivation tree $\mathbf{t}$, B accepts $\mathbf{t}$ if and only if $\mathcal{B}$ accepts $\mathbf{t}$. This BAG uses only synthesized attributes.*

Proof. The execution of $B = (Q, N \cup T, F, \delta)$ on $\mathbf{t}$ can easily be simulated by a BAG $\mathcal{B}$ having synthesized attributes q for all states q in Q. If $\mathbf{n}$ is a node of $\mathbf{t}$, then the attribute value $q(\mathbf{n})$ is true in $\mathcal{B}(\mathbf{t})$ iff $\delta^*(\mathbf{t_n}) = q$, that is, B assumes state q at $\mathbf{n}$ in its execution on $\mathbf{t}$. Let $p = X_0 \to X_1 \ldots X_n$ be a production of G, $T(p) := \{j \in \{1, \ldots, n\} \mid X_j \text{ is a terminal}\}$, and $N(p) := \{1, \ldots, n\} - T(p)$. Add for each $q \in Q$ the semantic rule

$$q(0) := \bigvee\{ \bigwedge_{i \in N(p)} q_i(i) \mid q_1, \ldots, q_n \in Q, \delta(q_1, \ldots, q_n, X_0) = q,$$
$$\text{and for each } i \in T(p),\ \delta(X_i) = q_i \ \}.$$

Finally, the attribute *result* of U is defined by the rule $\mathit{result}(0) := \bigvee_{q \in F} q(0)$. ■

It now follows from Theorem 2.13 and Proposition 3.27, that every MSO-definable Boolean query is expressible by a synthesized BAG. This then leads to the following bottom-up property for Boolean BAG queries:

Corollary 3.28 *For every BAG $\mathcal{B}$ there is a BAG $\mathcal{B}'$ having only synthesized attributes, such that $\mathcal{B}$ and $\mathcal{B}'$ express the same Boolean query.*

In the general case of arbitrary attribute grammars, where semantic rules can be arbitrary computable functions, it is well known that the use of inherited attributes can be simulated using synthesized attributes only [Knu68]; we thus see that a similar phenomenon holds when semantic rules can only be propositional formulas.

Corollary 3.28 does not hold for BAGs expressing unary queries, as illustrated in the following example.

Example 3.29 Consider again the grammar in Example 3.5. A query that can only be expressed with synthesized *and* inherited attributes is the one that retrieves all nodes, if both the first and the last letter of the string are x's and retrieves no nodes otherwise. This query can not be expressed with only synthesized attributes. Indeed, every synthesized BAG already has to decide to select the last letter of the string without having visited the first letter, that is, without knowing whether the first letter carries an x. A same argument holds for BAGs having only inherited attributes. ■

3.3 Expressive power of RAGs

In this section we characterize RAGs as the queries defined by first-order inductions of linear depth, or, equivalently those computable in linear time on a parallel machine with polynomially many processors. We also show that, in contrast to BAGs, even for Boolean queries, synthesized RAGs are strictly less expressive than RAGs with both synthesized and inherited attributes. Hence, there is no bottom-up property for Boolean RAG queries. In the last subsection, we discuss the relationship between MSO and RAGs. First, we introduce the necessary logical definitions.

3.3.1 Fixpoint logic

See Ebbinghaus and Flum's book [EF95] for more background on the logics we are about to define.

Partial and least fixpoint logic

Fixpoint logic allows first-order logic formulas to be iterated. We will consider several kinds of fixpoint logics. Let $\varphi(z_1, \dots, z_k, Z)$ be a first-order logic formula. The z_i's are free individual variables, Z is a k-ary relation variable that can be used in φ in addition to the relation symbols provided by the vocabulary. On any tree $\mathbf{t}$, φ defines the following relations obtained by iterating φ starting with the empty relation for Z. Define

$$\begin{aligned} \varphi^0(\mathbf{t}) &:= \emptyset; \\ \varphi^{i+1}(\mathbf{t}) &:= \{(\mathbf{n}_1, \dots, \mathbf{n}_k) \mid \mathbf{t} \models \varphi[\mathbf{n}_1, \dots, \mathbf{n}_k, \varphi^i(\mathbf{t})]\}. \end{aligned}$$

We say that φ converges to a fixpoint on $\mathbf{t}$ if there exists an n such that $\varphi^n(\mathbf{t}) = \varphi^{n+1}(\mathbf{t})$. We denote this fixpoint by $\varphi^\infty(\mathbf{t})$. If φ does not reach a fixpoint on $\mathbf{t}$ we define $\varphi^\infty(\mathbf{t})$ as the empty set. We define *partial fixpoint logic (PFP)* as follows: formulas are constructed just as in first-order logic, with the addition that we also allow formulas of the form $\mathrm{PFP}[\varphi, Z](z_1, \dots, z_k)$, where Z is k-ary and $\varphi(z_1, \dots, z_k, Z)$ is a first-order logic formula. The semantics is as follows: for any tree $\mathbf{t}$, and nodes $\mathbf{n}_1$, …, $\mathbf{n}_k$ of $\mathbf{t}$,

$$\mathbf{t} \models \mathrm{PFP}[\varphi, Z][\mathbf{n}_1, \dots, \mathbf{n}_k] \quad \Leftrightarrow \quad (\mathbf{n}_1, \dots, \mathbf{n}_k) \in \varphi^\infty(\mathbf{t}).$$

The formula φ is called *positive* if every occurrence of the variable Z occurs under an even number of negations. For such formulas the above described iteration process always reaches a fixpoint after a finite number of stages. Moreover, this fixpoint is also the least fixpoint of the operator defined by φ: over a tree $\mathbf{t}$, this operator maps k-ary relations R over the domain of $\mathbf{t}$ to k-ary relations and is defined by

$$\varphi(R) := \{(\mathbf{n_1}, \dots, \mathbf{n}_k) \mid \mathbf{t} \models \varphi[\mathbf{n_1}, \dots, \mathbf{n}_k, R]\}.$$

We now define *least fixpoint logic (LFP)*, in the same way as PFP except that for each formula of the form $\mathrm{LFP}[\varphi, Z](z_1, \dots, z_k)$, φ has to be positive.

Note that our definitions of PFP and LFP differ from those in the literature: we do not allow nesting of fixpoints and we do not allow parameters in the formula constituting the fixpoint. However, since these can be dispensed with, our definitions are equivalent to the usual ones [EF95].

Fixpoints of linear depth

Consider a PFP-formula of the form $\mathrm{PFP}[\varphi, Z](z_1, \ldots, z_k)$. If there exist natural numbers c and d such that for every tree $\mathbf{t}$, φ reaches its partial fixpoint after at most $c \cdot |\mathbf{t}| + d$ iterations, where $|\mathbf{t}|$ denotes the number of nodes in $\mathbf{t}$, then we say that φ is *linearly bounded by the partial fixpoint semantics*. We define the logic PFP-LIN as the fragment of PFP where only partial fixpoints of linearly bounded formulas are allowed.

LFP-LIN is then the fragment of PFP-LIN that only allows formulas under the fixpoint operator that are both positive and linearly bounded.

Simultaneous fixpoint logic

Let $\varphi_1(\bar{z}_1, Z_1, \ldots, Z_k), \ldots, \varphi_k(\bar{z}_k, Z_1, \ldots, Z_k)$ be a system of first-order formulas, where for $j = 1, \ldots, k$, Z_j is an r_j-ary relation variable. On a tree $\mathbf{t}$, consider for $j = 1, \ldots, k$, the stages defined by

$$\begin{aligned} \varphi_j^0(\mathbf{t}) &:= \emptyset; \\ \varphi_j^{i+1}(\mathbf{t}) &:= \{(\mathbf{n}_1, \ldots, \mathbf{n}_{r_j}) \mid \mathbf{t} \models \varphi_j[\mathbf{n}_1, \ldots, \mathbf{n}_{r_j}, \varphi_1^i(\mathbf{t}), \ldots, \varphi_k^i(\mathbf{t})]\}. \end{aligned}$$

We say that this system reaches a *simultaneous fixpoint* on $\mathbf{t}$ if there exists an n such that for all $j = 1, \ldots, k$, $\varphi_j^n(\mathbf{t}) = \varphi_j^{n+1}(\mathbf{t})$. We denote the relation defined by φ_j in this fixpoint by $\varphi_j^\infty(\mathbf{t})$. If there does not exist a simultaneous fixpoint on $\mathbf{t}$, then $\varphi_j^\infty(\mathbf{t})$ is defined as the empty set. We now define *simultaneous partial fixpoint logic (S-PFP)* as follows: formulas are constructed just as in first-order logic, with the addition that we also allow formulas of the form $\text{S-PFP}_j[\varphi_1, \ldots, \varphi_k, Z_1, \ldots, Z_k](z_1, \ldots, z_r)$, where Z_j is r-ary and φ_i is a first-order formula for $i = 1, \ldots, k$. The semantics is defined as follows: for any tree $\mathbf{t}$, and nodes $\mathbf{n}_1, \ldots, \mathbf{n}_r$

$$\mathbf{t} \models \text{S-PFP}_j[\bar{\varphi}, \overline{Z}][\mathbf{n}_1, \ldots, \mathbf{n}_r] \quad \Leftrightarrow \quad (\mathbf{n}_1, \ldots, \mathbf{n}_r) \in \varphi_j^\infty(\mathbf{t}).$$

We say that the system of first-order formulas $\varphi_1, \ldots, \varphi_k$ is *linearly bounded by the simultaneous partial fixpoint semantics* if there exist natural numbers c and d such that for every derivation tree $\mathbf{t}$, the system of first-order formulas $\varphi_1, \ldots, \varphi_k$ reaches its simultaneous partial fixpoint after at most $c \cdot |\mathbf{t}| + d$ iterations. We define the logic S-PFP-LIN as the fragment of S-PFP where only simultaneous partial fixpoints of linearly bounded systems of first-order formulas are allowed.

The next proposition states that S-PFP-LIN is equivalent to PFP-LIN. In particular this means that mutual recursion can be replaced by simple recursion while preserving linearly boundedness. The proof is exactly as the proof of the Simultaneous Induction Lemma known from the theory of inductive definitions and finite model theory [Mos74, EF95].

Proposition 3.30 *Every S-PFP-LIN-formula of the form*

$$\text{S-PFP}_j[\overline{\varphi}, \overline{Z}](\bar{z}),$$

where $j \in \{1, \dots, k\}$, is equivalent to a PFP-LIN-formula of the form

$$(\exists \bar{u})(\mathrm{PFP}[\psi, Z](\bar{z}\bar{u}));$$

3.3.2 Main Theorem

We now relate RAGs to PFP-LIN. In particular, PFP-LIN-formulas can express k-ary queries in the following way:

Definition 3.31 Let $\varphi(x_1, \dots, x_k)$ be a PFP-LIN-formula. Then φ expresses the k-ary query $\mathcal{Q}$ defined by

$$\mathcal{Q}(\mathbf{t}) := \{(\mathbf{n}_1, \dots, \mathbf{n}_k) \mid \mathbf{t} \models \varphi[\mathbf{n}_1, \dots, \mathbf{n}_k]\},$$

for every tree $\mathbf{t}$.

Theorem 3.32 *A query is expressible by a RAG if and only if it is definable in PFP-LIN.*

Proof. *Only if.* Let $\mathcal{R}$ be a RAG. We assume w.l.o.g. that no semantic rule contains a variable of $(z_a)_{a \in A}$, z_0, z_1, z_2, $\dots$. We define an S-PFP-LIN-formula that simulates $\mathcal{R}$. As induction variables of this system we have an[2] $(r_a + 1)$-ary relation variable Z_a for each attribute a; Z_a stands for the set of tuples $(\mathbf{n}, \mathbf{n}_1, \dots, \mathbf{n}_{r_a})$, where $\mathbf{n}$ is a node labeled X such that $a \in \mathrm{Att}(X)$, and $(\mathbf{n}_1, \dots, \mathbf{n}_{r_a})$ is a tuple in the currently computed value of $\mathcal{R}(\mathbf{t})(a(\mathbf{n}))$. For each attribute a there is a formula $\varphi_a(z_a, x_1, \dots, x_{r_a}, (Z_b)_{b \in A})$, defining the new value of Z_a from the old values of the Z_b's, built up as follows. Consider a rule $a(i) := \varphi(x_1, \dots, x_{r_a})$ of $\mathcal{R}$ in the context (p, a, i) for some production $p = X_0 \to X_1 \dots X_n$ of G. The formula $\rho_{p,a,i}(z_a, x_1, \dots, x_{r_a}, (Z_b)_{b \in A})$ is defined as

$$(\exists z_0)(\exists z_1) \dots (\exists z_n) \left(\bigwedge_{j=0}^{n} O_{X_j}(z_j) \wedge \bigwedge_{j=1}^{n} S_j(z_0, z_j) \wedge z_a = z_i \wedge \widehat{\varphi} \right),$$

where $\widehat{\varphi}$ is obtained from φ by replacing each occurrence of $b(j)(\bar{d})$ by $Z_b(z_j, \bar{d})$, and by replacing each occurrence of the constant symbol $\mathbf{k}$ by z_k. The formula φ_a then is the disjunction over all rules defining the attribute a:

$$\begin{aligned} &\varphi_a(z_a, x_1 \dots, x_{r_a}, (Z_b)_{b \in A}) := \\ &\qquad \bigvee \{\rho_{p,a,i}(z_a, x_1 \dots, x_{r_a}, (Z_b)_{b \in A}) \mid (p, a, i) \text{ is a context}\}. \end{aligned}$$

Let $\sigma(\bar{z})$ be the formula

$$(\exists z) \left(O_U(z) \wedge \text{S-PFP}_1[\varphi_{result}, (\varphi_a)_{a \in A - \{result\}}, (Z_a)_{(a \in A)}](z, \bar{z}) \right).$$

[2]Recall that r_a is the arity of the attribute a.

Here $|\bar{z}|$ equals the arity of *result*. By an easy induction on i one can now show that for any node $\mathbf{n}$ and attribute a, if $\mathcal{R}_i(\mathbf{t})$ is defined on $a(\mathbf{n})$ then for all nodes $\mathbf{m}_1, \ldots, \mathbf{m}_{r_a}$,

$$(\mathbf{n}, \mathbf{m}_1, \ldots, \mathbf{m}_{r_a}) \in \varphi_a^i(\mathbf{t}) \quad \Leftrightarrow \quad (\mathbf{m}_1, \ldots, \mathbf{m}_{r_a}) \in \mathcal{R}_i(\mathbf{t})(a(\mathbf{n})).$$

This implies that

$$(\mathbf{n}, \mathbf{m}_1, \ldots, \mathbf{m}_{r_a}) \in \varphi_a^\infty(\mathbf{t}) \quad \Leftrightarrow \quad (\mathbf{n}, \mathbf{m}_1, \ldots, \mathbf{m}_{r_a}) \in \mathcal{R}(\mathbf{t})(a(\mathbf{n})).$$

Further, for each tree $\mathbf{t}$, let $l_\mathbf{t}$ be the smallest integer such that $\mathcal{R}_{l_\mathbf{t}} = \mathcal{R}_{l_\mathbf{t}+1}$. Then, obviously, $l_\mathbf{t} \leq |A| \cdot |\mathbf{t}|$. Hence, the S-PFP-formula in σ reaches its fixpoint after at most $|A| \cdot |\mathbf{t}|$ iterations. Proposition 3.30 now gives us the desired formula in PFP-LIN.

If. The crux of the proof is the simple observation that there is a RAG that computes all the relations that make up a derivation tree, viewed as a relational structure, in one bottom-up pass over the tree. In a subsequent top-down pass, we can make these relations available at all nodes. A linearly-bounded iteration of a first-order formula can then be simulated in one preorder traversal of the tree, where the different stages are passed over as relational attribute values.

We now formally describe the RAG $\mathcal{R}$ that expresses the query defined by a PFP-LIN-formula. To compute the relations that make up a derivation tree we make use of the binary attributes $S'_1, \ldots, S'_r$, where r is the maximum width of any production in P, and the unary attributes $(O'_X)_{X \in N \cup T}$. These attributes are synthesized for non-terminals and inherited for terminals. They are defined by the following semantic rules. Consider the production $p = X_0 \to X_1 \ldots X_n$. For $j =, 1 \ldots, r$, define

$$S'_j(0) := \begin{cases} \bigcup_{i=1}^{n} S'_j(i) \cup \{(\mathbf{0}, \mathbf{j})\} & \text{if } j \leq n, \\ \bigcup_{i=1}^{n} S'_j(i) & \text{if } j > n. \end{cases}$$

For each $X \in N \cup T$, define

$$O'_X(0) := \begin{cases} \bigcup_{i=1}^{n} O'_X(i) \cup \{(\mathbf{0})\} & \text{if } X = X_0, \\ \bigcup_{i=1}^{n} O'_X(i) & \text{otherwise.} \end{cases}$$

For each i, such that X_i is a terminal, and for each $j = 1, \ldots, r$, define

$$S'_j(i) := \emptyset,$$

and for each $X \in N \cup T$ define

$$O'_X(i) := \begin{cases} \{(\mathbf{i})\} & \text{if } X = X_i, \\ \emptyset & \text{otherwise.} \end{cases}$$

The values of the relations $(S'_j)_{1\leq j\leq r}$ and $(O'_X)_{X\in N\cup T}$ at the root then form the relational structure that represents the derivation tree. These values are now made available to the other nodes in the attributes $(S_j)_{1\leq j\leq r}$ and $(O_X)_{X\in N\cup T}$. These attributes are synthesized for the start symbol U and inherited for all other symbols, and are defined via the following rules. For every production of the form $U \to X_1 \dots X_n$, for every $j = 1, \dots, r$, and $X \in N \cup T$, define

$$S_j(0) := S'_j(0) \qquad \text{and} \qquad O_X(0) := O'_X(0).$$

For every production of the form $X_0 \to X_1 \dots X_n$, where $X_0 \neq U$, and for every $j = 1, \dots, r$, $i = 1, \dots, n$, and $X \in N \cup T$, define

$$S_j(i) := S_j(0) \qquad \text{and} \qquad O_X(i) := O_X(0).$$

Let φ be a PFP-LIN-formula. Then φ is a first-order combination of formulas of the form $S_j(z_1, z_2)$, $O_X(z)$, and $\mathrm{PFP}[\psi, Z](z_1, \dots, z_k)$. Each relation S_j and O_X is already available at the root. Hence, it suffices to compute each subformula $\mathrm{PFP}[\psi, Z](\bar{z})$ occurring in φ in some attribute and make it available at the root.

Let c and d be numbers such that ψ reaches its fixpoint after at most $c \cdot |\mathbf{t}| + d$ iterations on each tree $\mathbf{t}$. For any i, there exists a first-order formula $\psi^i(\bar{z}, Z)$ that defines i stages of ψ at once. Indeed, let $y_1, \dots, y_k$ be variables that do not occur in ψ. Then, define $\psi^1(\bar{z}, Z)$ as $\psi(\bar{z}, Z)$, and for $i > 1$, $\psi^i(\bar{z}, Z)$ as the formula obtained from ψ by replacing each atomic formula of the form $Z(\bar{d})$, by the formula $(\exists \bar{y})(\bar{y} = \bar{d} \wedge (\exists \bar{z})(\bar{z} = \bar{y} \wedge \psi^{i-1}(\bar{z}, Z)))$.

The RAG $\mathcal{R}^\psi$ evaluates the formula $\mathrm{PFP}[\psi, Z](\bar{z})$ in the following way: First, d stages of ψ are evaluated at the root of the tree; this is achieved by evaluating the formula $\psi^d(\bar{z}, \emptyset)$. Then $\mathcal{R}^\psi$ makes a preorder traversal of the tree while evaluating c stages of ψ, i.e., evaluating the formula ψ^c, at each node.

We now formally describe the RAG $\mathcal{R}^\psi$. It uses the k-ary attribute Z, which is synthesized for the start symbol and inherited for the other grammar symbols, and the k-ary attribute Z', which is synthesized for the non-terminals and inherited for the terminals. The attributes Z and Z' are defined by the following semantic rules. Consider a production of the form $p = X_0 \to X_1 \dots X_n$.

1. If $X_0 = U$ then add the rule

$$Z(0) := \{\bar{z} \mid \psi^{d+c}(\bar{z}, \emptyset)\}.$$

2. Further, define

$$Z(1) := \{\bar{z} \mid \psi^c_1(\bar{z})\},$$

where ψ^c_1 is obtained from ψ^c by replacing each occurrence of $O_X(z)$ or $S_h(z_1, z_2)$ by $O_X(1)(z)$ or $S_h(1)(z_1, z_2)$ respectively, and by replacing each occurrence of $Z(\bar{d})$ by $Z(0)(\bar{d})$.

3. For each j, such that X_j is a terminal, define

$$Z'(j) := Z(j).$$

4. For each $j = 2, \ldots, n$, we have
$$Z(j) := \{\bar{z} \mid \psi^c_j(\bar{z})\},$$
where ψ^c_j is obtained from ψ^c by replacing each occurrence of $O_X(z)$ or $S_h(z_1, z_2)$ by $O_X(j)(z)$ or $S_h(j)(z_1, z_2)$ respectively, and by replacing each occurrence of $Z(\bar{d})$ by $Z'(j-1)(\bar{d})$.

5. Finally, define
$$Z'(0) := Z'(n).$$

Note that on every tree $\mathbf{t}$, the evaluation of $\mathcal{R}^\psi$ performs exactly $2 \cdot |\mathbf{t}|$ iterations. In each iteration exactly one attribute Z or exactly one attribute Z' is defined. For a tree $\mathbf{t}$, let for $i \geq 1$, $\alpha_i(\mathbf{t})$ ($\beta_i(\mathbf{t})$) be the number of Z (Z') attributes that are defined in $\mathcal{R}_i(\mathbf{t})$. The correctness of this construction now follows from the following lemma:

Lemma 3.33 *Let* $\mathbf{t}$ *be a derivation tree, let* $\mathbf{n}$ *be a node of* $\mathbf{t}$ *and let* $a \in \{Z, Z'\}$*. If* $a(\mathbf{n})$ *is defined in* $\mathcal{R}^\psi_i(\mathbf{t})$ *but not in* $\mathcal{R}^\psi_{i-1}(\mathbf{t})$*, then*
$$\mathcal{R}^\psi_i(\mathbf{t})(a(\mathbf{n})) = \{\bar{\mathbf{n}} \mid \mathbf{t} \models \psi^{c \cdot \alpha_i(\mathbf{t}) + d}[\bar{\mathbf{n}}, \emptyset]\}.$$

This lemma can be proved by induction on the pair $(\alpha_i(\mathbf{t}), \beta_i(\mathbf{t}))$.

Hence, $\mathcal{R}_\psi(\mathbf{t})(Z'(\mathbf{r}))$ equals the relation defined by $\mathrm{PFP}[\psi, Z](\bar{z})$, where $\mathbf{r}$ is the root of $\mathbf{t}$. ∎

The logic PFP-LIN has a rather bizarre syntax, as it allows the iteration of a formula only when that formula is linearly bounded, which is not an obvious syntactic property. Actually, we do not know whether linear boundedness of first-order formulas over derivation trees of some fixed grammar is decidable. Over graphs the property can be shown undecidable by a reduction from validity; but over derivation trees (or equivalently Σ-trees, for some ranked alphabet Σ), satisfiability and validity of first-order logic (even monadic second-order logic) is decidable [Don70, TW68, Tho97b].

This problem of bizarre syntax can be avoided, however, by defining PFP-LIN in an alternative manner. Under this alternative, the iteration of *any* formula is allowed (so that the syntax is now trivially decidable). We then build into the semantics that the iteration is performed exactly n times, where n is the cardinality of the domain. To this end, one could also employ first-order logic extended with for-loops [NOTV98] where head formulas of for-loops are only allowed to define the domain of the structure at hand. It is not difficult to adapt the proof of Theorem 3.32 for this alternative view of PFP-LIN.

3.3.3 Complexity of RAGs

Immerman [Imm89] showed that LFP-LIN captures the complexity class CRAM$[n]$ consisting of all queries computable in time $O(n)$ by a parallel machine with polynomially many processors.

Theorem 3.34 [Imm89] *LFP-LIN* = CRAM[n] *on the class of all ordered finite structures.*

Using Theorem 3.32 and Theorem 3.34, we show the following:

Corollary 3.35 *A query is expressible by a RAG if and only if it is computable in linear parallel time with polynomially many processors.*

Proof. In Lemma 3.36(ii) we show that PFP-LIN = LFP-LIN on the class of all trees. Hence, by Theorem 3.34, we have that PFP-LIN = CRAM[n] on the class of all ordered trees. By Theorem 3.32, it then suffices to show that PFP-LIN = CRAM[n] on the class of all trees without a linear order. The order requirement in Theorem 3.34 is only needed to show that every CRAM[n] program can indeed be simulated by an LFP-LIN formula. Hence, it readily follows that PFP-LIN $\subseteq$ CRAM[n] on the class of all trees without a linear order. It remains to show the converse inclusion. Let P be a CRAM[n] program over trees with a linear order and let ξ be the PFP-LIN formula simulating P. For expository purposes assume ξ is of the form PFP$[\varphi, X](\bar{x})$. By Lemma 3.36(i), there exists a PFP-LIN formula PFP$[\psi, Y](y_1, y_2)$ computing a linear order. We can not simply plug in the formula PFP$[\psi, Y](y_1, y_2)$ for each occurrence of $y_1 < y_2$ in ξ because we do not allow nesting of fixpoints. However, we can use the following composition trick: we first compute the ordering and only then start iterating φ. That is, we just use the formula

$$\text{S-PFP}_1[\varphi', \psi, X, Y](\bar{x}), \qquad (*)$$

where φ' is the formula $\rho_{\text{lin. ord.}}(Y) \wedge \hat{\varphi}$. Here, $\rho_{\text{lin. ord.}}(Y)$ is the first-order logic formula defining Y as a *total* linear order, and $\hat{\varphi}$ is the formula obtained from φ by replacing each occurrence of $y_1 < y_2$ by $Y(y_1, y_2)$. By definition of φ', the iteration of φ only starts when Y is indeed a linear order. Further, φ' is linearly bounded since both φ and ψ are. By Proposition 3.30, $(*)$ is equivalent to a PFP-LIN formula. ■

It remains to prove the following lemma.

Lemma 3.36 (i) *There exists a PFP-LIN-formula that uniformly defines a total order on all trees.*

(ii) *PFP-LIN = LFP-LIN on the class of all trees.*

Proof. (i) Example 3.20 shows how an ordering of a binary tree can be obtained using a RAG. It is straightforward to generalize this construction to arbitrary derivation trees. By Theorem 3.32, this RAG is equivalent to a PFP-LIN-formula.

(ii) In Example 3.20, we saw how we can compute an ordering of a tree using a RAG. We can also compute this ordering directly in LFP-LIN. Hence, the equivalence of LFP-LIN and PFP-LIN on trees reduces to their equivalence on ordered trees (we can compose the computation of the ordering with other PFP constructs like in the proof of the previous theorem). The proof of the latter equivalence is similar to the proof of the known fact that LFP equals PFP$|_{\text{PTIME}}$ on ordered structures [EF95, Thm 7.4.14] (see also [AV95]). Here $\mathrm{PFP}|_{\mathrm{PTIME}}$ denotes the fragment of PFP, where every fixpoint is reached after at most a polynomial number of iterations. ∎

3.3.4 No bottom-up property for RAGs

In this section we prove that *synthesized* RAGs, i.e., RAGs that only use synthesized attributes, are strictly less expressive than RAGs that can use both synthesized and inherited attributes.

For the rest of this section, let G be the grammar $\{U \to LL, L \to L, L \to f\}$. Derivation trees of this grammar consists simply of two monadic trees concatenated at the root:

$$\begin{array}{ccc} & U & \\ \swarrow & & \searrow \\ L & & L \\ \downarrow & & \downarrow \\ \vdots & & \vdots \\ \downarrow & & \downarrow \\ L & & L \end{array}$$

Let `equal_subtree` be the query that is true on $\mathbf{t}$ when the left subtree has the same number of nodes as the right subtree. We show that this query cannot be expressed by a synthesized RAG. However, it can be expressed by a RAG.

Proposition 3.37 *The query* `equal_subtree` *is expressible by a RAG.*

Proof. By Theorem 3.32, it suffices to show that `equal_subtree` is definable in PFP-LIN:

$$\varphi := (\exists x)(\exists y)\big(x \neq y \wedge O_f(x) \wedge O_f(y) \wedge \mathrm{PFP}[\sigma, X](x,y)\big),$$

where

$$\begin{aligned} \sigma(x,y,X) := \;& (\exists z)\big(O_U(z) \wedge S_1(z,x) \wedge S_2(z,y)\big) \\ & \vee (\exists x')(\exists y')\big(X(x',y') \wedge S_1(x',x) \wedge S_1(y',y)\big). \end{aligned}$$

This formula maintains a binary relation X. In the first iteration of σ, the first node of the left subtree and the first node of the right subtree are put in X. In the following iterations, the next pair of corresponding nodes is added to X, provided it exists. Hence, σ iterates at most $|\mathbf{t}|/2$ times on a tree $\mathbf{t}$, and thus belongs to PFP-LIN. The formula φ then becomes true if both the last node of the left subtree and the last node of the right subtree belong to X. ∎

In the rest of this section we prove that `equal_subtree` cannot be expressed by a synthesized RAG.

Definition 3.38 A *simple* RAG is a synthesized RAG over the attribute grammar vocabulary that has only the attribute c for L and only the attribute *result* for U.

We focus attention on simple RAGs and show later that any synthesized RAG can be transformed into an equivalent simple one.

For any integers n_1 and n_2 greater than 1, let $\mathbf{t}(n_1, n_2)$ denote the tree which has a left subtree of length n_1 and a right subtree of length n_2. Let $\mathcal{R}$ be a simple RAG. In Lemma 3.40 we show that the values of $\mathcal{R}(\mathbf{t}(n_1, n_2))(c(\mathbf{n}_1))$ and $\mathcal{R}(\mathbf{t}(n_1, n_2))(c(\mathbf{n}_2))$, where $\mathbf{n}_1$ is the first child and $\mathbf{n}_2$ is the second child of the root, can be uniformly defined in PFP over a structure that, essentially, only contains an ordering of part of the domain of $\mathbf{t}(n_1, n_2)$. First, we need some definitions.

Definition 3.39 Let $\tau_< = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, <\}$ be the vocabulary consisting of the constant symbols $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{2}$, and the binary relation symbol $<$. Let n_1 and n_2 be two integers greater than 1.

1. Define $\mathcal{N}_1(n_1, n_2)$ as the $\tau_<$-structure with domain $\{1, \ldots, n_1 + n_2 + 1\}$, where $\mathbf{0} = n_1 + n_2 + 1$, $\mathbf{1} = n_1$, $\mathbf{2} = n_1 + n_2$, and where $<$ is interpreted as the total order on $\{1, \ldots, n_1\}$;

2. Define $\mathcal{N}_2(n_1, n_2)$ similarly as $\mathcal{N}_1(n_1, n_2)$, except that now $<$ is interpreted as the total order on $\{n_1 + 1, \ldots, n_1 + n_2\}$.

Let $\xi(x_1, \ldots, x_\ell)$ be a PFP-formula over the vocabulary $\tau_<$. We define, for $i \in \{1, 2\}$, $\xi(\mathcal{N}_i(n_1, n_2))$ as the relation defined by ξ on $\mathcal{N}_i(n_1, n_2)$, i.e., by

$$\{(\mathbf{n}_1, \ldots, \mathbf{n}_\ell) \mid \mathcal{N}_i(n_1, n_2) \models \xi[\mathbf{n}_1, \ldots, \mathbf{n}_\ell]\}.$$

Lemma 3.40 *Let $\mathcal{R}$ be a simple RAG. There exists a PFP formula*

$$\xi(x_1, \ldots, x_{r_c}),$$

such that for all $n_1, n_2 > 1$

$$\mathcal{R}(\mathbf{t}(n_1, n_2))(c(\mathbf{n}_1)) = \xi(\mathcal{N}_1(n_1, n_2)),$$

and

$$\mathcal{R}(\mathbf{t}(n_1, n_2))(c(\mathbf{n}_2)) = \xi(\mathcal{N}_2(n_1, n_2)),$$

where $\mathbf{n}_1$ is the first child and $\mathbf{n}_2$ is the second child of the root of $\mathbf{t}(n_1, n_2)$.

Proof. Let $c(0) := \varphi(\bar{x})$ be the rule in the context $(L \to f, c, 0)$, and let $c(0) := \psi(\bar{x})$ be the rule in the context $(L \to L, c, 0)$. Let y_1, y_2, y_3, z_1, z_2, z_3, v_1, $\ldots$, v_{r_c} be variables not occurring in φ or ψ. Then define $\xi(\bar{x})$ as the formula

$$(\exists y_1)\,(\mathrm{root}(y_1) \wedge \mathrm{PFP}[\sigma, X](y_1, y_1, y_1, \bar{x}))\,.$$

Here $\mathrm{root}(y_1)$ is an FO-formula that defines the root of the tree, and

$$\sigma(y_1, y_2, y_3, x_1, \ldots, x_{r_c}, X)$$

is the formula

$$\begin{aligned}&(y_1 = \text{First} \wedge (y_2 = y_3 \rightarrow \varphi'(\bar{x})))\\ &\quad \vee (\exists z_1)(\text{Succ}(z_1) = y_1 \wedge (\exists z_2)(\exists z_3)(\exists \bar{v})(X(z_1, z_2, z_3, \bar{v}) \wedge (y_2 = y_3 \rightarrow \psi'(\bar{x}))),\end{aligned}$$

where First is the first element of $<$ and Succ is the successor function obtained from $<$; φ' is obtained from φ by replacing each occurrence of $\mathbf{1}$ by y_1 and each occurrence of $\mathbf{0}$ by $\text{Succ}(y_1)$; ψ' is obtained from ψ by replacing each occurrence of $c(1)(\bar{d})$ by $X(z_1, z_1, z_1, \bar{d})$, each occurrence of $\mathbf{0}$ by y_1, and each occurrence of $\mathbf{1}$ by z_1. The variables y_2 and y_3 make sure that the relation X is never empty. It might happen that φ defines an empty relation; then the fixpoint would be empty. ■

The next lemma is an immediate observation.

Lemma 3.41 *Let $\xi(x_1, \ldots, x_\ell)$ be a PFP-formula over $\tau_<$. For any n_1 and n_2 greater than 1, and $i \in \{1, \ldots, \ell\}$:*

(i) If $\mathcal{N}_1(n_1, n_2) \models \xi[\mathbf{n}_1, \ldots, \mathbf{n}_\ell]$ and

$$\mathbf{n}_i \in \{n_1 + 1, \ldots, n_1 + n_2 - 1\} - \{\mathbf{n}_1, \ldots, \mathbf{n}_{i-1}, \mathbf{n}_i, \ldots \mathbf{n}_\ell\}$$

then for all $\mathbf{m} \in \{n_1 + 1, \ldots, n_1 + n_2 - 1\} - \{\mathbf{n}_1, \ldots, \mathbf{n}_\ell\}$,

$$\mathcal{N}_1(n_1, n_2) \models \xi[\mathbf{n}_1, \ldots, \mathbf{n}_{i-1}, \mathbf{m}, \mathbf{n}_{i+1}, \ldots, \mathbf{n}_\ell].$$

(ii) If $\mathcal{N}_2(n_1, n_2) \models \xi[\mathbf{n}_1, \ldots, \mathbf{n}_\ell]$ and

$$\mathbf{n}_i \in \{1, \ldots, n_1 - 1\} - \{\mathbf{n}_1, \ldots, \mathbf{n}_{i-1}, \mathbf{n}_i, \ldots \mathbf{n}_\ell\}$$

then for all $\mathbf{m} \in \{1, \ldots, n_1 - 1\} - \{\mathbf{n}_1, \ldots, \mathbf{n}_\ell\}$,

$$\mathcal{N}_2(n_1, n_2) \models \xi[\mathbf{n}_1, \ldots, \mathbf{n}_{i-1}, \mathbf{m}, \mathbf{n}_{i+1}, \ldots, \mathbf{n}_\ell].$$

Proof. This follows from the fact that PFP cannot distinguish between elements that are automorphic. Clearly, the transposition of any two elements $\mathbf{n}, \mathbf{n}' \in \{n_1 + 1, \ldots, n_1 + n_2 - 1\}$ is an automorphism of $\mathcal{N}_1(n_1, n_2)$, and the transposition of any two elements $\mathbf{n}, \mathbf{n}' \in \{1, \ldots, n_1 - 1\}$ is an automorphism of $\mathcal{N}_2(n_1, n_2)$. ■

Definition 3.42 Let $\xi(x_1, \ldots, x_\ell)$ be a PFP-formula over the vocabulary $\tau_<$. Let n_1 and n_2 be two integers greater than 1. Let $\tau_{c,\ell}$ be the vocabulary $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, c_1, c_2\}$, consisting of the constant symbols $\mathbf{0}, \mathbf{1}, \mathbf{2}$ and the ℓ-ary relation symbols c_1 and c_2. Define $\mathbf{t}(\xi, n_1, n_2)$ as the $\tau_{c,\ell}$-structure with domain $\{1, \ldots, n_1 + n_2 + 1\}$, where $\mathbf{0} = n_1 + n_2 + 1$, $\mathbf{1} = n_1$, $\mathbf{2} = n_1 + n_2$, $c_1 = \xi(\mathcal{N}_1(n_1, n_2))$, and $c_2 = \xi(\mathcal{N}_2(n_1, n_2))$.

We now show that, under certain assumptions, on the structures $\mathbf{t}(\xi, n_1, n_2)$ every FO-formula can be split into formulas that essentially speak only about the relation c_1 or only about the relation c_2, but not about both.

Lemma 3.43 *Let ξ be a PFP-formula with ℓ free variables. Assume there are FO-formulas $P_1(x)$ and $P_2(x)$ such that for all $n_1, n_2 > 1$,*

$$\mathbf{t}(\xi, n_1, n_2) \models P_1[\mathbf{n}] \quad \Leftrightarrow \quad \mathbf{n} \in \{1, \dots, n_1 - 1\},$$

and

$$\mathbf{t}(\xi, n_1, n_2) \models P_2[\mathbf{n}] \quad \Leftrightarrow \quad \mathbf{n} \in \{n_1 + 1, \dots, n_1 + n_2 - 1\}.$$

For every FO-formula $\varphi(x_1, \dots, x_k)$ over the vocabulary $\tau_{c,\ell}$ expanded with the unary relational symbols P_1 and P_2, there exists a disjunction $\psi(x_1, \dots, x_k)$ of FO-formulas of the form

$$\alpha(x_1, \dots, x_k) \wedge \beta(x_1, \dots, x_k) \wedge \bigwedge_{j=1}^{k} \omega_j(x_j),$$

where α does neither contain P_2 nor c_2, β does neither contain P_1 nor c_1, and each $\omega_j(x_j)$ is of the form $x_j = \mathbf{0}$, $x_j = \mathbf{1}$, $x_j = \mathbf{2}$, $P_1(x_j)$ or $P_2(x_j)$. For every $n_1, n_2 > 1$, and for every $\mathbf{n}_1, \dots, \mathbf{n}_k \in \{1, \dots, n_1 + n_2 + 1\}$ it holds that

$$\mathbf{t}(\xi, n_1, n_2) \models \varphi[\mathbf{n}_1, \dots, \mathbf{n}_k] \quad \Leftrightarrow \quad \mathbf{t}(\xi, n_1, n_2) \models \psi[\mathbf{n}_1, \dots, \mathbf{n}_k].$$

Proof. The proof goes by induction on the structure of φ. We can assume w.l.o.g. that constants only appear in atomic formulas that are equalities.

1. If φ equals $\mathbf{c} = \mathbf{c}'$, where $\mathbf{c}, \mathbf{c}' \in \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ and $\mathbf{c} \neq \mathbf{c}'$, then ψ is the empty disjunction.

2. If φ equals $\mathbf{c} = \mathbf{c}$, where $\mathbf{c} \in \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$, then ψ is

$$(\text{true} \wedge \text{true}).$$

3. Suppose $\varphi(x)$ is $x = \mathbf{c}$ or $\mathbf{c} = x$, where $\mathbf{c} \in \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$. Then ψ is

$$(\text{true} \wedge \text{true} \wedge x = \mathbf{c}).$$

4. Suppose $\varphi(x_1, x_2)$ is $x_1 = x_2$. We first introduce some notation. Define D as the set of symbols $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, P_1, P_2\}$. Let p be a natural number. For any $\bar{d} = d_1, \dots, d_p \in D^p$, and variables $\bar{y} = y_1, \dots, y_p$, define for $j = 1, \dots, p$, the formula $\omega_{\bar{d},j}(\bar{y})$ as

$$\omega_{\bar{d},j}(\bar{y}) := \begin{cases} y_j = \mathbf{0} & \text{if } d_j = \mathbf{0}, \\ y_j = \mathbf{1} & \text{if } d_j = \mathbf{1}, \\ y_j = \mathbf{2} & \text{if } d_j = \mathbf{2}, \\ P_1(y_j) & \text{if } d_j = P_1, \\ P_2(y_j) & \text{if } d_j = P_2. \end{cases}$$

Now define

$$\psi(x_1, x_2) := \bigvee_{d \in D} \left(x_1 = x_2 \wedge x_1 = x_2 \wedge \bigwedge_{j=1}^{2} \omega_{d,d,j}(x_1, x_2) \right).$$

5. Suppose $\varphi(x_1, \ldots, x_\ell)$ is $c_1(x_1, \ldots, x_\ell)$. Then define

$$\psi := \bigvee_{\bar{d} \in D^\ell} \left(c_1(x_1, \ldots, x_\ell) \wedge \text{true} \wedge \bigwedge_{j=1}^{\ell} \omega_{\bar{d},j}(\bar{x}) \right).$$

6. Suppose $\varphi(x_1, \ldots, x_\ell)$ is $c_2(x_1, \ldots, x_\ell)$. This is symmetric to (5).

7. Suppose $\varphi(x_1, \ldots, x_k)$ is $\varphi_1(y_1, \ldots, y_{k_1}) \vee \varphi_2(z_1, \ldots, z_{k_2})$. Here,

$$\{y_1, \ldots, y_{k_1}\} \cup \{z_1, \ldots, z_{k_1}\} = \{x_1, \ldots, x_k\}.$$

   Let

$$\{u_1, \ldots, u_q\} = \{y_1, \ldots, y_{k_1}\} - \{z_1, \ldots, z_{k_1}\}$$

   and let

$$\{v_1, \ldots, v_p\} = \{z_1, \ldots, z_{k_1}\} - \{y_1, \ldots, y_{k_1}\}.$$

   By the inductive hypothesis, there exists a ψ_1 equivalent to φ_1 of the form

$$\bigvee_{i=1}^{n} \left(\alpha_i^1 \wedge \beta_i^1 \wedge \bigwedge_{j=1}^{k_1} \omega_{i,j}^1 \right),$$

   and a ψ_2 equivalent to φ_2 of the form

$$\bigvee_{i=1}^{m} \left(\alpha_i^2 \wedge \beta_i^2 \wedge \bigwedge_{j=1}^{k_2} \omega_{i,j}^2 \right).$$

   The formula ψ is obtained from ψ_1 and ψ_2, by replacing every disjunct $\alpha_i^1 \wedge \beta_i^1 \wedge \bigwedge_{j=1}^{k_1} \omega_{i,j}^1$ in ψ_1 by

$$\bigvee_{\bar{d} \in D^p} \left(\alpha_i^1 \wedge \beta_i^1 \wedge \bigwedge_{j=1}^{k_1} \omega_{i,j}^1 \wedge \bigwedge_{j=1}^{p} \omega_{\bar{d},j}(\bar{v}) \right),$$

   and by replacing every disjunct $\alpha_i^2 \wedge \beta_i^2 \wedge \bigwedge_{j=1}^{k_2} \omega_{i,j}^2$ in ψ_2 by

$$\bigvee_{\bar{d} \in D^q} \left(\alpha_i^2 \wedge \beta_i^2 \wedge \bigwedge_{j=1}^{k_2} \omega_{i,j}^1 \wedge \bigwedge_{j=1}^{q} \omega_{\bar{d},j}(\bar{u}) \right).$$

8. Suppose $\varphi(x_1, \ldots, x_k)$ is $\neg\varphi'(x_1, \ldots, x_k)$. By the inductive hypothesis, there is a ψ' equivalent to φ' of the form

$$\bigvee_{i=1}^{n} \left(\alpha_i' \wedge \beta_i' \wedge \bigwedge_{j=1}^{k} \omega_{i,j}' \right).$$

Then $\neg\psi'$ is equivalent to

$$\bigwedge_{i=1}^{n}\left(\neg\alpha_i' \vee \neg\beta_i' \vee \bigvee_{j=1}^{k} \neg\omega_{i,j}'\right).$$

We now transform this formula to an equivalent one in the right form. Replace each $\neg\omega_{i,j}'$ of the form

(a) $\neg(x_j = \mathbf{0})$ by $x_j = \mathbf{1} \vee x_j = \mathbf{2} \vee P_1(x_j) \vee P_2(x_j)$;

(b) $\neg(x_j = \mathbf{1})$ by $x_j = \mathbf{0} \vee x_j = \mathbf{2} \vee P_1(x_j) \vee P_2(x_j)$;

(c) $\neg(x_j = \mathbf{2})$ by $x_j = \mathbf{0} \vee x_j = \mathbf{1} \vee P_1(x_j) \vee P_2(x_j)$;

(d) $\neg P_1(x_j)$ by $x_j = \mathbf{0} \vee x_j = \mathbf{1} \vee x_j = \mathbf{2} \vee P_2(x_j)$;

(e) $\neg P_2(x_j)$ by $x_j = \mathbf{0} \vee x_j = \mathbf{1} \vee x_j = \mathbf{2} \vee P_1(x_j)$.

Put the resulting formula in disjunctive normal form (here the literals are the formulas $\neg\alpha_i'$, $\neg\beta_i'$, $z = \mathbf{0}$, $z = \mathbf{1}$, $z = \mathbf{2}$, $P_1(z)$ and $P_2(z)$). Each disjunct now looks like

$$\neg\alpha_{i_1} \wedge \ldots \wedge \neg\alpha_{i_s} \wedge \neg\beta_{i_1} \wedge \ldots \wedge \neg\beta_{i_r} \wedge \delta_1 \wedge \ldots \wedge \delta_b,$$

where each δ is of the form $z = \mathbf{0}$, $z = \mathbf{1}$, $z = \mathbf{2}$, $P_1(z)$ or $P_2(z)$. The disjunct is discarded if there are two different $\delta's$ for the same variable (e.g., one is $z = \mathbf{0}$ and the other is $P_1(z)$). Otherwise, define α as the formula $\neg\alpha_{i_1} \wedge \ldots \wedge \neg\alpha_{i_s}$, and define β as the formula $\neg\beta_{i_1} \wedge \ldots \wedge \neg\beta_{i_r}$. Let $y_1, \ldots, y_g$ be the variables in $\{x_1, \ldots, x_k\}$ for which there is no δ in the disjunct. Then replace this disjunct by

$$\bigvee_{\bar{d}\in D^g}\left(\alpha \wedge \beta \wedge \bigwedge_{j=1}^{b} \delta_j \wedge \bigwedge_{j=1}^{g} \omega_{\bar{d},j}(\bar{y})\right).$$

9. Suppose $\varphi(x_1, \ldots, x_k)$ is $(\exists x_{k+1})\varphi'(x_1, \ldots, x_k, x_{k+1})$. By the inductive hypothesis, there is a ψ' equivalent to φ' of the form

$$\bigvee_{i=1}^{m}\left(\alpha_i' \wedge \beta_i' \wedge \bigwedge_{j=1}^{k+1} \omega_{i,j}'\right).$$

Then φ is equivalent to

$$\bigvee_{i=1}^{m}(\exists x_{k+1})\left(\alpha_i' \wedge \beta_i' \wedge \bigwedge_{j=1}^{k+1} \omega_{i,j}'\right).$$

This is then equivalent to

$$\bigvee_{i=1}^{m}\bigvee_{p=1}^{k}\left(\alpha'_{i,p}\wedge\beta'_{i,p}\wedge\bigwedge_{j=1}^{k}\omega'_{i,j}\right) \tag{3.1}$$

$$\vee\bigvee_{i=1}^{m}(\exists x_{k+1})\left(\bigwedge_{p=1}^{k}(x_p\neq x_{k+1})\wedge\alpha'_i\wedge\beta'_i\wedge\bigwedge_{j=1}^{k+1}\omega'_{i,j}\right). \tag{3.2}$$

Here $\alpha'_{i,p}$ is false if $\omega'_{i,p}\not\equiv\omega'_{i,k+1}$, otherwise it equals the formula that is obtained from α'_i by replacing each occurrence of the variable x_{k+1} by x_p.

The subformula (3.1) is already in the right form. For each disjunct i of (3.2),

(a) if $\omega'_{i,k+1}$ is $x_{k+1}=\mathbf{c}$ then define α_i as

$$(\exists x_{k+1})(\bigwedge_{p=1}^{k}(x_p\neq x_{k+1})\wedge x_{k+1}=\mathbf{c}\wedge\alpha'_i),$$

β_i as

$$(\exists x_{k+1})(\bigwedge_{p=1}^{k}(x_p\neq x_{k+1})\wedge x_{k+1}=\mathbf{c}\wedge\beta'_i),$$

and for $j=1,\ldots,k$, define $\omega_{i,j}$ as $\omega'_{i,j}$.

(b) if $\omega'_{i,k+1}$ is $P_1(x_{k+1})$ then define α_i as

$$(\exists x_{k+1})(\bigwedge_{p=1}^{k}(x_p\neq x_{k+1})\wedge P_1(x_{k+1})\wedge\alpha'_i),$$

β_i as

$$\begin{aligned}(\exists x_{k+1})(&\bigwedge_{p=1}^{k}(x_p\neq x_{k+1})\\ &\wedge\neg P_2(x_{k+1})\wedge x_{k+1}\neq\mathbf{0}\wedge x_{k+1}\neq\mathbf{1}\wedge x_{k+1}\neq\mathbf{2}\wedge\beta'_i),\end{aligned}$$

(the correctness follows from Lemma 3.41(ii)), and for $j=1,\ldots,k$, define $\omega_{i,j}$ as $\omega'_{i,j}$.

(c) if $\omega'_{i,k+1}=P_2(x_{k+1})$ then define α_i as

$$\begin{aligned}(\exists x_{k+1})(&\bigwedge_{p=1}^{k}(x_p\neq x_{k+1})\\ &\wedge\neg P_1(x_{k+1})\wedge x_{k+1}\neq\mathbf{0}\wedge x_{k+1}\neq\mathbf{1}\wedge x_{k+1}\neq\mathbf{2}\wedge\alpha'_i),\end{aligned}$$

(the correctness follows from Lemma 3.41(i)), β_i as

$$(\exists x_{k+1})(\bigwedge_{p=1}^{k} x_p \neq x_{k+1} \wedge P_2(x_{k+1}) \wedge \beta'_i),$$

and for $j = 1, \ldots, k$, define $\omega_{i,j}$ as $\omega'_{i,j}$. ∎

In the next lemma we prove that if an FO-formula can only speak about c_1 (respectively c_2) then this formula in general cannot distinguish between $\xi(\mathcal{N}_1(n_1, n_1))$ and $\xi(\mathcal{N}_1(n_1, n_2))$ (respectively $\xi(\mathcal{N}_2(n_1, n_1))$ and $\xi(\mathcal{N}_2(n_1, n_2))$), where $n_1 \neq n_2$.

Lemma 3.44 *Let $\xi(x_1, \ldots, x_\ell)$ be a PFP-formula over the vocabulary $\tau_<$. Suppose ξ contains only m distinct variables.*

(i) Let ψ be an FO-sentence over the vocabulary $\tau_{c,\ell}$ that contains only m' distinct variables and that does not contain the relation symbol c_2. Let $n_1 \geq m + m'$. Then for all $n_2, n'_2 \geq m + m'$,

$$\mathbf{t}(\xi, n_1, n_2) \models \psi \quad \Leftrightarrow \quad \mathbf{t}(\xi, n_1, n'_2) \models \psi.$$

(ii) Let ψ be an FO-sentence over the vocabulary $\tau_{c,\ell}$ that contains only m' distinct variables and that does not contain the relation symbol c_1. Let $n_2 \geq m + m'$. Then for all $n_1, n'_1 \geq m + m'$,

$$\mathbf{t}(\xi, n_1, n_2) \models \psi \quad \Leftrightarrow \quad \mathbf{t}(\xi, n'_1, n_2) \models \psi.$$

Proof. We only prove (i), (ii) is symmetric. We can assume w.l.o.g. that ξ and ψ have no variables in common. Let $t(\xi, n_1, n_2)^{\widehat{c_2}}$ be the structure $t(\xi, n_1, n_2)$ restricted to **0**, **1**, **2** and c_1. Since ψ does not speak about c_2, it suffices to prove that for all $n_1, n_2, n'_2 \geq m + m'$:

$$t(\xi, n_1, n_2)^{\widehat{c_2}} \models \psi \quad \Leftrightarrow \quad t(\xi, n_1, n'_2)^{\widehat{c_2}} \models \psi.$$

Suppose there exist $n_2, n'_2 \geq m + m'$ such that

$$t(\xi, n_1, n_2)^{\widehat{c_2}} \models \psi \quad \text{and} \quad t(\xi, n_1, n'_2)^{\widehat{c_2}} \not\models \psi.$$

Let ψ' be the formula obtained from ψ by replacing each atomic subformula $c_1(z_1, \ldots, z_\ell)$ by $\xi(z_1, \ldots, z_\ell)$. Hence, $\mathcal{N}_1(n_1, n_2) \models \psi'$ and $\mathcal{N}_1(n_1, n'_2) \not\models \psi'$. Thus, there exists a sentence with $m + m'$ variables that distinguishes between $\mathcal{N}_1(n_1, n_2)$ and $\mathcal{N}_1(n_1, n'_2)$. However, for $n_1, n_2, n'_2 \geq m + m'$, using pebble games [KV92, EF95] it is easy to show that $\mathcal{N}_1(n_1, n_2)$ and $\mathcal{N}_1(n_1, n'_2)$ are indistinguishable in PFP with $m + m'$ variables. This leads to the desired contradiction. ∎

By putting all the pieces together we obtain the following theorem:

Theorem 3.45 *The query* `equal_subtree` *is not expressible by a synthesized RAG.*

Proof. Towards a contradiction suppose `equal_subtree` is expressible by a synthesized RAG. Suppose $\mathcal{R}$ has attributes $a_1, \ldots, a_k$ for the grammar symbol L, and attributes *result*, $b_1, \ldots, b_\ell$ for the start symbol U. W.l.o.g., we can assume that a_1 is a set-valued attribute that contains for each node all its descendants: for production $L \to \ell$ define $a_1(0) := \{\mathbf{0}, \mathbf{1}\}$, and for production $L \to L$ define $a_1(0) := \{\mathbf{0}\} \cup a_1(1)$.

We first show that $\mathcal{R}$ is equivalent to a simple RAG $\mathcal{R}'$. Let $r_c = k + 1 + \max\{r_{a_1}, \ldots, r_{a_k}\}$. Assume, w.l.o.g., that none of the variables $y_1, \ldots, y_{r_c}$ occurs in a semantic rule of $\mathcal{R}$. For $i = 1, \ldots, k$, let $\gamma_i(y_1, \ldots, y_{k+1})$ be the formula

$$\gamma_i(y_1, \ldots, y_{k+1}) \ := \ y_i = y_{k+1} \wedge \bigwedge \{y_j \neq y_{k+1} \mid j \in \{1, \ldots, k\} \wedge j \neq i\}.$$

For $i = 1, \ldots, k$, let $a_i(0) := \varphi_i$ be the rule in the context $(L \to f, a_i, 0)$. In $\mathcal{R}'$, define c in context $(L \to f, c, 0)$ as

$$c(0) := \{(y_1, \ldots, y_{r_c}) \mid \bigvee_{i=1}^{k} \gamma_i(y_1, \ldots, y_{k+1}) \wedge \varphi_i(y_{k+2}, \ldots, y_{k+1+r_{a_i}})\}.$$

For $i = 1, \ldots, k$, let $a_i(0) := \psi_i$ be the rule in context $(L \to L, a_i, 0)$. In $\mathcal{R}'$, define c in context $(L \to L, c, 0)$ as

$$c(0) := \{(y_1, \ldots, y_{r_c}) \mid \bigvee_{i=1}^{k} \gamma_i(y_1, \ldots, y_{k+1}) \wedge \psi'_i(y_{k+2}, \ldots, y_{k+1+r_{a_i}})\},$$

where ψ'_i is obtained from ψ_i by replacing each occurrence of $a_i(1)(\bar{x})$ by

$$(\exists \bar{y})(\exists \bar{z})(\gamma_i(\bar{y}) \wedge c(1)(\bar{y}, \bar{x}, \bar{z})),$$

where $\bar{z}$ and $\bar{x}$ have no variables in common.

Finally, let $result(0) := \sigma$ be the rule in context $(U \to LL, result, 0)$, and let for $i = 1, \ldots, \ell$, $b_i(0) := \sigma_i$ be the rule in context $(U \to LL, b_i, 0)$. Assume, w.l.o.g., that σ, $\sigma_1, \ldots, \sigma_\ell$ have no variables in common. Let σ' be the formula obtained from σ by replacing each occurrence of $b_i(0)(\bar{y})$ by $\sigma_i(\bar{y})$. Then define *result* in $\mathcal{R}'$ by the formula obtained from σ' by replacing each occurrence of $a_i(j)(\bar{x})$ by $(\exists \bar{y})(\exists \bar{z})(\gamma_i(\bar{y}) \wedge c(j)(\bar{y}, \bar{x}, \bar{z}))$, where $\bar{z}$ and $\bar{x}$ have no variables in common.

It follows that for any tree $\mathbf{t}$ with root $\mathbf{r}$, $\mathcal{R}(\mathbf{t})(result(\mathbf{r}))$ is true if and only if $\mathcal{R}'(\mathbf{t})(result(\mathbf{r}))$ is true. Hence, if $\mathcal{R}$ expresses `equal_subtree` then so does $\mathcal{R}'$. We will now show that $\mathcal{R}'$ cannot express `equal_subtree`.

According to Lemma 3.40, there exists a PFP-formula $\xi(x_1, \ldots, x_{r_c})$ such that for all $n_1, n_2 > 1$

$$\mathcal{R}'(\mathbf{t}(n_1, n_2))(c(\mathbf{n}_1)) = \xi(\mathcal{N}_1(n_1, n_2)),$$

and

$$\mathcal{R}'(\mathbf{t}(n_1, n_2))(c(\mathbf{n}_2)) = \xi(\mathcal{N}_2(n_1, n_2)),$$

where $\mathbf{n}_1$ is the first child and $\mathbf{n}_2$ is the second child of the root. Let ψ be the sentence that defines *result* in $\mathcal{R}'$. Then for all $n_1, n_2 > 1$,

$$\mathcal{R}'(\mathbf{t}(n_1, n_2))(result(\mathbf{r})) \text{ is true} \quad \Leftrightarrow \quad \mathbf{t}(\xi, n_1, n_2) \models \psi', \tag{3.3}$$

where ψ' is obtained from ψ by replacing each occurrence of $c(1)(\bar{x})$ by $c_1(\bar{x})$ and each occurrence of $c(2)(\bar{x})$ by $c_2(\bar{x})$, and where $\mathbf{r}$ is the root of $\mathbf{t}(\xi, n_1, n_2)$. Now, define $P_1(x)$ as

$$P_1(x) := (\exists \bar{y})(\exists \bar{z})(\gamma_1(\bar{y}) \wedge c_1(\bar{y}, x, \bar{z}) \wedge x \neq \mathbf{0} \wedge x \neq \mathbf{1} \wedge x \neq \mathbf{2}),$$

and $P_2(x)$ as

$$P_2(x) := (\exists \bar{y})(\exists \bar{z})(\gamma_1(\bar{y}) \wedge c_2(\bar{y}, x, \bar{z}) \wedge x \neq \mathbf{0} \wedge x \neq \mathbf{1} \wedge x \neq \mathbf{2}).$$

Then for all $n_1, n_2 > 1$,

$$\mathbf{t}(\xi, n_1, n_2) \models P_1[\mathbf{n}] \quad \Leftrightarrow \quad \mathbf{n} \in \{1, \ldots, n_1 - 1\},$$

and

$$\mathbf{t}(\xi, n_1, n_2) \models P_2[\mathbf{n}] \quad \Leftrightarrow \quad \mathbf{n} \in \{n_1 + 1, \ldots, n_1 + n_2 - 1\}.$$

Hence, by Lemma 3.43, ψ' is equivalent to a sentence of the form $\bigvee_{i=1}^{n} \alpha_i \wedge \beta_i$, where the α_i's are sentences that do not contain c_2, and the β_i's are sentences that do not contain c_1 (since there are no free variables, there are no ω's). W.l.o.g., we can assume that ξ and $\bigvee_{i=1}^{n} \alpha_i \wedge \beta_i$ have no variables in common. Let m be the number of variables in ξ, and let m' be the number of variables in $\bigvee_{i=1}^{n} \alpha_i \wedge \beta_i$. By a simple counting argument, there have to exist $n_1' > n_1 \geq m + m'$ such that for all $i = 1, \ldots, n$:

$$\mathbf{t}(\xi, n_1, m + m') \models \alpha_i \quad \Leftrightarrow \quad \mathbf{t}(\xi, n_1', m + m') \models \alpha_i.$$

Hence, by applying Lemma 3.44(i) twice, for $i = 1, \ldots, n$:

$$\mathbf{t}(\xi, n_1, n_1) \models \alpha_i \quad \Leftrightarrow \quad \mathbf{t}(\xi, n_1', n_1) \models \alpha_i.$$

From Lemma 3.44(ii), it follows that for $i = 1, \ldots, n$:

$$\mathbf{t}(\xi, n_1, n_1) \models \beta_i \quad \Leftrightarrow \quad \mathbf{t}(\xi, n_1', n_1) \models \beta_i.$$

Hence,

$$\mathbf{t}(\xi, n_1, n_1) \models \bigvee_{i=1}^{n} \alpha_i \wedge \beta_i \quad \Leftrightarrow \quad \mathbf{t}(\xi, n_1', n_1) \models \bigvee_{i=1}^{n} \alpha_i \wedge \beta_i.$$

But then by (3.3), we have that

$$\mathcal{R}'(\mathbf{t}(n_1, n_1))(\mathit{result}(\mathbf{r})) \text{ is true} \quad \Leftrightarrow \quad \mathcal{R}'(\mathbf{t}(n_1, n_1'))(\mathit{result}(\mathbf{r})) \text{ is true},$$

and $n_1 \neq n_1'$. Hence, $\mathcal{R}'$ does not express `equal_subtree`. ■

3.3.5 RAGs versus MSO

We have characterized BAGs as the unary queries definable in MSO (Theorem 3.24), and RAGs as the queries (of arbitrary arity) definable in PFP-LIN (Theorem 3.32). It remains to compare these two formalisms with each other. We will show that synthesized RAGs are actually strictly more powerful than MSO.

Theorem 3.46 *Every k-ary query definable in MSO is expressible by a RAG using only synthesized attributes.*

Proof. Consider an MSO-formula $\varphi(z_1, \dots, z_k)$. For any tree $\mathbf{t}$ and nodes $\mathbf{n}_1, \dots, \mathbf{n}_k$ of $\mathbf{t}$, we can view the tuple $(\mathbf{t}, \mathbf{n}_1, \dots, \mathbf{n}_k)$ as a labeling of $\mathbf{t}$ with elements of $\{0,1\}^k$ by labeling a node $\mathbf{n}$ with $u_1 \dots u_k$ such that for $i = 1, \dots, k$,

$$u_i = \begin{cases} 1 & \text{if } \mathbf{n} = \mathbf{n}_i, \\ 0 & \text{otherwise.} \end{cases}$$

So $(\mathbf{t}, \mathbf{n}_1, \dots, \mathbf{n}_k)$ is a Σ^k-tree where $\Sigma = N \cup T$ and $\Sigma^k = \{\sigma\bar{u} \mid \sigma \in \Sigma, \bar{u} \in \{0,1\}^k\}$.

It is easy to write an MSO sentence ψ over the vocabulary induced by Σ^k such that for every derivation tree $\mathbf{t}$ and nodes $\mathbf{n}_1, \dots, \mathbf{n}_k$ of $\mathbf{t}$:

$$(\mathbf{t}, \mathbf{n}_1, \dots, \mathbf{n}_k) \models \psi \quad \Leftrightarrow \quad \mathbf{t} \models \varphi[\mathbf{n}_1, \dots, \mathbf{n}_k].$$

Indeed, for $i = 1, \dots, k$, define J_i as the set of grammar symbols for which the i-th component is 1, i.e.,

$$J_i := \{X\bar{u} \mid X \in N \cup T, \bar{u} \in \{0,1\}^k \text{ and } u_i = 1\}.$$

Then ψ is defined as

$$(\exists z_1) \dots (\exists z_k) \left(\left(\left(\bigvee_{X\bar{u} \in J_1} O_{X\bar{u}}(z_1) \right) \wedge \dots \wedge \left(\bigvee_{X\bar{u} \in J_k} O_{X\bar{u}}(z_k) \right) \right) \rightarrow \varphi'(z_1, \dots, z_k) \right),$$

where φ' is the formula obtained by replacing each atomic formula $O_X(z)$ in φ by $\bigvee_{\bar{u} \in \{0,1\}^k} O_{X\bar{u}}(z)$.

By Theorem 2.13 there exists a deterministic bottom-up tree automaton $B = (Q, \Sigma^k, \delta, F)$ that accepts only those $(\mathbf{t}, \mathbf{n}_1, \dots, \mathbf{n}_k)$ such that $(\mathbf{t}, \mathbf{n}_1, \dots, \mathbf{n}_k) \models \psi$, where $\mathbf{t}$ is a derivation tree. Hence, the theorem is proved if we can construct a RAG $\mathcal{R}$ which, on each derivation tree $\mathbf{t}$, simulates B in parallel on all possible labelings of $\mathbf{t}$, returning those labelings $(\mathbf{n}_1, \dots, \mathbf{n}_k)$ such that $(\mathbf{t}, \mathbf{n}_1, \dots, \mathbf{n}_k)$ is accepted by B. To this end, we use a k-ary relation-valued synthesized attributes q for each state q of B. The semantic rules are such that for each node $\mathbf{n}$, $(\mathbf{n}_1, \dots, \mathbf{n}_k) \in q(\mathbf{n})$ iff B assumes state q on node $\mathbf{n}$ in its execution on $(\mathbf{t}, \mathbf{n}_1, \dots, \mathbf{n}_k)$. The attribute *result* at the root is then defined as $\bigcup_{q \in F} q$, where F is the set of final states of B.

The formula $\text{match}_{\bar{u}}(z_1, \ldots, z_k, y)$ defines the labelings $(z_1, \ldots, z_k)$ of the tree $\mathbf{t}$ such that node y is labeled with $\bar{u}$:

$$\text{match}_{\bar{u}}(z_1, \ldots, z_k, y) := \bigwedge_{u_i=1} z_i = y \wedge \bigwedge_{u_i=0} z_i \neq y.$$

Let $p = X_0 \to X_1 \ldots X_n$ be a production of G. Let $T(p) := \{i \in \{1, \ldots, n\} \mid X_i \text{ is a terminal}\}$, and $N(p) := \{1, \ldots, n\} - T(p)$. For each $q \in Q$, add the semantic rule

$$\begin{aligned} q(0) \; := \bigvee \{ & \text{match}_{\bar{u}_0}(\bar{z}, \mathbf{0}) \wedge \bigwedge_{i \in N(p)} q_i(\mathbf{i})(\bar{z}) \wedge \bigwedge_{i \in T(p)} \sigma_i^{\bar{u}_i, q_i}(\bar{z}, \mathbf{i}) \mid \\ & \bar{u}_0, \ldots, \bar{u}_n \in \{0,1\}^k, q_1, \ldots, q_n \in Q, \text{ and } \delta(q_1, \ldots, q_n, X_0\bar{u}) = q\}, \end{aligned}$$

where $\sigma_i^{\bar{u},q}(\bar{z}, y)$ is the formula $\text{match}_{\bar{u}}(\bar{z}, y)$ when $\delta(X_i\bar{u}) = q$ and is false otherwise.

The correctness of this construction now follows from the following lemma, which is easily proven by induction on the height of $\mathbf{n}$.

Lemma 3.47 *Let $\mathbf{t}$ be a tree. For each internal node $\mathbf{n}$ of $\mathbf{t}$, $(\mathbf{m}_1, \ldots, \mathbf{m}_k) \in q(\mathbf{n})$ iff B assumes state q on node $\mathbf{n}$ in its execution on the labeled tree $(\mathbf{t}, \mathbf{m}_1, \ldots, \mathbf{m}_k)$.*

This concludes the proof. ■

While Theorem 3.46 states that synthesized attributes are enough for a RAG to express all of MSO, this is not the case for BAGs. Indeed as explained at the end of Section 3.2.2, BAGs with only synthesized attributes are weaker than MSO.

We finally show that synthesized RAGs are strictly more powerful than MSO. However, this only holds under the assumption that the underlying grammar can generate an infinite number of derivation trees.

Definition 3.48 A grammar is *unbounded* if the number of its derivation trees is infinite.

Clearly, if the grammar is not unbounded, i.e., the number of derivation trees is finite, then RAGs and MSO are equally powerful because a query simply reduces to a case analysis. We next show:

Theorem 3.49 *Over any unbounded grammar, synthesized RAGs can express Boolean queries not definable in MSO.*

Proof. Consider an unbounded grammar G. There exists a sequence of productions $\bar{p} = p_1, \ldots, p_m$, a sequence of numbers $\bar{k} = k_1, \ldots, k_m$, and a non-terminal X, such that (i) X is the left-hand side of p_1 and occurs in position k_m of the right-hand side of p_m; (ii) for $i = 1, \ldots, m-1$ the symbol in position k_i of the right-hand side of p_i equals the non-terminal on the left-hand side of p_{i+1}; (iii) the left-hand sides of $p_1, \ldots, p_m$ are mutually distinct; and (iv) X is reachable from the start symbol.

Consider a derivation tree $\mathbf{t}$ of G. We say that a node $\mathbf{n}_1$ is an *occurrence* of $(\bar{p}, \bar{k})$ in $\mathbf{t}$ if there exists a sequence of nodes $\mathbf{n}_2, \dots, \mathbf{n}_m, \mathbf{n}_{m+1}$ such that for $i = 1, \dots, m$, $\mathbf{n}_i$ is derived with p_i and $\mathbf{n}_{i+1}$ is the k_i-th child of $\mathbf{n}_i$. We say that $\mathbf{n}_{m+1}$ is the *tail* of the occurrence $\mathbf{n}_1$. Note that $\mathbf{n}_{m+1}$ is labeled with X. We call a sequence of nodes $\mathbf{n}_1, \dots, \mathbf{n}_s$ a *chain of occurrences* of $(\bar{p}, \bar{k})$ if for each $i = 1, \dots, s-1$, $\mathbf{n}_i$ is an occurrence, $\mathbf{n}_{i+1}$ is the tail of the occurrence $\mathbf{n}_i$, and $\mathbf{n}_s$ is not derived with p_1. The length of the chain of occurrences $\mathbf{n}_1, \dots, \mathbf{n}_s$ is s.

Let $\mathcal{Q}$ be the Boolean query defined as follows: on every derivation tree $\mathbf{t}$, $\mathcal{Q}(\mathbf{t})$ is true if there is a chain of occurrences starting on the first X-labeled node in the preorder traversal of the tree and its length is a power of two. Note that $\mathcal{Q}$ is true on any tree where there is no X-labeled node.

The query $\mathcal{Q}$ is not definable in MSO. We now show that $\mathcal{Q}$ is not expressible in MSO. Let φ be an MSO-sentence. By Theorem 2.13 there exists a deterministic bottom-up tree automaton $B = (Q, N \cup T, \delta, F)$ accepting precisely the trees satisfying φ. Consider a tree $\mathbf{t}$ such that the length, which we denote by c, of the chain of occurrences of $(\bar{p}, \bar{k})$ starting in the first X-labeled node in the preorder traversal of the tree is a power of two and is bigger than $|Q| + 1$. There have to be two nodes $\mathbf{n}$ and $\mathbf{n}'$ of $\mathbf{t}$, such that $\mathbf{n}'$ is a descendant of $\mathbf{n}$, both are occurrences of $(\bar{p}, \bar{k})$ in the chain, and $\delta^*(\mathbf{t}_\mathbf{n}) = \delta^*(\mathbf{t}_{\mathbf{n}'})$. Let $\mathbf{n}$ be the o_1-th occurrence and $\mathbf{n}'$ be the o_2-th occurrence in the chain. Let $\mathbf{t}'$ be the tree obtained from $\mathbf{t}$ by replacing the subtree $\mathbf{t}(\mathbf{n}')$ by the subtree $\mathbf{t}(\mathbf{n})$. Then the chain of occurrences of $(\bar{p}, \bar{k})$ starting in the first X-labeled node in the preorder traversal of $\mathbf{t}'$ has length $c + o_2 - o_1$. This is not a power of two because $c < c + o_2 - o_1 < 2c$. However, $\delta^*(\mathbf{t}) = \delta^*(\mathbf{t}')$, and thus $\mathbf{t}' \models \varphi$ if and only if $\mathbf{t} \models \varphi$. Hence, φ does not define $\mathcal{Q}$.

The query $\mathcal{Q}$ is expressible by a synthesized RAG. The RAG uses the following synthesized attributes for all non-terminals:

1. X is a Boolean attribute: $X(\mathbf{n})$ is true if there is a node labeled X among the descendants of $\mathbf{n}$ (note that X is a non-terminal);

2. *chain* is a Boolean attribute:

   (a) $\mathit{chain}(\mathbf{n})$ is false if there is no X-labeled node among the descendants of $\mathbf{n}$, or if there is no chain of occurrences starting on the the first X-labeled node in the preorder traversal of the subtree with root $\mathbf{n}$;

   (b) $\mathit{chain}(\mathbf{n})$ is true if the length of the chain of occurrences of $(\bar{p}, \bar{k})$ starting at the first X-labeled node in the preorder traversal of the subtree with root $\mathbf{n}$ is a power of two. Note that if this node is not derived by p_1, then the length of the chain of occurrences starting at that node is 1, which is a power of two.

3. D is a set-valued attribute: $D(\mathbf{n})$ contains $\mathbf{n}$ and all descendants of $\mathbf{n}$.

4. $<$ is a binary attribute: $<(\mathbf{n})$ is a total order on $D(\mathbf{n})$.

5. *occ* is a Boolean attribute: *occ* is true if $\mathbf{n}$ is derived with p_i, for some $i \in \{1, \ldots, m\}$, and there exist nodes $\mathbf{n}_{i+1}, \ldots, \mathbf{n}_{m+1}$, such that a chain of occurrences starts at $\mathbf{n}_{m+1}$, and for $j = i+1, \ldots, m$, $\mathbf{n}_j$ is derived with p_j, and $\mathbf{n}_{j+1}$ is the k_j-th child of $\mathbf{n}_j$.

6. *is-*p_1 is a Boolean attribute: *is-*$p_1(\mathbf{n})$ is true if $\mathbf{n}$ is derived with p_1.

7. a is a set-valued attribute: $a(\mathbf{n})$ is a subset of $D(\mathbf{n})$.

   (a) If *occ* is false, then $a(\mathbf{n})$ is empty;

   (b) If *occ* is true and $\mathbf{n}$ is not derived by p_1, then the nodes in $a(\mathbf{n})$ encode, w.r.t. $<(\mathbf{n})$, in binary the length of the chain of occurrences of $(\bar{p}, \bar{k})$ starting at $\mathbf{n}_{m+1}$, where $\mathbf{n}_{m+1}$ is as defined in 5: if $a(\mathbf{n})$ contains the nodes $\mathbf{n}_1, \ldots,$ $\mathbf{n}_r$, and these occur respectively in the i_1-th, $\ldots$, i_k-th position in the ordering $<(\mathbf{n})$, then $a(\mathbf{n})$ encodes the number $\Sigma_{p=1}^{r} 2^{i_p - 1}$.

   (c) If *occ* is true and $\mathbf{n}$ is derived by p_1, then the nodes in $a(\mathbf{n})$ encode, w.r.t. $<(\mathbf{n})$, in binary the length of the chain of occurrences of $(\bar{p}, \bar{k})$ starting at $\mathbf{n}$.

The RAG is now defined as follows:

1. Consider the production $p = X_0 \to X_1 \ldots X_n$ not in $\bar{p}$. Define $T(p) = \{i \in \{1, \ldots, n\} \mid X_i \text{ is a terminal}\}$. We write $i \notin T(p)$ as a shorthand for $i \in \{1, \ldots, n\} - T(p)$. Define

$$X(0) \;:=\; \begin{cases} \text{true} & \text{if } X_0 = X; \\ \bigvee_{i \notin T(p)} X(i) & \text{otherwise;} \end{cases}$$

   and

$$\mathit{chain}(0) \;:=\; \begin{cases} \text{true} & \text{if } X_0 = X; \\ \gamma & \text{otherwise,} \end{cases}$$

   where γ is an FO-sentence whose truth value equals that of $\mathit{chain}(i)$, where i is the smallest such that $X(i)$ is true; if such an i does not exists, γ is false. For $i = 1, \ldots, n$, let $E(i)$ be $D(i)$ if $i \notin T(p)$, and $\{(\mathbf{i})\}$ when $i \in T(p)$. Then define

$$D(0) \;:=\; \bigcup_{i=1}^{n} E(i) \cup \{(\mathbf{0})\};$$

$$\begin{aligned} <(0) \;:=\; & \bigcup_{i \notin T(p)} <(i) \cup (D(0) \times \{(\mathbf{0})\}) \cup \bigcup_{i \in T(p)} \{(\mathbf{i}, \mathbf{i})\} \\ & \cup \bigcup \{E(i) \times E(j) \mid i, j \in \{1, \ldots, n\} \wedge i < j\}; \end{aligned}$$

$$\begin{aligned} occ(0) &:= \text{false}; \\ is\text{-}p_1(0) &:= \text{false}; \end{aligned}$$

and

$$a(0) := \emptyset.$$

If $X_0 = U$ then define

$$result(0) := X(0) \to chain(0).$$

2. For $p_1 = X_0 \to X_1 \dots X_{n_1}$, define

$$X(0) := \text{true};$$

and

$$chain(0) := occ(0) \wedge (\exists x)\big(a(0)(x) \wedge (\forall y)(a(0)(y) \to x = y)\big).$$

The attribute *chain* becomes true if $occ(0)$ is true and if $a(0)$ contains exactly one element, i.e., $a(0)$ encodes a number which is a power of two. Define,

$$D(0) := \bigcup_{i=1}^{n_1} E(i) \cup \{(\mathbf{0})\};$$

$$\begin{aligned} <(0) := & \bigcup_{i \notin T(p_1)} <(i) \cup \bigcup_{i \in \{1,\dots,n_1\}-\{k_1\}} (D(k_1) \times E(i)) \\ & \cup (D(0) \times \{(\mathbf{0})\}) \cup \bigcup_{i \in T(p_1)} \{(\mathbf{i}, \mathbf{i})\} \\ & \cup \bigcup \{E(i) \times E(i') \mid \\ & \qquad i, i' \in \{1, \dots, n_1\},\ i < i',\ i \neq k_1 \wedge i' \neq k_1\}; \end{aligned}$$

The reason for this definition is that, in order to correctly represent the number in $a(k_j)$, we have to make sure that all elements not in $D(k_j)$ come after the elements in $D(k_j)$ in the ordering $<(0)$. Finally, define

$$\begin{aligned} occ(0) &:= occ(k_1); \\ is\text{-}p_1(0) &:= \text{true}; \end{aligned}$$

and

$$a(0) := occ(0) \wedge \varphi(z, a(k_1), <(0)),$$

where $\varphi(z, Z, <)$ is the formula

$$\begin{aligned} & (\exists z')\big(Z(z') \wedge \neg Z(\mathrm{Succ}(z')) \wedge (\forall z'')(z'' < z' \to Z(z'')) \\ & \qquad\qquad \wedge \big(z = \mathrm{Succ}(z') \vee (z' < z \wedge Z(z))\big)\big) \\ & \vee (\exists z')\big(\mathrm{First}(z') \wedge \neg Z(z') \wedge (z = z' \vee Z(z))\big), \end{aligned}$$

with Succ the successor function, and First is the first element in the ordering $<$. This formula augments the number in $a(k_1)$ with 1.

3. For $j = 2, \ldots, m-1$, define for $p_j = X_0 \to X_1 \ldots X_{n_j}$,

$$X(0) \ := \ \bigvee_{i \notin T(p_j)} X(i)$$

and

$$\mathit{chain}(0) \ := \ \gamma,$$

where γ is defined as in (1). Further, define

$$D(0) \ := \ \bigcup_{i=1}^{n_j} E(i) \cup \{(\mathbf{0})\};$$

$$\begin{aligned} <(0) \ := \ & \bigcup_{i \notin T(p_j)} <(i) \cup \bigcup_{i \in \{1,\ldots,n_j\}-\{k_j\}} (D(k_j) \times E(i)) \\ & \cup (D(0) \times \{(\mathbf{0})\}) \cup \bigcup_{i \in T(p_j)} \{(\mathbf{i}, \mathbf{i})\} \\ & \cup \bigcup \{E(i) \times E(i') \mid \\ & \qquad i, i' \in \{1, \ldots, n_j\},\ i < i',\ i \neq k_j \wedge i' \neq k_j\}; \end{aligned}$$

$$\begin{aligned} \mathit{occ}(0) \ &:= \ \mathit{occ}(k_j); \\ \mathit{is}\text{-}p_1(0) \ &:= \ \text{false}; \end{aligned}$$

and

$$a(0) \ := \ \mathit{occ}(0) \wedge a(k_j)(z).$$

4. For $p_m = X_0 \to X_1 \ldots X_{n_m}$, define

$$X(0) \ := \ \text{true};$$

and

$$\mathit{chain}(0) \ := \ \gamma.$$

Here γ is defined as in (1). Define,

$$D(0) \ := \ \bigcup_{i=1}^{n_m} E(i) \cup \{(\mathbf{0})\};$$

$$\begin{aligned} <(0) \ := \ & \bigcup_{i \notin T(p_1)} <(i) \cup \bigcup_{i \in \{1,\ldots,n_m\}-\{k_m\}} (D(k_m) \times E(i)) \\ & \cup (D(0) \times \{(\mathbf{0})\}) \cup \bigcup_{i \in T(p_1)} \{(\mathbf{i}, \mathbf{i})\} \\ & \cup \bigcup \{E(i) \times E(i') \mid \\ & \qquad i, i' \in \{1, \ldots, n_m\},\ i < i',\ i \neq k_m \wedge i' \neq k_m\}; \end{aligned}$$

$$\begin{aligned} occ(0) &:= \neg is\text{-}p_1(k_m) \vee occ(k_m); \\ is\text{-}p_1(0) &:= \text{false}; \end{aligned}$$

and define

$$a(0) := \sigma.$$

Here, σ is an FO-sentence that expresses the following: $a(0)$ contains the singleton consisting of the first element in $<(0)$ if $is\text{-}p_1(k_m)$ is false; $a(0)$ equals $a(k_m)$ if $occ(k_m)$ is true; and $a(0) = \emptyset$ otherwise. ■

As a corollary, we note:

Corollary 3.50 *RAGs can express more unary queries than BAGs.*

3.4 Relational attribute grammars

Relational attribute grammars[3] are a generalization of standard attribute grammars introduced by Courcelle and Deransart [CD88]. In relational attribute grammars, the semantic rules no longer specify functions, computing attributes in terms of other attributes, but rather relations among attributes. Also there is no longer a distinction between synthesized and inherited attributes, and the values of the attributes are no longer uniquely determined for every tree. We consider relational attribute grammars in the context of BAGs and RAGs, and discuss how they can express queries. We show that for BAGs this does not increase the expressive power, while in the case of RAGs the complexity classes NP, coNP and UP $\cap$ coUP are captured.

3.4.1 Relational BAGs

An attribute grammar vocabulary is now just a tuple (A, Att), where A is a finite set of attributes and Att is a function from A to the powerset of $N \cup T$. A *relational BAG* $\mathcal{B}$ assigns to each production, $p = X_0 \to X_1 \dots X_n$ a propositional formula φ_p over the set of propositional symbols

$$\{a(j) \mid j \in \{0, \dots, n\}, a \in \mathrm{Att}(X_j)\}.$$

A *valuation* of a derivation tree $\mathbf{t}$ is a mapping that assigns a truth value to each $a(\mathbf{n})$, where $a \in A$, $\mathbf{n}$ is a node of $\mathbf{t}$, and a is an attribute of the label of $\mathbf{n}$. Let $\mathbf{n}$ be a node of $\mathbf{t}$ of arity n derived by production p. Let φ_p be the formula associated to p. Then define $\Delta(\mathcal{B}, \mathbf{t}, \mathbf{n})$ as the formula obtained from φ_p by replacing each propositional symbol of the form $b(j)$ by the new propositional symbol $b(\mathbf{n}j)$. An arbitrary total

[3] Relational attribute grammars are not to be confused with the relation-valued standard attribute grammars (RAGs) of the previous section. In fact, in the present section, we will indeed consider relational versions of RAGs.

$U \to S$	$\neg x\_before(1)$
$S \to BS$	$(x\_before(2) \leftrightarrow is\_x(1) \vee x\_before(0))$
	$\wedge\, (even(0) \leftrightarrow \neg even(2))$
	$\wedge\, (result(0) \leftrightarrow even(0) \wedge x\_before(0))$
$S \to B$	$\neg even(0) \wedge \neg result(0)$
$B \to x$	$is\_x(0)$
$B \to y$	$\neg is\_x(0)$

Figure 3.8: Example of a relational BAG.

valuation v of $\mathbf{t}$ is said to *satisfy* $\mathcal{B}$ if $\Delta(\mathcal{B}, \mathbf{t}, \mathbf{n})$ is true under v, for every internal node $\mathbf{n}$.

A relational BAG can express unary queries in various ways. Let $\mathcal{Q}$ be a unary query and let $\mathcal{B}$ be a relational BAG. Designate among the attributes of A an attribute *result*.

(i) $\mathcal{Q}$ is expressed *existentially* by $\mathcal{B}$ iff for every $\mathbf{t}$

$$\mathcal{Q}(\mathbf{t}) = \{\mathbf{n} \mid \text{there exists a valuation } v \text{ of } \mathbf{t} \text{ that satisfies } \mathcal{B}, \text{ and } v(\mathit{result}(\mathbf{n})) \text{ is true}\};$$

(ii) $\mathcal{Q}$ is expressed *universally* by $\mathcal{B}$ iff for every $\mathbf{t}$

$$\mathcal{Q}(\mathbf{t}) = \{\mathbf{n} \mid \text{for every valuation } v \text{ of } \mathbf{t} \text{ that satisfies } \mathcal{B}, v(\mathit{result}(\mathbf{n})) \text{ is true}\};$$

(iii) $\mathcal{Q}$ is expressed *implicitly* by $\mathcal{B}$ iff for every $\mathbf{t}$ there exists exactly one valuation v of $\mathbf{t}$ that satisfies $\mathcal{B}$, and $\mathbf{n} \in \mathcal{Q}(\mathbf{t})$ iff $v(\mathit{result}(\mathbf{n}))$ is true.

We denote the class of unary queries existentially (respectively, universally and implicitly) expressible by relational BAGs by $\exists$-BAG (respectively, $\forall$-BAG and IBAG).

Example 3.51 In Figure 3.8 an example of a relational BAG is depicted. It expresses existentially, universally and implicitly the same query expressed by the BAG in Example 3.5. ■

The following theorem says that going from BAGs to relational BAGs does not increase the expressive power.

Theorem 3.52 *BAG* $=$ *$\exists$-BAG* $=$ *$\forall$-BAG* $=$ *IBAG.*

Proof. Clearly, by Lemma 3.9, BAG $\subseteq$ $\exists$-BAG, BAG $\subseteq$ $\forall$-BAG and BAG $\subseteq$ IBAG. By using Theorem 3.24, we then only have to prove that every query in $\exists$-BAG, $\forall$-BAG and IBAG is definable in MSO.

1. Let $\mathcal{Q}$ be a query that is existentially expressed by the relational BAG $\mathcal{B}$. Then $\mathcal{Q}$ is defined by the MSO-formula $\psi(x) :=$

$$(\exists Z_a)_{a\in A}\Big(\Big(\bigwedge_{p=X_0\to X_1\ldots X_n}(\forall x_0)\ldots(\forall x_n)(p(x_0,\ldots,x_n)\to\tilde{\varphi}_p)\Big) \wedge Z_{result}(x)\Big)$$

   where A is the set of attributes of $\mathcal{B}$, $p(x_0,\ldots,x_n)$ is the FO-formula that expresses that nodes $x_0,\ldots,x_n$ are derived by production p, and $\tilde{\varphi}_p$ is the formula obtained from φ_p by replacing each occurrence of $b(j)$ by $Z_b(x_j)$. Intuitively, the Z_a's define valuations that satisfy $\mathcal{B}$.

2. Let $\mathcal{Q}$ be a query that is universally expressed by the relational BAG $\mathcal{B}$. Then $\mathcal{Q}$ is defined by the MSO-formula $\psi(x) :=$

$$(\forall Z_a)_{a\in A}\Big(\Big(\bigwedge_{p=X_0\to X_1\ldots X_n}(\forall x_0)\ldots(\forall x_n)(p(x_0,\ldots,x_n)\to\tilde{\varphi}_p)\Big) \to Z_{\mathrm{result}}(x)\Big).$$

   Here $p(x_0,\ldots,x_n)$ and $\tilde{\varphi}_p$ are defined as in (1).

3. Let $\mathcal{Q}$ be a query that is implicitly expressed by the relational BAG $\mathcal{B}$. Then the MSO-formula that defines $\mathcal{Q}$ is the same as in (1). ■

3.4.2 Relational RAGs

Each attribute a has an associated arity r_a. A *relational RAG* $\mathcal{R}$ associates to each production $p = X_0 \to X_1 \ldots X_n$ an FO-sentence φ_p over the vocabulary

$$\bigcup_{j=0}^{n}\{a(j) \mid a \in \mathrm{Att}(X_j)\} \cup \{\mathbf{0},\mathbf{1},\ldots,\mathbf{n}\},$$

where for each $j = 0,\ldots,n$, $\mathbf{j}$ is a constant symbol and $a(j)$ is a relation symbol of arity r_a. A *valuation* of a derivation tree $\mathbf{t}$ is a mapping that assigns to each $a(\mathbf{n})$ an r_a-ary relation over the nodes of $\mathbf{t}$, where $a \in A$, $\mathbf{n}$ is a node of $\mathbf{t}$ and a is an attribute of the label of $\mathbf{n}$. Let $\mathbf{n}$ be a node of $\mathbf{t}$ of arity n derived by production p. Let φ_p be the FO-sentence associated to p. Then define $\Delta(\mathcal{R},\mathbf{t},\mathbf{n})$ as the FO-sentence obtained from φ_p by replacing each occurrence of the relation symbol $b(j)$ by the relation symbol $b(\mathbf{n}j)$ and by replacing each constant symbol $\mathbf{j}$ by the node $\mathbf{n}j$. A valuation v of $\mathbf{t}$ is said to *satisfy* $\mathcal{R}$ if $\Delta(\mathcal{R},\mathbf{t},\mathbf{n})$ evaluates to true for all $\mathbf{n}$ when each relation symbol $b(\mathbf{m})$ in $\Delta(\mathcal{R},\mathbf{t},\mathbf{n})$ is interpreted by v.

A relational RAG can express k-ary queries in various ways. Let $\mathcal{Q}$ be a k-query and let $\mathcal{R}$ be a relational RAG. Designate among the attributes of A a k-ary attribute *result*.

$$
\begin{array}{ll}
U \to LL & (\forall x)(\exists! y)(C(1)(x) \to (C(2)(y) \wedge R(0)(x,y))) \\
 & \wedge\, (\forall y)(\exists! x)(C(2)(y) \to (C(1)(x) \wedge R(0)(x,y))) \\
 & \wedge\, \mathit{result}(0) \\
L \to L & (\forall x)(C(0)(x) \leftrightarrow x = \mathbf{0} \vee C(1)(x)) \\
L \to f & (\forall x)(C(0)(x) \leftrightarrow x = \mathbf{0} \vee x = \mathbf{1})
\end{array}
$$

Figure 3.9: Example of a relational RAG.

(i) $\mathcal{Q}$ is expressed *existentially* by $\mathcal{R}$ iff for every $\mathbf{t}$

$$\mathcal{Q}(\mathbf{t}) = \{(\mathbf{n}_1, \ldots, \mathbf{n}_k) \mid \text{there exists a valuation } v \text{ of } \mathbf{t} \text{ satisfying } \mathcal{R} \text{ with } (\mathbf{n}_1, \ldots, \mathbf{n}_k) \in v(\mathit{result}(\mathrm{root}(\mathbf{t})))\}.$$

(ii) $\mathcal{Q}$ is expressed *universally* by $\mathcal{R}$ iff for every $\mathbf{t}$

$$\mathcal{Q}(\mathbf{t}) = \{(\mathbf{n}_1, \ldots, \mathbf{n}_k) \mid \text{for every valuation } v \text{ that satisfies } \mathcal{R}, (\mathbf{n}_1, \ldots, \mathbf{n}_k) \in v(\mathit{result}(\mathrm{root}(\mathbf{t})))\}$$

(iii) $\mathcal{Q}$ is expressed *implicitly* by $\mathcal{R}$ iff for every $\mathbf{t}$ there exists exactly one valuation v of $\mathbf{t}$ that satisfies $\mathcal{R}$, and $(\mathbf{n}_1, \ldots, \mathbf{n}_k) \in \mathcal{Q}(\mathbf{t})$ iff $(\mathbf{n}_1, \ldots, \mathbf{n}_k) \in v(\mathit{result}(\mathrm{root}(\mathbf{t})))$.

We denote the class of unary queries existentially (respectively, universally and implicitly) expressible by relational RAGs by $\exists$-RAG (respectively, $\forall$-RAG and IRAG).

Example 3.53 In Figure 3.9 an example of a relational RAG $\mathcal{R}$ is depicted. This RAG expresses existentially the Boolean query which is true for a tree if the number of nodes in its left subtree equals the number of nodes in its right subtree. Let $\mathbf{t}$ be a tree, $\mathbf{r}$ the root of $\mathbf{t}$, $\mathbf{r}_1$ the left child and let $\mathbf{r}_2$ be the right child of the root. For any valuation v of $\mathbf{t}$ that satisfies $\mathcal{R}$, $v(C(\mathbf{r}_1))$ contains the nodes of the left subtree, and $v(C(\mathbf{r}_2)$ contains the nodes of the right subtree. The sentence associated to the root can only be true under v if $v(R(\mathbf{r}))$ contains a bijection between $v(C(\mathbf{r}_1))$ and $v(C(\mathbf{r}_2))$. ■

Clearly, any query expressible by a RAG is in $\exists$-RAG, $\forall$-RAG and IRAG. Indeed, let $a_1(i_1) := \varphi_1, \ldots, a_n(i_n) := \varphi_n$ be all the semantic rules in a RAG $\mathcal{R}$ associated to a production p. In the corresponding relational RAG, we just replace these by the single rule

$$(\forall \bar{x})(a_1(i_1)(\bar{x}) \leftrightarrow \varphi_1(\bar{x})) \wedge \ldots \wedge (\forall \bar{x})(a_n(i_n)(\bar{x}) \leftrightarrow \varphi_n(\bar{x})).$$

This is indeed a correct translation for all three discussed semantics as, by Lemma 3.18, there is only one valuation for each tree that satisfies $\mathcal{R}$.

In the following theorem, we characterize the 3 classes of relational RAG queried in terms of the complexity classes NP and UP (and their complements). NP is well known; UP is the class of problems decidable by a polynomial time non-deterministic Turing machine that is unambiguous, i.e., that has at most one accepting computation for every input [Val76, Pap94].[4] We obtain the following:

Theorem 3.54 *1. $\exists$-RAG equals the class of queries in NP;*

2. $\forall$-RAG equals the class of queries in coNP; and

3. IRAG equals the class of queries in UP $\cap$ coUP.

Proof.

1. The containment $\exists$-RAG $\subseteq$ NP is clear. For the converse, we make use of Fagin's Theorem [Fag74, EF95], which states that the queries expressible in NP are exactly those that are definable in existential second-order logic (ESO). Every ESO-formula is of the form

$$(\exists Z_1)\dots(\exists Z_n)(\psi(\bar{x}, Z_1, \dots, Z_n)),$$

where the Z_i are relation variables and ψ is an FO-formula over the vocabulary expanded with the relation symbols $\{Z_1, \dots, Z_n\}$.

Consider the ESO-formula: $(\exists Z_1)\dots(\exists Z_n)\psi(\bar{x}, \bar{Z})$. Like in the proof of Theorem 3.32, we can construct a RAG that computes all the relations that make up a derivation tree viewed as a relational structure. Add to this RAG the rule for the start symbol

$$(\forall \bar{x})(\mathrm{result}(0)(\bar{x}) \leftrightarrow \psi'(\bar{x})),$$

where ψ' is obtained from ψ by replacing each $Z_i(\bar{y})$ by $Z_i(0)(\bar{y})$, and by replacing each relation of the vocabulary of the relational structure by its corresponding attribute. It then follows that this relational RAG expresses existentially the query defined by $(\exists Z_1)\dots(\exists Z_n)\psi(\bar{x}, \bar{Z})$.

2. To prove that $\forall$-RAG are the queries computable in coNP, we make use of the complement of Fagin's Theorem: coNP = Universal second-order logic. The proof is then analogous to (1).

3. Clearly, IRAG $\subseteq$ UP $\cap$ coUP.

Let $(\mathcal{Q}_1, \dots, \mathcal{Q}_n)$ be a sequence of queries. We say that the sequence

$$(\mathcal{Q}_1, \dots, \mathcal{Q}_n)$$

is *implicitly definable in FO* if there is an FO-sentence $\psi(Z_1, \dots, Z_n)$ over the vocabulary of derivation trees augmented with $\{Z_1, \dots, Z_n\}$ such that for every tree $\mathbf{t}$ the sequence $(\mathcal{Q}_1(\mathbf{t}), \dots, \mathcal{Q}_n(\mathbf{t}))$ is the only sequence of relations $(R_1, \dots, R_n)$ over $\mathbf{t}$ such that $\mathbf{t} \models \psi[R_1, \dots, R_n]$.

[4] A k-ary query $\mathcal{Q}$ belongs to a complexity class $\mathcal{C}$ if the decision problem $\{(\mathbf{t}, \mathbf{n}_1, \dots, \mathbf{n}_k) \mid (\mathbf{n}_1, \dots, \mathbf{n}_k) \in \mathcal{Q}(\mathbf{t})\}$ is in $\mathcal{C}$.

We write IMP(FO) to denote the collection of all queries $\mathcal{Q}$ such that $\mathcal{Q} = \mathcal{Q}_1$ for some sequence $(\mathcal{Q}_1, \ldots, \mathcal{Q}_n)$ of queries which is implicitly definable in FO.

Analogously to (1) it can be shown that every query in IMP(FO) is expressible by a RAG. Kolaitis [Kol90] proved that on every class of ordered structures, a query is definable in IMP(FO) if and only if it is computable in UP $\cap$ coUP. The trees we consider are not ordered. However, they can be ordered by a RAG, as we already saw in Lemma 3.36. ■

The results obtained in this chapter are summarized in Figure 3.10. An arrow from a class of queries C to a class of queries C', means $C \subseteq C'$. A negated arrow from C to C', means there is a Boolean query in C that is not in C'.

$$\begin{tikzcd}
\exists\text{-RAG} = \mathrm{NP} & \forall\text{-RAG} = \mathrm{coNP} & \mathrm{IRAG} = \mathrm{UP} \cap \mathrm{coUP} \\
& \mathrm{RAG} = \text{PFP-LIN} = \mathrm{CRAM}[n] \arrow[ul] \arrow[u] \arrow[ur] \arrow[d, shift left, "/" marking] & \\
& \text{synthesized RAG} \arrow[u, shift left] \arrow[d, shift left, "/" marking] & \\
& \mathrm{BAG} = \exists\text{-BAG} = \forall\text{-BAG} = \mathrm{IBAG} = \mathrm{MSO} \arrow[u, shift left] \arrow[d, shift left, "/" marking, "\text{non-Boolean queries}"] & \\
& \text{synthesized BAG} \arrow[u, shift left] &
\end{tikzcd}$$

Figure 3.10: Summary of results on BAGs and RAGs.

4

Attribute grammars for extended context-free grammars

In this chapter we define extended AGs, a new formalism of attribute grammars suited to query derivation trees of extended context-free grammars. We examine the expressiveness of the formalism and study the complexity of some relevant optimization problems. The latter results will be used in Chapter 6 to drastically improve the known upperbound on the equivalence problem of Region Algebra expressions introduced by Consens and Milo [CM98a]. We conclude by examining relational extended AGs.

We start by introducing the necessary notions to define extended AGs. More concretely, we recall the definition of unambiguous regular expressions and define tree automata over unranked trees on which extended AGs are inspired.

4.1 Basic definitions

4.1.1 Unambiguous regular expressions

As is customary, we denote by $L(r)$ the language defined by the regular expression r. Further, we denote by $\mathrm{Sym}(r)$ the set of Σ-symbols occurring in r. The *marking* $\tilde{r}$ of r is obtained by subscripting in r the first occurrence of a symbol of $\mathrm{Sym}(r)$ by 1, the second by 2, and so on. For example, $a_1(a_2 + b_3^*)^* a_4$ is the marking of

$a(a+b^*)^*a$. We let $|r|$ denote the number of occurrences of Σ-symbols in r, while $r(i)$ denotes the Σ-symbol at the i-th occurrence in r for each $i \in \{1, \ldots, |r|\}$. Let $\widetilde{\Sigma}$ be the alphabet obtained from Σ by subscribing every symbol by all natural numbers, i.e., $\widetilde{\Sigma} := \{a_i \mid a \in \Sigma, i \in \mathbb{N}\}$. If $w \in \widetilde{\Sigma}^*$ then $w^\#$ denotes the string obtained from w by dropping the subscripts.

In the definition of extended AGs we shall restrict ourselves to unambiguous regular expressions defined as follows:

Definition 4.1 A regular expression r over Σ is *unambiguous* if for all $v, w \in L(\tilde{r})$, $v^\# = w^\#$ implies $v = w$.

That is, a regular expression r is unambiguous if every string in $L(r)$ can be matched to r in only one way. For example, the regular expression $(a+b)^*$ is unambiguous while $(aa+a)^*$ is not. Indeed, it is easily checked that the string aa can be matched to $(aa+a)^*$ in two different ways.

The following proposition, obtained by Book et al. [BEGO71], says that the restriction to unambiguous regular expressions is no loss of generality.

Proposition 4.2 *For every regular language R there exists an unambiguous regular expression r such that $L(r) = R$.*

Now, for every unambiguous regular expression r there exists an NFA M_r with the property that can be informally stated as follows: if $w \in L(r)$ then there exists only one path in M_r that accepts w. That is, M_r can accept w only in one manner. We introduce some more notation to define this automaton M_r.

If w is a string and r is an unambiguous regular expression with $w \in L(r)$, then $\tilde{w}_r$ denotes the unique string over $\widetilde{\Sigma}$ such that $\tilde{w}_r^\# = w$ and $\tilde{w}_r \in L(\tilde{r})$. For $i = 1, \ldots, |w|$, define $\mathrm{pos}_r(i, w)$ as the subscript of the i-th letter in $\tilde{w}_r$. Intuitively, $\mathrm{pos}_r(i, w)$ indicates the position in r matching the i-th letter of w. For example, if $r = a(b+a)^*$ and $w = abba$, then $\tilde{r} = a_1(b_2 + a_3)^*$ and $\tilde{w}_r = a_1 b_2 b_2 a_3$. Hence,

$$\mathrm{pos}_r(1, w) = 1, \quad \mathrm{pos}_r(2, w) = 2, \quad \mathrm{pos}_r(3, w) = 2, \text{ and } \mathrm{pos}_r(4, w) = 3.$$

The following lemma is obtained by Book et al. [BEGO71].

Lemma 4.3 *For every unambiguous regular expression r there exists an NFA M_r over the states $\{0, \ldots, |r|\}$ with start state 0 such that*

1. *$L(r) = L(M_r)$;*
2. *for every string $w \in L(r)$ there exists only one valid state assignment ρ_w for w; and*
3. *for $i = 1, \ldots, n$, $\rho_w(i) = \mathrm{pos}_r(i, w)$.*

Moreover, M_r can be constructed in time polynomial in the size of r.

Proviso 4.4 In the remaining of this chapter, when we say *regular expression*, we always mean *unambiguous* regular expression.

4.1.2 Extended context-free grammars

Extended AGs are defined over extended context-free grammars which are defined as follows:

Definition 4.5 An *extended context-free grammar* (ECFG) is a tuple $G = (N, T, P, U)$, where

- T and N are disjoint finite non-empty sets, called the set of *terminals* and *non-terminals*, respectively;
- $U \in N$ is the *start symbol*; and
- P is a set of *productions* consisting of rules of the form $X \to r$ where $X \in N$ and r is a regular expression over $N \cup T$ such that $\varepsilon \notin L(r)$ and $L(r) \neq \emptyset$. Additionally, if $X \to r_1$ and $X \to r_2$ belong to P then $L(r_1) \cap L(r_2) \neq \emptyset$.

A *derivation tree* $\mathbf{t}$ over an ECFG G is a tree labeled with symbols from $N \cup T$ such that

- the root of $\mathbf{t}$ is labeled with U;
- for every interior node $\mathbf{n}$ with children $\mathbf{n}_1, \ldots, \mathbf{n}_m$ there exists a production $X \to r$ such that $\mathbf{n}$ is labeled with X, for $i = 1, \ldots, m$, $\mathbf{n}_i$ is labeled with X_i, and $X_1 \cdots X_m \in L(r)$; we say that $\mathbf{n}$ is *derived* by $X \to r$; and
- every leaf node is labeled with a terminal.

Note that derivation trees of ECFGs are *unranked* in the sense that the number of children of a node need not be bounded by any constant and does not depend on the label of that node.

Throughout this chapter we make the harmless technical assumption that the start symbol does not occur on the right-hand side of a production.

4.1.3 Tree automata over unranked trees

We continue with the definition of nondeterministic bottom-up tree automata over unranked trees [BKMW98] by which the mechanism of extended AGs is inspired. Interestingly, these automata will also be used to obtain the exact complexity of testing non-emptiness and equivalence of extended AGs in Section 4.5.

Definition 4.6 A *nondeterministic bottom-up tree automaton* (NBTA) is a tuple $B = (Q, \Sigma, F, \delta)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and δ is a function $Q \times \Sigma \to 2^{Q^*}$ such that $\delta(q, a)$ is a regular language for every $a \in \Sigma$ and $q \in Q$. The semantics of B on a tree $\mathbf{t}$, denoted by $\delta^*(\mathbf{t})$, is defined inductively as follows: if $\mathbf{t}$ consists of only one node labeled with a then $\delta^*(\mathbf{t}) = \{q \mid \varepsilon \in \delta(q, a)\}$; if $\mathbf{t}$ is of the form

$$\begin{array}{ccc} & a & \\ \swarrow & \cdots & \searrow \\ \mathbf{t}_1 & & \mathbf{t}_n, \end{array}$$

then

$$\delta^*(\mathbf{t}) = \{q \mid \exists q_1 \in \delta^*(\mathbf{t}_1), \ldots, \exists q_n \in \delta^*(\mathbf{t}_n) \text{ and } q_1 \cdots q_n \in \delta(q, a)\}.$$

A tree $\mathbf{t}$ over Σ is *accepted* by the automaton B if $\delta^*(\mathbf{t}) \cap F \neq \emptyset$. The tree language *defined* by B, denoted by $L(B)$, consists of the trees accepted by B. A tree language $\mathcal{T}$ is *recognizable* if there exists an NBTA B such that $\mathcal{T} = L(B)$.

Further, we say that B is *deterministic* when $\delta(q, a) \cap \delta(q', a) = \emptyset$ for every $a \in \Sigma$ and $q, q' \in Q$ with $q \neq q'$. We again use the abbreviation DBTA to refer to such automata. It will always be clear from the context whether we are considering ranked or unranked trees.

We represent the string languages $\delta(q, a)$ by NFAs. The *size* of B then is the sum of the sizes of Q, Σ, and the NFAs defining the transition function.

Proviso 4.7 In this chapter, all tree automata will work over unranked trees. Similarly, when we say recognizable language in this chapter, we always talk about unranked trees.

A detailed study of tree automata over unranked trees has been initiated by Brüggemann-Klein, Murata and Wood [BKMW98, Mur95]. Among many things, they show that DBTAs are as expressive as NBTAs and that the recognizable languages are closed under the Boolean operations.

Tree automata are defined over an arbitrary alphabet, but we consider derivation trees of ECFGs in this chapter. This seeming distinction can be dispensed with since we can always restrict an NBTA to the derivation trees of an ECFG as illustrated next. We point out that this lemma is well known for the ranked case with respect to CFGs [GS97].

Lemma 4.8 *Let $G = (N, T, P, U)$ be an ECFG and let B be an NTBA over $\Sigma \subseteq N \cup T$. Then there exists an NBTA B^G such that $L(B^G) = L(G) \cap L(B)$.*

Proof. We define an NBTA M such that $L(M) = L(G)$. Since recognizable tree languages are closed under the Boolean operations, we can then define B^G as an automaton accepting $L(M) \cap L(B)$.

Define $M = (Q, N \cup T, F, \delta)$, where $Q = T \cup P$, $F = \{U \to r \mid U \to r \in P\}$, and δ is defined as follows: for every $\sigma_1 \in T$ and $\sigma_2 \in T \cup N$,

$$\delta(\sigma_1, \sigma_2) := \begin{cases} \{\varepsilon\} & \text{if } \sigma_1 = \sigma_2; \\ \emptyset & \text{otherwise,} \end{cases}$$

and for every $X \to r \in P$ and $Y \in N \cup T$

$$\delta(X \to r, Y) := \begin{cases} L(r) & \text{if } X = Y; \\ \emptyset & \text{otherwise.} \end{cases}$$

■

The proof of the following lemma is a straightforward generalization of the ranked case (see, e.g., the survey paper by Vardi [Var89]).

Lemma 4.9 *Deciding whether the tree language accepted by an NBTA is non-empty is in PTIME.*

Proof. Let $B = (Q, \Sigma, F, \delta)$ be an NBTA. We inductively compute the set of *reachable* states R defined as follows: $q \in R$ iff there exists a tree $\mathbf{t}$ with $q \in \delta^*(\mathbf{t})$. Obviously, $L(B) \neq \emptyset$ if and only if $R \cap F \neq \emptyset$. Define for all $n > 0$,

$$\begin{aligned} R_1 &:= \{q \in Q \mid \exists a \in \Sigma : \varepsilon \in \delta(q, a)\}; \\ R_{n+1} &:= \{q \in Q \mid \exists a \in \Sigma : \delta(q, a) \cap R_n^* \neq \emptyset\}. \end{aligned}$$

Note that for all n, $R_n \subseteq R_{n+1} \subseteq Q$. Hence, $R_{|Q|} = R_{|Q|+1}$. Thus, define R as $R_{|Q|}$.

Clearly, R_1 can be computed in time linear in the size of B. Since testing non-emptiness of $\delta(q, a) \cap R_n^*$ can be done in time polynomial in the sum of the sizes of these (see, e.g., [HU79]), each R_{n+1} can be computed in time polynomial in the size of B. This concludes the proof of the lemma. ∎

We end this section by generalizing Theorem 2.13 to unranked trees by showing that an unranked tree language is recognizable if and only if it is definable in MSO. The ideas involved in this proof will be generalized in Section 4.4 to characterize the expressiveness of extended AGs.

A DBTA B can be defined in MSO in the usual manner: the MSO sentence defining the behavior of B just guesses states and verifies the consistency of its guesses with the transition function. The latter can now no longer be done in FO, as was the case for ranked trees, because the transition functions are now determined by regular languages. However, by Theorem 2.7 this check can be readily done in MSO.

For the other direction, we again show that the $\equiv_k^{\mathrm{MSO}}$-type of a tree can be computed by a DBTA $B = (\Phi_k, \Sigma, \delta, F)$, for some fixed k. The idea is the same as for the ranked case. Again, the type of the children of a node $\mathbf{n}$ of a tree $\mathbf{t}$ plus the label of $\mathbf{n}$ determine $\tau_k^{\mathrm{MSO}}(\mathbf{t_n})$. The problem is that now, as there is no bound on the number of children of a vertex, the correspondence between the children's types and the type of the whole subtree is no longer given by a finite function, as was the case for ranked trees. Instead, this correspondence is controlled by a regular language. Therefore, for each $\sigma \in \Sigma$ and $\theta \in \Phi_k$, we define $\delta(\theta, \sigma)$ as the set of strings $\theta_1 \cdots \theta_n$ where for $i = 1, \ldots, n$, $\theta_i \in \Phi_k$, and whenever for a tree $\mathbf{t}$ and a node $\mathbf{n}$ labeled with σ with n children, $\tau_k(\mathbf{t_{n}}_i) = \theta_i$, for each $i = 1, \ldots, n$, then $\tau_k(\mathbf{t_n}) = \theta$. We now show that $\delta(\theta, \sigma)$ is indeed a regular language. To this end we state the following proposition.

Proposition 4.10 *Let k be a natural number, $\sigma \in \Sigma$, and let $\mathbf{t}_1, \ldots, \mathbf{t}_n, \mathbf{s}_1, \ldots, \mathbf{s}_m, \mathbf{t}, \mathbf{s}$ be trees. If $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n) \equiv_k^{\mathrm{MSO}} \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_m)$ and $\mathbf{t} \equiv_k^{\mathrm{MSO}} \mathbf{s}$, then $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n, \mathbf{t}) \equiv_k^{\mathrm{MSO}} \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_m, \mathbf{s})$.*

Proof. The proof is almost identical to the proof of Proposition 2.12. We just combine the winning strategies in the subgames

$$G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n); \sigma(\mathbf{s}_1, \ldots, \mathbf{s}_m))$$

and $G_k^{\mathrm{MSO}}(\mathbf{t};\mathbf{s})$ to obtain a winning strategy in

$$G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n,\mathbf{t});\sigma(\mathbf{s}_1,\ldots,\mathbf{s}_m,\mathbf{s})).$$

At the end of the game, the selected nodes define partial isomorphisms for all pairs of respective substructures. To ensure that they also define a partial isomorphism between the entire structures one only has to check the relations E and $<$ between selected nodes coming from different substructures. There is only one technicality in showing this. To this end, we note the following. If the spoiler picks the root of a $\mathbf{t}_a$ ($a \in \{1,\ldots,n\}$) in his l-th move with $l < k$ in $G_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n);\sigma(\mathbf{s}_1,\ldots,\mathbf{s}_m))$, then the duplicator is forced to answer with the root of an $\mathbf{s}_b$ ($b \in \{1,\ldots,m\}$). Indeed, if he does not do so and picks another node, say $\mathbf{e}$, then in the next round the spoiler just picks the parent of $\mathbf{e}$ to which the duplicator has no answer. The same holds when the spoiler picks the root of $\mathbf{t}$, then the duplicator is forced to pick the root of $\mathbf{s}$.

Suppose in a play elements c_i and c_j are chosen such that $c_i < c_j$, c_i is the root of a $\mathbf{t}_a$, and c_j is the root of $\mathbf{t}$, then the above discussion implies that d_i is the root of a $\mathbf{s}_b$ and d_j is the root of $\mathbf{s}$ simply because they are picked before the k-th move in their subgames. Hence, $d_i < d_j$, as had to be shown.

Suppose in a play c_i and c_j are chosen such that c_i is the root of $\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n)$ and c_j is the root of $\mathbf{t}$. Then, $E(c_i,c_j)$. By a similar argument as above, it can be shown that d_i has to be the root of $\sigma(\mathbf{s}_1,\ldots,\mathbf{s}_m)$ and d_j has to be the root of $\mathbf{s}$. Hence, $E(d_i,d_j)$. ■

The above proposition implies that we can compute the $\equiv_k^{\mathrm{MSO}}$-type of a tree $\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n)$, by incrementally reading the $\equiv_k^{\mathrm{MSO}}$-types of the $\mathbf{t}_i$, starting from the state $\tau_k^{\mathrm{MSO}}(\sigma)$. Indeed, define $M_{\theta,\sigma} = (\Phi_k,\Phi_k,s_M,\delta_M,F_M)$ where $s_M = \{\tau_k^{\mathrm{MSO}}(\sigma)\}$, $F_M = \{\theta\}$, and for each $\theta_1,\theta_2 \in \Phi_k$, $\delta_M(\theta_1,\theta_2) = \tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n,\mathbf{t}))$ where $\mathbf{t}_1$, $\ldots$, $\mathbf{t}_n$, and $\mathbf{t}$ are trees such that $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\ldots,\mathbf{t}_n)) = \theta_1$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}) = \theta_2$. Now clearly, $L(M_{\theta,\sigma}) = \delta(\theta,\sigma)$.

From the above the following theorem readily follows.

Theorem 4.11 *An unranked tree language is recognizable if and only if it is definable in MSO.*

In the next chapter we will be needing a tree automaton computing the type $\tau_k^{\mathrm{MSO}}(\mathbf{t},\mathrm{root}(\mathbf{t}))$ for each input tree. Such an automaton is just a slight extension of the automaton discussed above. To show this we have the following proposition. To be precise, we only need item 2, the other items are used in Section 4.4. We abbreviate $\tau_k^{\mathrm{MSO}}(\mathbf{t},\mathrm{root}(\mathbf{t}))$ by $\tau_k^{\mathrm{MSO}}(\mathbf{t},\mathrm{root})$.

Proposition 4.12 *Let k be a natural number, σ be a label, $\mathbf{t}$ and $\mathbf{s}$ be two trees, $\mathbf{n}$ be a node of $\mathbf{t}$ with children $\mathbf{n}_1,\ldots,\mathbf{n}_n$, and $\mathbf{m}$ be a node of $\mathbf{s}$ with children $\mathbf{m}_1,\ldots,\mathbf{m}_m$.*

1. *If $(\overline{\mathbf{t}_\mathbf{n}},\mathbf{n}) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_\mathbf{m}},\mathbf{m})$ and $(\mathbf{t}_\mathbf{n},\mathbf{n}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_\mathbf{m},\mathbf{m})$, then $(\mathbf{t},\mathbf{n}) \equiv_k^{\mathrm{MSO}} (\mathbf{s},\mathbf{m})$.*

2. *If* $(\sigma(\mathbf{t}_{\mathbf{n}_1},\dots,\mathbf{t}_{\mathbf{n}_{n-1}}),\mathrm{root}) \equiv_k^{\mathrm{MSO}} (\sigma(\mathbf{s}_{\mathbf{m}_1},\dots,\mathbf{s}_{\mathbf{m}_{m-1}}),\mathrm{root})$ *and* $(\mathbf{t}_{\mathbf{n}_n},\mathbf{n}_n) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_{\mathbf{m}_m},\mathbf{m}_m)$*, then* $(\mathbf{t}_{\mathbf{n}},\mathbf{n}) \equiv_k^{\mathrm{MSO}} (\mathbf{s}_{\mathbf{m}},\mathbf{m})$.

3. *Let the label of* $\mathbf{n}$ *and* $\mathbf{m}$ *be* σ*. For* $i \in \{1,\dots,n\}$ *and* $j \in \{1,\dots,m\}$*, if*

   - $(\overline{\mathbf{t}_{\mathbf{n}}},\mathbf{n}) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_{\mathbf{m}}},\mathbf{m})$,
   - $(\sigma(\mathbf{t}_{\mathbf{n}_1},\dots,\mathbf{t}_{\mathbf{n}_{i-1}}),\mathrm{root}) \equiv_k^{\mathrm{MSO}} (\sigma(\mathbf{s}_{\mathbf{m}_1},\dots,\mathbf{s}_{\mathbf{m}_{j-1}}),\mathrm{root})$,
   - $(\sigma(\mathbf{t}_{\mathbf{n}_{i+1}},\dots,\mathbf{t}_{\mathbf{n}_n}),\mathrm{root}) \equiv_k^{\mathrm{MSO}} (\sigma(\mathbf{s}_{\mathbf{m}_{j+1}},\dots,\mathbf{s}_{\mathbf{m}_m}),\mathrm{root})$*, and*
   - *the label of* $\mathbf{n}_i$ *equals the label of* $\mathbf{m}_j$*,*

   then $(\overline{\mathbf{t}_{\mathbf{n}_i}},\mathbf{n}_i) \equiv_k^{\mathrm{MSO}} (\overline{\mathbf{s}_{\mathbf{m}_j}},\mathbf{m}_j)$.

Proof. We focus on the third case where there are altogether 4 subgames including the trivial game in which one structure consists only of $\mathbf{n}i$ and the other of $\mathbf{m}j$. The winning strategy in the game on $(\overline{\mathbf{t}_{\mathbf{n}i}},\mathbf{n}i)$ and $(\overline{\mathbf{s}_{\mathbf{m}j}},\mathbf{m}j)$ just combines the winning strategies in those 4 subgames. At the end of the game, the selected vertices define partial isomorphisms for all pairs of respective substructures. To ensure that they also define a partial isomorphism between the entire structures one only has to check the relations $<$ and E between the chosen elements, and the distinguished constants $\mathbf{n}i$ and $\mathbf{m}i$. This immediately follows from the following observation. The distinguished constants in the subgames make sure that (i) whenever in the game on $(\overline{\mathbf{t}_{\mathbf{n}i}},\mathbf{n}i)$ and $(\overline{\mathbf{s}_{\mathbf{m}j}},\mathbf{m}j)$ a child of $\mathbf{n}$ $(\mathbf{m})$ is chosen, the duplicator has to reply with a child of $\mathbf{m}$ $(\mathbf{n})$; and, (ii) whenever $\mathbf{n}$ $(\mathbf{m})$ is chosen, the duplicator has to reply with $\mathbf{m}$ $(\mathbf{n})$. ■

We will need the following lemma in Section 5.3 and Section 6.2.

Lemma 4.13 *Let* k *be a natural number. There exists a DBTA* $B = (Q,\Sigma,\delta,F)$ *such that* $\delta^*(\mathbf{t}) = \tau_k^{\mathrm{MSO}}(\mathbf{t},\mathrm{root}(\mathbf{t}))$*, for every unranked tree* $\mathbf{t}$.

Proof. Define Q as the set Φ_k. Here we take Φ_k as the set of of $\equiv_k^{\mathrm{MSO}}$-types of trees with one distinguished node. For each $\theta \in \Phi_k$ and $\sigma \in \Sigma$, define $\delta(\theta,\sigma)$ as the regular language defined by the automaton $M_{\theta,\sigma} = (\Phi_k,\Phi_k,s_M,\delta_M,F_M)$. This automaton is defined as follows: $s_M = \{\tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma),\mathrm{root})\}$; $F_M = \{\theta\}$; and for each $\theta_1,\theta_2 \in \Phi_k$, $\delta_M(\theta_1,\theta_2) = \tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\dots,\mathbf{t}_n,\mathbf{t}),\mathrm{root})$ where $\mathbf{t}_1$, ..., $\mathbf{t}_n$, and $\mathbf{t}$ are trees such that $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_1,\dots,\mathbf{t}_n),\mathrm{root}) = \theta_1$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t},\mathrm{root}) = \theta_2$. By Proposition 4.12(2), it does not matter which members of the classes θ_1 and θ_2 we choose. ■

4.2 Attribute grammars over extended context-free grammars

Before defining extended attribute grammars formally, we give an example illustrating the basic ideas.

Consider the ECFG consisting of the sole rule $U \to (A+B)^*$. We now want to construct an attribute grammar selecting those A's that are preceded by an even number of A's and succeeded by an odd number of B's. As for BAGs, we will use semantic rules defining an attribute *select*. This gives rise to two problems not present for BAGs:

(i) U can have an unbounded number of children labeled with A which implies that an unbounded number of attributes should be defined;

(ii) the definition of the *select* attribute of an A depends on its siblings, whose number is again unbounded.

We resolve this in the following way. For (i), we just define *select* uniformly for each node that corresponds to the first position in the regular expression $(A+B)^*$. For (ii), we use regular languages as semantic rules rather than propositional formulas. The following extended AG now expresses the above query:

$$U \to (A+B)^* \qquad \mathit{select}(1) := \langle \sigma_1 = \mathrm{lab}, \sigma_2 = \mathrm{lab}; \\ R_{true} = (B^*AB^*AB^*)^*\#A^*BA^*(A^*BA^*BA^*)^*, \\ R_{false} = (A+B+\#)^* - R_{true}\rangle.$$

The 1 in *select*(1) indicates that the attribute *select* is defined uniformly for every node corresponding to the first position in $(A+B)^*$. In the first part of the semantic rule, each σ_i lists the attributes of position i that will be used. Here, both for position 1 and 2 this is only the attribute lab which is a special attribute containing the label of the node. Consider the input tree $U(AAABBB)$. Then to check whether the third A is selected we enumerate the attributes mentioned in the first part of the rule and insert the symbol # before the node under consideration. This is just the string

1	1	1	2	2	2	position in $(A+B)^*$
A	**A**	**#A**	**B**	**B**	**B**	
1	2	3	4	5	6	position in $AAABBB$

The attribute *select* of the third child, for instance, will now be assigned the value *true* since the above string belongs to R_{true}. Note that

$$(B^*AB^*AB^*)^* \text{ and } A^*BA^*(A^*BA^*BA^*)^*$$

define the set of strings with an even number of A's and with an odd number of B's, respectively. The above will now be defined formally.

We now define extended attribute grammars (extended AGs) over ECFGs whose attributes can take only values from a finite set D.

Proviso 4.14 Unless explicitly stated otherwise, we always assume an ECFG $G = (N, T, P, U)$. When we say tree we always mean derivation tree of G.

Definition 4.15 An *attribute grammar vocabulary* is a tuple $(D, A, \mathrm{Syn}, \mathrm{Inh})$, where

- D is a finite set of values called the *semantic domain*. We assume that D always contains the Boolean values 0 and 1;

- A is a finite set of symbols called *attributes*; we always assume that A contains the attribute *lab*;
- Syn and Inh are functions from $N \cup T$ to the powerset of $A - \{\text{lab}\}$ such that for every $X \in N$, $\text{Syn}(X) \cap \text{Inh}(X) = \emptyset$; for every $X \in T$, $\text{Syn}(X) = \emptyset$; and $\text{Inh}(U) = \emptyset$.

If $a \in \text{Syn}(X)$, we say that a is a *synthesized attribute of* X. If $a \in \text{Inh}(X)$, we say that a is an *inherited attribute of* X. We also agree that *lab* is an attribute of every X (this is a predefined attribute; for each node its value will be the label of that node). The above conditions express that an attribute cannot be a synthesized and an inherited attribute of the same grammar symbol, that terminal symbols do not have synthesized attributes, and that the start symbol does not have inherited attributes.

We now formally define the semantic rules of extended AGs. For a production $p = X \to r$, define $p(0) = X$, and for $i \in \{1, \ldots, |r|\}$, define $p(i) = r(i)$. We fix some attribute grammar vocabulary $(A, D, \text{Syn}, \text{Inh})$ in the following definitions.

Definition 4.16 1. Let $p = X \to r$ be a production of G and let a be an attribute of $p(i)$ for some $i \in \{0, \ldots, |r|\}$. The triple (p, a, i) is called a *context* if $a \in \text{Syn}(p(i))$ implies $i = 0$, and $a \in \text{Inh}(p(i))$ implies $i > 0$.

2. A *rule in the context* (p, a, i) is an expression of the form

$$a(i) := \langle \sigma_0, \ldots, \sigma_{|r|}; (R_d)_{d \in D} \rangle,$$

where

- for $j = \{0, \ldots, |r|\}$, σ_j is a sequence of attributes of $p(j)$;
- if $i = 0$, then, for each $d \in D$, R_d is a regular language over the alphabet D; and
- if $i > 0$, then, for each $d \in D$, R_d is a regular language over the alphabet $D \cup \{\#\}$.

For all $d, d' \in D$, if $d \neq d'$ then $R_d \cap R_{d'} = \emptyset$. Further, if $i = 0$ then $\bigcup_{d \in D} R_d = D^*$. If $i > 0$ then $\bigcup_{d \in D} R_d$ should contain all strings over D with exactly one occurrence of the symbol $\#$. Note that a R_d is allowed to contain strings with several occurrences of $\#$. We always assume that $\# \notin D$.

An extended AG is then defined as follows:

Definition 4.17 An *extended attribute grammar (extended AG)* $\mathcal{F}$ consists of an attribute grammar vocabulary, together with a mapping assigning to each context a rule in that context.

It will always be understood which rule is associated to which context. We illustrate the above definitions with an example.

Example 4.18 In Figure 4.1 an example of an extended AG $\mathcal{F}$ is depicted over the ECFG of Figure 1.6. Recall that every grammar symbol has the attribute lab; for each node this attribute has the label of that node as value. We have Syn(Word) $=$ $\{king, lord\}$, Syn(Verse) $= \{king\_lord\}$, Syn(Poem) $= \{result\}$, and Inh(Poem) $=$ $\{first\}$. The grammar symbols DB, a, ... , z, Verse, and Word have no attributes apart from lab. The semantics of this extended AG will be explained below. Here, $D = \{0, 1, \mathrm{a}, \ldots, \mathrm{z}, \mathrm{DB}, \mathrm{Poem}, \mathrm{Verse}, \mathrm{Word}\}$. We use regular expressions to define the languages R_1; for the first rule, R_0 is defined as $(D \cup \{\#\})^* - R_1$; for all other rules, R_0 is defined as $D^* - R_1$; those R_d that are not specified are empty; ε stands for the empty sequence of attributes. ■

DB $\to$ Poem$^+$	$\mathit{first}(1) := \langle \sigma_0 = \mathit{lab}, \sigma_1 = \mathit{lab}; R_1 = \mathrm{DB}\#\mathrm{Poem}^+ \rangle$
Poem $\to$ Verse$^+$	$\mathit{result}(0) := \langle \sigma_0 = \mathit{first}, \sigma_1 = \mathit{king\_lord};$ $R_1 = 1(1+0)^* + 0(1(1+0))^*(1+\varepsilon) \rangle$
Verse $\to$ Word$^+$	$\mathit{king\_lord}(0) := \langle \sigma_0 = \varepsilon, \sigma_1 = (\mathit{king}, \mathit{lord});$ $R_1 = (0+1)^* + 1 + (0+1)^* \rangle$
Word $\to (a + \ldots + z)^+$	$\mathit{king}(0) := \langle \sigma_0 = \varepsilon, \sigma_1 = \mathit{lab}, \ldots, \sigma_{26} = \mathit{lab}; R_1 = \{\mathrm{king}\} \rangle$
	$\mathit{lord}(0) := \langle \sigma_0 = \varepsilon, \sigma_1 = \mathit{lab}, \ldots, \sigma_{26} = \mathit{lab}; R_1 = \{\mathrm{lord}\} \rangle$

Figure 4.1: Example of an extended AG.

The semantics of an extended AG is that it defines attributes of the nodes of derivation trees of the underlying grammar G. This is formalized next.

Definition 4.19 If $\mathbf{t}$ is a derivation tree of G then a *valuation v of* $\mathbf{t}$ is a function that maps each pair $(\mathbf{n}, a)$, where $\mathbf{n}$ is a node in $\mathbf{t}$ and a is an attribute of the label of $\mathbf{n}$, to an element of D, and that maps for every $\mathbf{n}$, $v((\mathit{lab}, \mathbf{n}))$ to the label of $\mathbf{n}$.

In the sequel, for a pair $(\mathbf{n}, a)$ as above we will use the more intuitive notation $a(\mathbf{n})$. To define the semantics of $\mathcal{F}$ we first need the following definition. If $\sigma = a_1 \cdots a_k$ is a sequence of attributes and $\mathbf{n}$ is a node of $\mathbf{t}$, then define $\sigma(\mathbf{n})$ as the sequence of attribute-node pairs $\sigma(\mathbf{n}) = a_1(\mathbf{n}) \cdots a_k(\mathbf{n})$.

Definition 4.20 Let $\mathbf{t}$ be a derivation tree, $\mathbf{n}$ a node of $\mathbf{t}$, and a an attribute of the label of $\mathbf{n}$.

Synthesized Let $\mathbf{n}_1, \ldots, \mathbf{n}_m$ be the children of $\mathbf{n}$ derived by $p = X \to r$, and let $\langle \sigma_0, \ldots, \sigma_{|r|}; (R_d)_{d \in D} \rangle$ be the rule associated to the context $(p, a, 0)$. Define for $l \in \{1, \ldots, m\}$, $j_l = \mathrm{pos}_r(l, w)$, where w is the string formed by the labels of the children of $\mathbf{n}$. Then define $W(a(\mathbf{n}))$ as the sequence

$$\sigma_0(\mathbf{n}) \cdot \sigma_{j_1}(\mathbf{n}_1) \cdots \sigma_{j_m}(\mathbf{n}_m).$$

For each d, we denote the language R_d associated to $a(\mathbf{n})$ by $R_d^{a(\mathbf{n})}$.

Inherited Let $\mathbf{n}_1, \ldots, \mathbf{n}_{k-1}$ be the left siblings, $\mathbf{n}_{k+1}, \ldots, \mathbf{n}_m$ be the right siblings, and $\mathbf{n}_0$ be the parent of $\mathbf{n}$. Let $\mathbf{n}_0$ be derived by $p = X \to r$, and define for $l \in \{1, \ldots, m\}$, $j_l = \mathrm{pos}_r(l, w)$, where w is the string formed by the labels of the children of $\mathbf{n}_0$. Let $\langle \sigma_0, \ldots, \sigma_{|r|}; (R_d)_{d \in D} \rangle$ be the rule associated to the context (p, a, j_k). Now define $W(a(\mathbf{n}))$ as the sequence

$$\sigma_0(\mathbf{n}_0) \cdot \sigma_{j_1}(\mathbf{n}_1) \cdots \sigma_{j_{k-1}}(\mathbf{n}_{k-1}) \cdot \# \cdot \sigma_{j_k}(\mathbf{n}) \cdots \sigma_{j_{k+1}}(\mathbf{n}_{k+1}) \sigma_{j_m}(\mathbf{n}_m).$$

For each d, we denote the language R_d associated to $a(\mathbf{n})$ by $R_d^{a(\mathbf{n})}$.

If v is a valuation then define $v(W(a(\mathbf{n})))$ as the string obtained from $W(a(\mathbf{n}))$ by replacing each $b(\mathbf{m})$ in $W(a(\mathbf{n}))$ by $v(b(\mathbf{m}))$. Note that the empty sequence is just replaced by the empty string.

We are now ready to define the semantics of an extended AG $\mathcal{F}$ on a derivation tree.

Definition 4.21 Given an extended AG $\mathcal{F}$ and a derivation tree $\mathbf{t}$, we define a sequence of partial valuations $(\mathcal{F}_j)_{j \geq 0}$ as follows:

1. $\mathcal{F}_0(\mathbf{t})$ is the valuation that maps, for every node $\mathbf{n}$, $\mathrm{lab}(\mathbf{n})$ to the label of $\mathbf{n}$ and is undefined everywhere else;

2. for $j > 0$, if $\mathcal{F}_{j-1}(\mathbf{t})$ is defined on all $b(\mathbf{m})$ occurring in $W(a(\mathbf{n}))$ then

$$\mathcal{F}_j(\mathbf{t})(a(\mathbf{n})) = d,$$

where $\mathcal{F}_{j-1}(W(a(\mathbf{n}))) \in R_d^{a(\mathbf{n})}$. Note that this is well defined.

If for every $\mathbf{t}$ there is an l such that $\mathcal{F}_l(\mathbf{t})$ is totally defined (this implies that $\mathcal{F}_l(\mathbf{t}) = \mathcal{F}_{l-1}(\mathbf{t})$) then we say that $\mathcal{F}$ is *non-circular*. Obviously, non-circularity is an important property. In the next section we show that it is decidable whether an extended AG is non-circular. Therefore, we can state the following proviso.

Proviso 4.22 In the sequel we always assume an extended AG to be non-circular.

Definition 4.23 The valuation $\mathcal{F}(\mathbf{t})$ equals $\mathcal{F}_l(\mathbf{t})$ with l such that $\mathcal{F}_l(\mathbf{t}) = \mathcal{F}_{l+1}(\mathbf{t})$.

Proviso 4.24 In this chapter, whenever we say query, we always mean *unary* query.

An extended AG $\mathcal{F}$ can be used in a simple way to express queries. Among the attributes in the vocabulary of $\mathcal{F}$, we designate some attribute *result*, and define:

Definition 4.25 An extended AG $\mathcal{F}$ expresses the query $\mathcal{Q}$ defined by

$$\mathcal{Q}(\mathbf{t}) = \{\mathbf{n} \mid \mathcal{F}(\mathbf{t})(\mathit{result}(\mathbf{n})) = 1\},$$

for every tree $\mathbf{t}$.

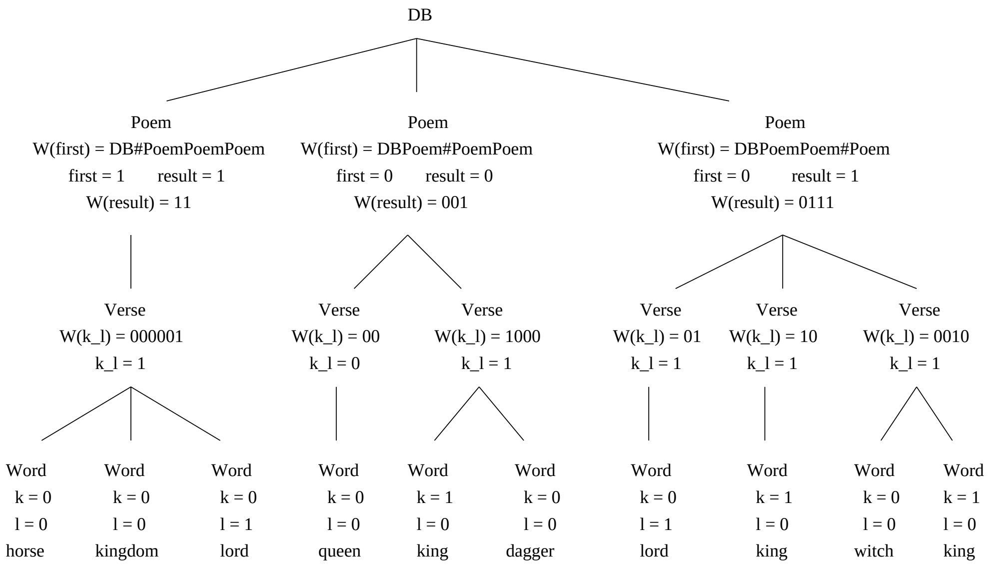

Figure 4.2: A derivation tree and its valuation as defined by the extended AG in Figure 4.1.

Example 4.26 Recall the extended AG $\mathcal{F}$ of Figure 4.1. This extended AG selects the first poem and every poem that has the strings king or lord in every other verse starting from the first one. In Figure 4.2 an illustration is given of the result of $\mathcal{F}$ on a derivation tree $\mathbf{t}$. At each node $\mathbf{n}$, we show the values $\mathcal{F}(W(a(\mathbf{n})))$ and $\mathcal{F}(\mathbf{t})(a(\mathbf{n}))$. We abbreviate $a(\mathbf{n})$ by a, *king* by *k*, *lord* by *l*, and *king_lord* by *k_l*.

The definition of the inherited attribute *first* indicates how the use of # can distinguish in a uniform way between different occurrences of the grammar symbol Poem. This is only a simple example. In the next section we show that extended AGs can express all queries definable in MSO. Hence, they can also specify all relationships between siblings definable in MSO.

The language R_1 associated to *result* (cf. Figure 4.1), contains those strings representing that the current Poem is the first one, or representing that for every other verse starting at the first one the value of the attribute *king_lord* is 1. ■

The *size* of an extended AG is the sum of the sizes of the attribute grammar vocabulary, the ECFG and the size of the semantic rules where we represent the regular languages R_d by NFAs.

4.3 Non-circularity

In this section we show that it is decidable whether an extended AG is non-circular. More concretely, we show that deciding non-circularity is in EXPTIME. As it is well known that deciding non-circularity of *standard* AGs is complete for EXPTIME [JOR75], going from ranked to unranked does not increase the complexity of the non-circularity problem.

We first make the following remark indicating that testing non-circularity for extended AGs is slightly more subtle than for standard AGs.

Remark 4.27 Not all the specified attributes in a semantic rule are always used. Indeed, consider the grammar with productions $C \to A + B$, $A \to c$ and $B \to c$. Let $\mathcal{F}$ be an extended AG where the inherited attribute a of A and B is defined in the context $(C \to A + B, a, 1)$ as

$$a(1) := \langle \sigma_0 = \varepsilon, \sigma_1 = \varepsilon, \sigma_2 = a; R_1 \rangle,$$

and in the context $(C \to A + B, a, 2)$ as

$$a(2) := \langle \sigma_0 = \varepsilon, \sigma_1 = a, \sigma_2 = \varepsilon; R_1 \rangle.$$

At first sight $\mathcal{F}$ seems circular. This is, however, not the case since A and B never occur simultaneously in a derivation tree. Consider for example the tree graphically represented as

$$\begin{array}{c} C \\ \downarrow \\ A \\ \downarrow \\ c. \end{array}$$

If the label of $\mathbf{n}$ is A then $W(a(\mathbf{n}))$ is the empty sequence and consequently $\mathcal{F}(a(\mathbf{n})) = 1$ if and only if the empty string belongs to R_1. ■

A naive approach to testing non-circularity is to transform an extended AG $\mathcal{F}$ into a standard AG $\mathcal{F}'$ such that $\mathcal{F}$ is non-circular if and only if $\mathcal{F}'$ is non-circular and then use the known exponential algorithm on $\mathcal{F}'$. We can namely always find an integer N (polynomially depending on $\mathcal{F}$) such that we only have to test non-circularity of $\mathcal{F}$ on trees of rank N. Unfortunately, this approach exponentially increases the size of the AG. Indeed, a production $X \to (a + b) \cdots (a + b)$ (n times), for example, has to be translated to the set of productions $\{X \to w \mid w \in \{a, b\}^* \wedge |w| = N\}$. So, the complexity of the above algorithm is double exponential time. Therefore, we abandon this approach and give a different algorithm whose complexity is in EXPTIME.

To this end, we first generalize the tree walking automata of Bloem and Engelfriet [BE97] to unranked trees. In particular, we show that for each extended AG $\mathcal{F}$, there exists a tree walking automata $W_{\mathcal{F}}$ such that $\mathcal{F}$ is non-circular if and only if $W_{\mathcal{F}}$ does not cycle. Moreover, the size of $W_{\mathcal{F}}$ is polynomial in the size of $\mathcal{F}$. We thus obtain our result by showing that testing whether a tree walking automaton cycles is in EXPTIME.

Definition 4.28 A *nondeterministic tree walking automaton* is a tuple $W = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of final states, and
- $\delta \subseteq Q \times \Sigma \times Q \times \{\downarrow_{\text{first}}, \downarrow_{\text{last}}, \rightarrow, \leftarrow, \uparrow, \text{stay}\}$ is the transition relation.

Intuitively, a tree walking automaton walks over the tree starting at the root. To make sure that the automaton cannot fall off the tree, we augment input trees with the boundary symbols $\leftarrow$, $\rightarrow$, $\downarrow$, and $\uparrow$. For example, the tree $a(b, c)$ augmented with boundary symbols looks like $\downarrow(\rightarrow, a(\rightarrow, b(\uparrow), c(\uparrow), \leftarrow), \leftarrow)$, or more graphically:

$$\begin{array}{ccc} & \downarrow & \\ & \rightarrow a \leftarrow & \\ \rightarrow b & & c \leftarrow \\ \uparrow & & \uparrow \end{array} .$$

When we refer to the root of a tree we mean the root of the tree without the boundary symbols.

A perhaps more elegant solution is to have a separate transition function for the root node, internal nodes and leaf nodes. But since this last approach terribly complicates the proof of the next lemma we just stick to the tree representation with boundary symbols.

Depending on the current state and on the label at the current node, the transition relation determines in which direction the automaton can move and into which states it can change. The possible directions w.r.t. the current node are: go to the first child, the last child, the left sibling, the right sibling, or the parent, or stay at the current node. Of course, we have the obvious restrictions that W can only move to the left, right, down and up, when it reads the symbols $\leftarrow$, $\rightarrow$, $\downarrow$, and $\uparrow$, respectively.

The automaton accepts an input tree when there exists a walk started at the root in the start state that again reaches the root node in a final state. We make this more precise. A *configuration* of W on a tree t is a pair $(\mathbf{n}, q)$ where $\mathbf{n}$ is a node of $\mathbf{t}$ and $q \in Q$. The *start* configuration is $(\text{root}(\mathbf{t}), q_0)$, and each $(\text{root}(\mathbf{t}), q)$ with $q \in F$ is an *accepting* configuration. A *walk* of W on $\mathbf{t}$ is a (possibly infinite) sequence of configurations $c_1 c_2 c_3 \cdots$ where c_1 is the start configuration and each c_{i+1} can be reached from c_i by making one transition. The latter is defined in the obvious way. A walk is *accepting* when it is finite and the last configuration is a final one. Finally, W *accepts* $\mathbf{t}$ when there exists an accepting walk of W on $\mathbf{t}$. However, we will not need this definition further on as we will concentrate on infinite walks.

We need the following definition to state the next lemma.

Definition 4.29 A nondeterministic tree walking automaton *cycles* if there is a tree on which it has an infinite walk.

Lemma 4.30 *Deciding whether a nondeterministic walking tree automaton is cycling is in EXPTIME.*

Proof. Let $W = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic tree walking automaton. For a tree $\mathbf{t}$ define the *behavior relation* of W on $\mathbf{t}$ as the relation $f_{\mathbf{t}}^W \subseteq Q \times (Q \cup \{\#\})$ as follows. For each $q, q' \in Q$,

1. $f_{\mathbf{t}}^W(q, q')$ if there exists a walk of W starting at the root of $\mathbf{t}$ in state q that again returns at the root in state q' with the additional requirement that W is not allowed to move to the left sibling, the right sibling or the parent of the root (recall these are labeled with $\rightarrow$, $\leftarrow$, and $\downarrow$, respectively) during this walk;
2. $f_{\mathbf{t}}^W(q, \#)$ if there is an infinite walk of W starting at the root in state q, again with the additional requirement that W is not allowed to move to the left sinling, the right sibling or the parent of the root during this walk.

The additional requirement mentioned in both of the above cases is needed because we want to compute behavior relations of nodes in a tree, in terms of the behavior relations at the children of those nodes. Therefore, the behavior relations of the subtrees should only be defined by computations that do not leave those subtrees.

Let $f \subseteq Q \times (Q \cup \{\#\})$ be a relation and let $\sigma \in \Sigma$. Then, we say that (f, σ) is *satisfiable* whenever there exists a tree $\mathbf{t}$ with $f_{\mathbf{t}}^W = f$ and the label of root($\mathbf{t}$) is σ. We refer to the tuples (f, σ) as *behavior tuples*. It now suffices to compute the set of all satisfiable behavior tuples to decide whether W is cycling. To see this, we first introduce the following relations that determine the behavior of W when it encounters the boundary of the tree. Define the relations $\delta^\sigma_{\rightleftarrows}$, $\delta^\sigma_{\leftrightarrows}$, and $\delta^\sigma_{\uparrow\downarrow}$, as follows: for each $q, q' \in Q$,

- $\delta^\sigma_{\rightleftarrows}(q, q')$ iff there exists a q'' such that $\delta(q, \sigma, q'', \rightarrow)$ and $\delta(q'', \leftarrow, q', \leftarrow)$;
- $\delta^\sigma_{\leftrightarrows}(q, q')$ iff there exists a q'' such that $\delta(q, \sigma, q'', \leftarrow)$ and $\delta(q'', \rightarrow, q', \rightarrow)$;
- $\delta^\sigma_{\uparrow\downarrow}(q, q')$ iff there exists a q'' such that $\delta(q, \sigma, q'', \uparrow)$ and $\delta(q'', \downarrow, q', \downarrow_{\text{first}})$;

We define $\text{States}(f_1, \ldots, f_n, q) \subseteq Q \cup \{\#\}$ as the set of states reachable from q by applying relations in $f_1, \ldots, f_n$. Further, if one of the relations introduces cycling then $\#$ also belongs to $\text{States}(f_1, \ldots, f_n, q)$. Formally, $\text{States}(f_1, \ldots, f_n, q)$ is the smallest set of states containing q such that if $q' \in \text{States}(f_1, \ldots, f_n, q)$ and $f_i(q', q'')$ then $q'' \in \text{States}(f_1, \ldots, f_n, q)$. Additionally, if $q_1, \ldots, q_{m+1} \in \text{States}(f_1, \ldots, f_n, q)$, $q_1 = q_{m+1}$, and for $i = 1, \ldots, m$, there is a j_i with $f_{j_i}(q_i, q_{i+1})$, then $\# \in \text{States}(f_1, \ldots, f_n, q)$.

So W is cycling whenever there exists a satisfiable behavior tuple (f, σ) such that $\# \in \text{States}(f, \delta^\sigma_{\rightleftarrows}, \delta^\sigma_{\leftrightarrows}, \delta^\sigma_{\uparrow\downarrow}, q_0)$. This just says that an infinite walk can be reached from the start state q_0.

Input: W
% Initialization
for each behavior tuple (f, σ) **do**
    construct $M_{f,\sigma}$
$G := \{(f^W_{\mathbf{t}(\sigma)}, \sigma) \mid \sigma \in \Sigma\}$
% Main loop
repeat
    for each $(f, \sigma) \notin G$ **do**
        if $L(M_{f,\sigma}) \cap G^* \neq \emptyset$
        then $G := G \cup \{(f, \sigma)\}$
until no more changes occur

Figure 4.3: An algorithm computing the set S of weakly satisfiable behavior tuples.

To reduce the complexity of our algorithm we make use of a weaker notion of satisfiability. We say that a behavior tuple (f, σ) is *weakly satisfiable* whenever there exists a satisfiable behavior tuple (g, σ) such that $f \subseteq g$. Note that every satisfiable tuple is also weakly satisfiable.

If g is witnessed by $\mathbf{t}$ then we say that f is *weakly witnessed* by $\mathbf{t}$. Let S be the set of all weakly satisfiable behavior tuples. Then W is cycling whenever there exists a behavior tuple $(f, \sigma) \in S$ such that $\# \in \text{States}(f, \delta^\sigma_{\overleftarrow{\rightarrow}}, \delta^\sigma_{\overrightarrow{\leftarrow}}, \delta^\sigma_{\uparrow\downarrow}, q_0)$.

In Figure 4.3, we give an algorithm computing S. In this algorithm, G is initialized by the set of behavior tuples for all 1-vertex trees. Hereafter, the algorithm tests for each behavior tuple (f, σ) whether it can be obtained by combining behavior tuples in G and adds (f, σ) to G if this is the case. To this end, we use an automaton $M_{f,\sigma}$ over the alphabet consisting of all behavior tuples. In particular, if (f_1, σ_1), ..., (f_n, σ_n) are weakly satisfiable and $(f_1, \sigma_1) \cdots (f_n, \sigma_n) \in L(M_{f,\sigma})$ then (f, σ) is weakly satisfiable. Moreover, if each (f_i, σ_i) is weakly witnessed by $\mathbf{t}_i$, then (f, σ) is weakly witnessed by $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$. From this it follows that all tuples in G are weakly satisfiable. The converse can be shown by induction on the minimal height of the trees weakly witnessing the weakly satisfiable behavior tuples. It follows that after completion of the algorithm $G = S$. Since the size of each $M_{f,\sigma}$ will be exponential in the size of W, the test $L(M_{f,\sigma}) \cap G^* \neq \emptyset$ can be done in exponential time. As there are only exponentially many behavior tuples, the REPEAT loop will iterate at most an exponential number of times. Thus, the total execution time of the algorithm will be exponential in the size of W.

We explain the construction of $M_{f,\sigma}$. First, we define a nondeterministic two-way string automaton $M'_{f,\sigma}$ with one pebble whose size is polynomial in the size of W. By Proposition 2.10, $M'_{f,\sigma}$ is equivalent to a one-way nondeterministic automaton whose size is only exponential in the size of $M'_{f,\sigma}$. We then define $M_{f,\sigma}$ as the latter automaton. On input $(f_1, \sigma_1) \cdots (f_n, \sigma_n)$, $M'_{f,\sigma}$ works as follows. Let each (f_i, σ_i) be weakly witnessed by $\mathbf{t}_i$.

1. For each $q, q' \in Q$ for which $f(q, q')$, the automaton $M'_{f,\sigma}$ has to check whether there exists a walk starting at the root of $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ in state q that again reaches the root in state q'. However, $M'_{f,\sigma}$ does not need to know the tree $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$: $M'_{f,\sigma}$ just guesses this path using the f_i's. That is, $M'_{f,\sigma}$ starts in state q at the root. If W, for example, decides to move to the last child in state q_1, then $M'_{f,\sigma}$ walks to the last position of the string $(f_1, \sigma_1) \cdots (f_n, \sigma_n)$ arriving there in state q_1. Further, if $M'_{f,\sigma}$ arrives at a position labeled with (f_i, σ_i) and W decides to enter the subtree below this position, then $M'_{f,\sigma}$ just examines the relation f_i to see in which states it can return. If W makes a move to, say, the right sibling in state q_2, then $M'_{f,\sigma}$ just makes a right move to state q_2. If $M'_{f,\sigma}$ succeeds in reaching the root in state q', then it considers the next pair of states q_1 and q'_1 for which $f(q_1, q'_1)$. Clearly, $M'_{f,\sigma}$ only needs a number of states that is polynomial in the size of W.

2. For every $q \in Q$ such that $f(q, \#)$, $M'_{f,\sigma}$ has to verify the existence of an infinite walk on $\sigma(\mathbf{t}_1, \ldots, \mathbf{t}_n)$ starting from state q at the root. This can happen in two ways. The first possibility is that W gets into a cycle in one of the subtrees $\mathbf{t}_1, \ldots, \mathbf{t}_n$, say $\mathbf{t}_i$. This can be detected, like in the previous case, by simply guessing a walk reaching position i of the input string $(f_1, \sigma_1) \cdots (f_n, \sigma_n)$ in a state q' such that $f_i(q', \#)$. The second possibility is that W can walk forever on the children of the root. We use the pebble to detect this: $M'_{f,\sigma}$ now just guesses a walk of W using the relations $f_1, \ldots, f_n$ as explained above and nondeterministically puts down its pebble on a position of $(f_1, \sigma_1) \cdots (f_n, \sigma_n)$, memorizes the current state, and proceeds its walk. It then accepts when it reaches the pebble again in the memorized state, which means that W indeed has reached a cycle and hence can walk forever. If this succeeds then $M'_{f,\sigma}$ checks the next state $q' \in Q$ for which $f(q', \#)$. ∎

Next, we define a tree walking automaton $W_{\mathcal{F}}$ for an extended AG $\mathcal{F}$ such that $W_{\mathcal{F}}$ cycles if and only if $\mathcal{F}$ is circular. The idea is that on input $\mathbf{t}$, $W_{\mathcal{F}}$ follows all possible paths in the dependency graph[1] of $\mathcal{F}$ for $\mathbf{t}$. Hence, $W_{\mathcal{F}}$ will terminate on $\mathbf{t}$ if and only if this dependency graph is acyclic. This idea is similar in spirit to a result by Maneth and the present author [MN99] where a standard attribute grammar is transformed to a $\mathcal{DTL}$ program such that the latter terminates on every input if and only if the former is non-circular. Here, the complication arises from the fact that we have to deal with extended AGs rather than with standard AGs.

[1] The *dependency graph* $\mathcal{D}_{\mathcal{F}}(\mathbf{t})$ of $\mathcal{F}$ for a derivation tree $\mathbf{t}$ is defined as follows. Its nodes are all $a(\mathbf{n})$, such that $\mathbf{n}$ is a node of $\mathbf{t}$ and a is an attribute of the label of $\mathbf{n}$. Further, there is an edge from $a(\mathbf{n})$ to $b(\mathbf{m})$ if and only if $a(\mathbf{n}) \in W(b(\mathbf{m}))$ (cf. Definition 4.20). Clearly, $\mathcal{F}$ is well-defined on $\mathbf{t}$ if and only if $\mathcal{D}(\mathbf{t})$ contains no cycle. Hence, $\mathcal{F}$ is non-circular if and only if there does not exist a $\mathbf{t}$ such that $\mathcal{D}_{\mathcal{F}}(\mathbf{t})$ is cyclic.

Theorem 4.31 *Deciding non-circularity of extended AGs is in EXPTIME.*

Proof. Let $\mathcal{F}$ be an extended AG with attribute set A and semantic domain D. For ease of exposition we assume that all grammar symbols have all attributes, i.e., for every $X \in N \cup T$, $\mathrm{Inh}(X) \cup \mathrm{Syn}(X) = A$.

We now construct a tree walking automaton $W_{\mathcal{F}}$ such that $W_{\mathcal{F}}$ cycles if and only if $\mathcal{F}$ is circular. Rather than letting $W_{\mathcal{F}}$ work over derivation trees of G, we let it work on the set of all trees over the alphabet $(N \cup T) \times (P \cup T) \times (P \cup \{U\}) \times \{1, \ldots, m\}$ where $m = \max\{|r| \mid X \to r \in P\}$. That is, m denotes the maximal number of positions of a regular expression in a production of P.

The automaton $W_{\mathcal{F}}$ first checks the following. For each node $\mathbf{n}$ of the input tree labeled with (σ, p_1, p_2, i),

1. if $p_1 \in T$ then $\sigma = p_1$ and $\mathbf{n}$ should be a leaf; if $p_1 = X \to r \in P$ then $\sigma = X$ and $\mathbf{n}$ should be derived by p_1;

2. if $p_2 = U$ then $\mathbf{n}$ should be the root; if $p_2 \in P$ then the parent of $\mathbf{n}$ should be derived with p_2; and

3. if the parent $\mathbf{p}$ of $\mathbf{n}$ is derived by $X \to r$, w is the string formed by the children of $\mathbf{p}$, and $\mathbf{n}$ is the j-th child of $\mathbf{p}$, then $\mathrm{pos}_r(j, w) = i$.

The automaton checks this in the following way. It makes a depth first traversal of the tree. At each node $\mathbf{n}$ labeled with (σ, p_1, p_2, i) it can check (1) by first checking whether the current node is a leaf, and if not, by simulating the NFA for r on the children of $\mathbf{n}$ where $p_1 = X \to r$. Only when the NFA accepts it moves to the next node in the depth first traversal. To check (2), $W_{\mathcal{F}}$ makes another depth first traversal of the tree. It first checks whether the root is labeled with $(U, p, U, 1)$. Next, for each internal node $\mathbf{n}$ labeled with (σ, p_1, p_2, i) it checks whether every child of $\mathbf{n}$ has p_1 in the third component of its label. Finally, (3) is checked by making a third depth first traversal through the tree simulating the automaton M_r of Lemma 4.3.

If all this succeeds then $W_{\mathcal{F}}$ nondeterministically walks to a node and chooses an attribute a which it keeps in its state. Now, suppose $W_{\mathcal{F}}$ arrives at a node $\mathbf{n}$ labeled with (X, p_1, p_2, j) with the attribute a in its state. We distinguish two cases.

1. a is a synthesized attribute of X: Let $a(0) := \langle \sigma_0, \ldots, \sigma_{|r|}; (R_d)_{d \in D} \rangle$ be the rule in the context $(p_1, a, 0)$. Then $W_{\mathcal{F}}$ nondeterministically chooses an attribute b in a σ_i and replaces a in its state with b. If $i = 0$ then $W_{\mathcal{F}}$ just stays at the current node. If $i > 0$ then $W_{\mathcal{F}}$ walks nondeterministically to a child of the current node having i as the last component of its label.

2. a is a synthesized attribute of X: Let $a(j) := \langle \sigma_0, \ldots, \sigma_{|r|}; (R_d)_{d \in D} \rangle$ be the rule in the context (p_2, a, j). Then $W_{\mathcal{F}}$ nondeterministically chooses an attribute b in a σ_i and replaces a in its state with b. If $i = 0$ then $W_{\mathcal{F}}$ walks to the parent of $\mathbf{n}$. If $i > 0$ then $W_{\mathcal{F}}$ walks nondeterministically to a sibling of $\mathbf{n}$ having i as the last component of its label (or possibly stays at $\mathbf{n}$ if $i = j$).

Clearly, $W_{\mathcal{F}}$ is cycling if and only if $\mathcal{F}$ is circular. ■

Since deciding non-circularity for standard attribute grammars is also hard for EXPTIME, we obtain that testing whether a nondeterministic tree walking automaton cycles is EXPTIME-complete.

4.4 Expressiveness of extended AGs

In this section we generalize Theorem 3.24 to extended AGs. We start with the easy direction.

Lemma 4.32 *Every query expressible by an extended AG is definable in MSO.*

Proof. Let $\mathcal{F}$ be an extended AG. We say that an arbitrary total valuation v of $\mathbf{t}$ *satisfies* $\mathcal{F}$ if for every node $\mathbf{n}$ of $\mathbf{t}$ and attribute a of the label of $\mathbf{n}$, $v(W(a(\mathbf{n}))) \in R^{a(\mathbf{n})}_{v(a(\mathbf{n}))}$. It follows immediately from the definitions that $\mathcal{F}(\mathbf{t})$ satisfies $\mathcal{F}$. Moreover, $\mathcal{F}(\mathbf{t})$ is the only valuation that satisfies $\mathcal{F}$. Indeed, suppose that v satisfies $\mathcal{F}$. An easy induction on l, using non-circularity, then shows that if $a(\mathbf{n})$ is defined in $\mathcal{F}_l(\mathbf{t})$ then $\mathcal{F}_l(\mathbf{t})(a(\mathbf{n})) = v(a(\mathbf{n}))$.

In MSO we just guess the values of the attributes, verify our guesses and select those nodes for which the result attribute is true. For ease of exposition we assume that all grammar symbols have all attributes, i.e., for every $X \in N \cup T$, $\mathrm{Inh}(X) \cup \mathrm{Syn}(X) = A$. We use set variables to represent the assignment of values: for each function $\alpha : A \to D$, Z_α will contain those nodes $\mathbf{n}$ such that for every attribute $a \in A$, $\mathcal{F}(\mathbf{t})(a(\mathbf{n})) = \alpha(a)$. We then only have to verify that all semantic rules are satisfied under this assignment. This can easily be done since by Lemma 2.7 every regular language can be defined in MSO. The result of the query expressed by $\mathcal{F}$ then consists of the nodes in all the Z_α where $\alpha(\mathit{result}) = 1$. Since for every tree there is only one assignment that satisfies $\mathcal{F}$ we can just existentially quantify over the Z_α.

We omit the formal construction of the MSO formula simulating $\mathcal{F}$ which is straightforward but tedious. ■

To prove the other direction, we show that extended AGs can compute the MSO-equivalence type of the input tree. This computation is similar to the one in the proof of Theorem 3.24. The only complication arises from the fact that trees are now unranked.

Theorem 4.33 *A query is expressible by an extended AG if and only if it is definable in MSO.*

Proof. The only-if direction was already given in Lemma 4.32.

Let $\varphi(x)$ be an MSO formula of quantifier depth k. We will define an extended AG $\mathcal{F}$ expressing the query defined by φ. Define $D = \Phi_k \cup \{0, 1\}$ and $A = \{\mathit{env}, \mathit{sub}, \mathit{result}, \mathrm{lab}\}$, where *env* is inherited for all grammar symbols except for the start symbol for which it is synthesized, and *sub* and *result* are synthesized for all non-terminals and inherited for all terminals. The intended meaning is the following: for a node $\mathbf{n}$ of a tree $\mathbf{t}$,

- $\mathcal{F}(\mathbf{t})(sub(\mathbf{n})) = \tau_k(\mathbf{t_n}, \mathbf{n})$,
- $\mathcal{F}(\mathbf{t})(env(\mathbf{n})) = \tau_k(\overline{\mathbf{t_n}}, \mathbf{n})$, and
- $\mathcal{F}(\mathbf{t})(result(\mathbf{n})) = 1$ if and only if $\mathbf{t} \models \varphi[\mathbf{n}]$.

By Proposition 4.12(1), $\mathbf{t} \models \varphi[\mathbf{n}]$ only depends on $\tau_k(\overline{\mathbf{t_n}}, \mathbf{n})$ and $\tau_k(\mathbf{t_n}, \mathbf{n})$. Hence, $\mathcal{F}(\mathbf{t})(result(\mathbf{n}))$ only depends on the attribute values $\mathcal{F}(\mathbf{t})(env(\mathbf{n}))$ and $\mathcal{F}(\mathbf{t})(sub(\mathbf{n}))$.

The extended AG we will construct works in two passes. In the first bottom-up pass all the *sub* attributes are computed (using the regular languages SUB, defined below); in the subsequent top-down pass all the *env* attributes are computed (using the regular languages ENV, defined below). To initiate the top-down pass we use our convention, that the start symbol cannot appear in the left-hand side of a production. After this second pass, there is enough information at each node $\mathbf{n}$ to decide whether $\mathbf{t} \models \varphi[\mathbf{n}]$.

We now define the regular languages SUB, which we will use to compute τ_k^{MSO}-types of subtrees in a bottom-up fashion. We again abbreviate $\tau_k^{\text{MSO}}(\mathbf{t}, \text{root}(\mathbf{t}))$ by $\tau_k^{\text{MSO}}(\mathbf{t}, \text{root})$. Define for $\theta \in \Phi_k$ and $X \in N$ the language $\text{SUB}(X, \theta)$ over Φ_k as follows:

$$\theta_1 \cdots \theta_n \in \text{SUB}(X, \theta)$$

if there exist trees $\mathbf{t}_1, \ldots, \mathbf{t}_n$ such that for $i = 1, \ldots, n$, $\tau_k^{\text{MSO}}(\mathbf{t}_i, \text{root}) = \theta_i$ and $\tau_k^{\text{MSO}}(X(\mathbf{t}_1, \ldots, \mathbf{t}_n), \text{root}) = \theta$. Similar to the proof of Lemma 4.13 we show that $\text{SUB}(X, \theta)$ is a regular language.

Lemma 4.34 *Let $X \in N$ and $\theta \in \Phi_k$. There exists a DFA $M = (S, \Phi_k, \delta, s_0, F)$ accepting* $\text{SUB}(X, \theta)$.

Proof. Define $M = (S, \Phi_k, \delta, s_0, F)$ as the DFA where $S = \Phi_k \cup \{s_0\}$ and $F = \{\theta\}$; define the transition function as follows: for all $\theta, \theta_1, \theta_2 \in \Phi_k$, $\delta(s_0, \theta) = \tau_k^{\text{MSO}}(X(\mathbf{t}), \text{root})$ with $\tau_k^{\text{MSO}}(\mathbf{t}, \text{root}) = \theta$, and $\delta(\theta_1, \theta) = \theta_2$, whenever there exists a tree $\mathbf{t}$ with an X-labeled node $\mathbf{n}$ of arity n (for some n) such that $\tau_k^{\text{MSO}}(X(\mathbf{t_{n1}}, \ldots, \mathbf{t_{n\mathit{n}-1}}), \text{root}) = \theta_1$, $\tau_k^{\text{MSO}}(\mathbf{t_{n\mathit{n}}}, \mathbf{n}n) = \theta$, and $\tau_k^{\text{MSO}}(\mathbf{t_n}, \mathbf{n}) = \theta_2$. By Proposition 4.12(2), it does not matter which elements in the equivalence classes θ, θ_1, and θ_2 we take. ∎

Note that, $\delta^*(s_0, \theta_1 \cdots \theta_n) = \theta'$ if and only if there exist trees $\mathbf{t}_1, \ldots, \mathbf{t}_n$ such that for $i = 1, \ldots, n$, $\tau_k^{\text{MSO}}(\mathbf{t}_i, \text{root}) = \theta_i$ and $\tau_k^{\text{MSO}}(X(\mathbf{t}_1, \ldots, \mathbf{t}_n), \text{root}) = \theta'$.

Now, define for $\theta \in \Phi_k$ the language $\text{ENV}(\theta)$ over $\Phi_k \cup \{\#\}$ as follows:

$$\bar{\theta} = \theta_0 \theta_1 \cdots \theta_{i-1} \# \theta_i \theta_{i+1} \cdots \theta_n \in \text{ENV}(\theta)$$

iff

- $\theta_j \in \Phi_k$ for $j = 0, \dots, n$, and
- there exists a tree $\mathbf{t}$ with a node $\mathbf{n}$ of arity n (for some n) such that $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \theta_0$, $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_{n}}_i}, \mathbf{n}i) = \theta$, and $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}_j, \mathbf{n}j) = \theta_j$ for $j = 1, \dots, n$.

By Proposition 4.12(3), $\bar{\theta} \in \mathrm{ENV}(\theta)$ only depends on $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$, $\tau_k^{\mathrm{MSO}}(X(\mathbf{t_n}_1, \dots, \mathbf{t_n}_{i-1}), \mathrm{root})$, $\tau_k^{\mathrm{MSO}}(X(\mathbf{t_n}_{i+1} \cdots \mathbf{t_n}_n), \mathrm{root})$, and the label of $\mathbf{n}i$ which in turn only depends on $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}_i, \mathbf{n}i)$. In terms of the automaton M of Lemma 4.34, $\bar{\theta} \in \mathrm{ENV}(\theta)$ only depends on θ_0, $\delta^*(s_0, \theta_1 \cdots \theta_{i-1})$, $\delta^*(s_0, \theta_{i+1} \cdots \theta_n)$, and θ_i. It is, hence, not difficult to construct an automaton accepting $\mathrm{ENV}(\theta)$. Indeed, such an automaton stores θ_0 in its state; then simulates M until it reaches the symbol $\#$; this gives the state $\delta^*(s_0, \theta_1 \cdots \theta_{i-1})$; hereafter M stores θ_i in its state and again simulates M until the end of the string which gives the state $\delta^*(s_0, \theta_{i+1} \cdots \theta_n)$; M then accepts if $\xi_X(\theta_0, \delta^*(s_0, \theta_1 \cdots \theta_{i-1}), \theta_i, \delta^*(s_0, \theta_{i+1} \cdots \theta_n)) = \theta$. Here the function ξ_X is defined as follows. For θ_1, θ_2, θ_3, θ_4, $\theta \in \Phi_k$ and $X \in N$, $\xi_X(\theta_1, \theta_2, \theta_3, \theta_4) = \theta$ if there exists a tree $\mathbf{t}$ with an X-labeled node $\mathbf{n}$ of arity n (for some n) and an $i \in \{1, \dots, n\}$, such that

- $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \theta_1$;
- $\tau_k^{\mathrm{MSO}}(X(\mathbf{t_n}_1 \dots \mathbf{t_n}_{i-1}), \mathrm{root}) = \theta_2$;
- $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}_i, \mathbf{n}i) = \theta_3$;
- $\tau_k^{\mathrm{MSO}}(X(\mathbf{t_n}_{i+1} \dots \mathbf{t_n}_n), \mathrm{root}) = \theta_4$; and
- $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}_i}, \mathbf{n}i) = \theta$.

By Proposition 4.12, for all $X \in N$ and $\theta_1, \theta_2 \in \Phi_k$, if $\theta_1 \neq \theta_2$ then $\mathrm{SUB}(X, \theta_1) \cap \mathrm{SUB}(X, \theta_2) = \emptyset$ and $\mathrm{ENV}(\theta_1) \cap \mathrm{ENV}(\theta_2) = \emptyset$. Also, for all $X \in N$, $\bigcup_{\theta \in \Phi_k} \mathrm{SUB}(X, \theta) = \Phi_k^*$ and $\bigcup_{\theta \in \Phi_k} \mathrm{ENV}(\theta) = \Phi_k^* \# \Phi_k^*$.

We now define the semantic rules of $\mathcal{F}$. For every production $X \to r$, define in the context $(X \to r, sub, 0)$ the rule

$$sub(0) := \langle \sigma_0 = \varepsilon, \sigma_1 = sub, \dots, \sigma_{|r|} = sub; (R_\theta = \mathrm{SUB}(X, \theta))_{\theta \in \Phi_k}, R_0 = \{0, 1\}^* \rangle.$$

As before, the R_d's that are not mentioned are defined as the empty set. For every i such that $r(i) = \sigma$ is a terminal define in the context $(X \to r, sub, i)$ the rule

$$sub(i) := \langle \sigma_0 = \varepsilon, \sigma_1 = \varepsilon, \dots, \sigma_{|r|} = \varepsilon; R_{\theta_\sigma} = \{\varepsilon\}, R_0 = D^* - R_{\theta_\sigma} \rangle \rangle.$$

The above rule just assigns the type $\theta_\sigma = \tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma), \mathrm{root})$ to every non-terminal σ. For $i = 1, \dots, |r|$, define in the context $(X \to r, env, i)$ the rule

$$env(i) := \langle \sigma_0 = env, \sigma_1 = sub, \dots, \sigma_{|r|} = sub; (R_\theta = \mathrm{ENV}(\theta))_{\theta \in \Phi_k}, R_0 = \{0, 1, \#\}^* \rangle.$$

For the start symbol, define in the context $(U \to r, env, 0)$ the rule

$$env(0) := \langle \sigma_0 = \varepsilon, \sigma_1 = \varepsilon, \dots, \sigma_{|r|} = \varepsilon; R_{\theta(U)} = \{\varepsilon\}, R_0 = DD^* \rangle \rangle,$$

where $\theta(U) = \tau_k^{\mathrm{MSO}}\big(\overline{\mathbf{t}(U)}, \mathrm{root}(\mathbf{t}(U))\big)$. Finally, add in the context $(X \to r, \mathit{result}, 0)$ the rule

$$\mathit{result}(0) := \langle \sigma_0 = (\mathit{env}, \mathit{sub}), \sigma_1 = \varepsilon, \dots, \sigma_{|r|} = \varepsilon; R_1, R_0 = D^* - R_1 \rangle,$$

and for every i such that $r(i)$ is a terminal, add in the context $(X \to r, \mathit{result}, i)$ the rule

$$\begin{aligned} \mathit{result}(i) := \langle \sigma_0 = \varepsilon, \sigma_1 = \varepsilon, \dots, \sigma_{i-1} = \varepsilon, \sigma_i = (\mathit{env}, \mathit{sub}),& \\ \sigma_{i+1} = \varepsilon, \dots, \sigma_{|r|} = \varepsilon; R_1, R_0 = (D \cup \{\#\})^* - R_1 \rangle,& \end{aligned}$$

where R_1 consists of those two letter strings $\theta_1\theta_2 \in \Phi_k^2$ for which there exists a tree $\mathbf{t}$ with a node $\mathbf{n}$, with $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \theta_1$, $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathrm{root}) = \theta_2$, and $\mathbf{t} \models \varphi[\mathbf{n}]$. ■

4.5 Optimization

An important research topic in the theory of query languages is that of optimization of queries. This comprises, for example, the detection and elimination of subqueries that always return the empty relation, or more general, the rewriting of queries, stated in a certain formalism, into equivalent ones that can be evaluated more efficiently. The central problem in the case of the latter is, hence, to decide whether the rewritten queries are indeed equivalent to the original ones. In this section we study the complexity of the emptiness and equivalence test of extended AGs. Interestingly, these results will be applied in Chapter 6 to obtain a new upper bound for deciding equivalence of Region Algebra expressions introduced by Consens and Milo [CM98a].

We consider the following problems:

- **Non-emptiness**: Given an extended AG $\mathcal{F}$, does there exists a tree $\mathbf{t}$ and a node $\mathbf{n}$ of $\mathbf{t}$ such that $\mathcal{F}(\mathbf{t})(\mathit{result}(\mathbf{n})) = 1$?
- **Equivalence**: Given two extended AGs $\mathcal{F}_1$ and $\mathcal{F}_2$ over the same grammar, do $\mathcal{F}_1$ and $\mathcal{F}_2$ express the same query?

To show the EXPTIME-hardness for the above decision problems we use a reduction from TWO PERSON CORRIDOR TILING which is known to be complete for EXPTIME (see Chlebus [Chl86]).

For natural numbers n and m we view $\{1, \dots, n\} \times \{1, \dots, m\}$ as a rectangle consisting of m rows of width n. Let T be a finite set of tiles, let $H, V \subseteq T \times T$ be horizontal and vertical constraints, and let $\bar{b} = b_1, \dots, b_n, \bar{t} = t_1, \dots, t_n \in T^n$ be the bottom and the top row. A *corridor tiling* from $\bar{b}$ to $\bar{t}$ is a mapping $\lambda : \{1, \dots, n\} \times \{1, \dots, m\} \to T$, for some natural number m, such that

- the first row is $\bar{b}$, that is, $\lambda(1, 1) = b_1, \dots, \lambda(1, n) = b_n$;
- the m-th row is $\bar{t}$, that is, $\lambda(m, 1) = t_1, \dots, \lambda(m, n) = t_n$;

- for $i = 1, \dots, n-1$ and $j = 1, \dots, m$, $(\lambda(i,j), \lambda(i+1,j)) \in H$; and
- for $i = 1, \dots, n$ and $j = 1, \dots, m-1$, $(\lambda(i,j), \lambda(i,j+1)) \in V$.

In a two person corridor tiling game from $\bar{b}$ to $\bar{t}$, two players, on turn, place tiles row wise from bottom to top, and from left to right in each row. The first player starts and each newly placed tile should be consistent with the tiles already placed. The first player tries to make a corridor tiling from $\bar{b}$ to $\bar{t}$, whereas the second player tries to prevent this. If the first player always can achieve such a tiling no matter how the second player plays, then we say that player one *wins* the corridor game. A player that puts down a tile not consistent with the tiles already placed, immediately looses.

TWO PERSON CORRIDOR TILING is the problem to decide, given a set of tiles T, $H, V \subseteq T \times T$, a sequence of tiles $\bar{b} = b_1, \dots, b_n$ and $\bar{t} = t_1, \dots, t_n \in T^n$, whether player one wins the corridor game.

Lemma 4.35 *Deciding non-emptiness of extended AGs is hard for EXPTIME.*

Proof. The proof is a reduction from TWO PERSON CORRIDOR TILING to non-emptiness of extended AGs. A strategy for player one can be represented by a tree where the nodes are labeled with tiles. Indeed, if we put the rows of a tiling next to each other rather than on top of each other, then every branch, i.e., the sequence of labels from the root to a leaf, of a tree represents a possible tiling. If we forget about the start row $\bar{b}$ for a moment, then the odd depth nodes have no siblings and represent moves of player one and the even depth nodes do have siblings and represent all the choices of player two. A strategy is then winning when every branch is either a corridor tiling or is a tiling where player two made a false move.

The extended AG we construct will only accept trees that correspond to winning strategies for player one. The AG essentially only has to check the horizontal and vertical constraints. Since n, the width of the corridor, is constant, the vertical constraints can be checked by storing at each node the tile carried by its n-th ancestor. The horizontal constraints can be checked for each node by looking at the tile carried by its parent.

Let $(T, H, V, b_1, \dots, b_n, t_1, \dots, t_n)$ be an instance of TWO PERSON CORRIDOR TILING where $T = \{c_1, \dots, c_k\}$. Define $G_{\text{corr}} = (N_{\text{corr}}, T_{\text{corr}}, P_{\text{corr}}, U_{\text{corr}})$ as the ECFG, where the set of terminals T_{corr} contains only the symbol Ξ, and the set of non-terminals N_{corr} consists of all triples $\{(c,i,j) \mid c \in T, i \in \{1,2\}, j \in \{1, \dots, n\}\}$ together with the set $\{b_1, \dots, b_n, t_1, \dots, t_n\}$. If a node is labeled with (c,i,j) then this means that player i has put tile c on the j-th square of the current row. The terminal Ξ functions as an end delimiter, indicating that either the end row $\bar{t}$ has been reached or that player two has put a tile on the board that is inconsistent with the tiles already present. The set of productions of G_{corr} now consists of the following rules:

1. $U_{\text{corr}} \to b_1$;
2. $b_n \to (c_1, 1, 1) + \cdots + (c_k, 1, 1) + t_1$ and $b_i \to b_{i+1}$ for $i = 1, \dots, n-1$;

3. $t_n \to \Xi$ and $t_i \to t_{i+1}$ for $i = 1, \dots, n-1$;

4. $$(c, 1, j) \to (c_1, 2, j+1) \cdots (c_k, 2, j+1),$$

   and
   $$(c, 2, j) \to (c_1, 1, j+1) + \cdots + (c_k, 1, j+1) + \Xi,$$

   for each $c \in P$ and $j \in \{1, \dots, n-1\}$; and

5. $$(c, 1, n) \to (c_1, 2, 1) \cdots (c_k, 2, 1) + t_1,$$

   and
   $$(c, 2, n) \to (c_1, 1, 1) + \cdots + (c_k, 1, 1) + \Xi + t_1,$$

   for each $c \in P$.

If $\mathbf{t}$ is a derivation tree and $\mathbf{n}$ is a node of $\mathbf{t}$, then we call the j-th node on the path from the parent of $\mathbf{n}$ to the root, the j-th *ancestor* of $\mathbf{n}$. If $\mathbf{n}$ is labeled with (c, i, j) then we say that $\mathbf{n}$ is *admissible* if $(c', c) \in V$ where the n-th ancestor of $\mathbf{n}$ contains the tile c' in its label, and, additionally, if $j > 1$, then $(c'', c) \in H$ where $(c'', i'', j-1)$ is the label of the parent of $\mathbf{n}$. Similarly, we say that a node labeled with c_i is *admissible* if $(c', c_i) \in V$ where the n-th ancestor of $\mathbf{n}$ is labeled with c'.

The extended AG $\mathcal{F}$ has attributes $A = \{1, \dots, n, \mathit{local}, \mathit{result}, \mathrm{lab}\}$ and semantic domain $\{0, 1, c_1, \dots, c_k\}$, and works in three passes. In the first top-down pass it defines the inherited attributes $1, \dots, n$ such that for $j = 1, \dots, n$, $\mathcal{F}(\mathbf{t})(j(\mathbf{n}))$ equals the tile in the label of the j-th ancestor of $\mathbf{n}$ for each node $\mathbf{n}$ of $\mathbf{t}$. Next, $\mathcal{F}$ uses these attributes to check local consistency of the tiling. More precisely, the attribute *local* is defined true for a node $\mathbf{n}$ labeled with $(c, 1, j)$ iff $\mathbf{n}$ is admissible; and the attribute *local* is defined true for a node $\mathbf{n}$ labeled with $(c, 2, j)$ iff $\mathbf{n}$ is admissible or $\mathbf{n}$ is not admissible and the only child of $\mathbf{n}$ is labeled with Ξ (the latter captures the intuition that player one wins when player two uses a wrong tile). Finally, $\mathcal{F}$ checks whether all attributes *local* are true by making a bottom-up pass through the tree. If the latter is the case then $\mathcal{F}$ selects the root. Clearly, $\mathcal{F}$ can be constructed in polynomial time and is non-empty if and only if player one wins the corridor tiling game. We omit the formal description of $\mathcal{F}$. ■

Non-emptiness of extended AGs can in fact also be decided in EXPTIME. The proof essentially works as follows. For each extended AG $\mathcal{F}$ we construct an NBTA $T_{\mathcal{F}}$ guessing the attribute values at each node; it then accepts when the *result* attribute of at least one node is true. Since the size of $T_{\mathcal{F}}$ will be exponential in the size of $\mathcal{F}$ and non-emptiness of NBTAs can be checked in polynomial time (see Lemma 4.9), we obtain an EXPTIME algorithm for testing non-emptiness of extended AGs.

Bloem and Engelfriet [BE] already showed that tree automata can guess attribute values of nodes defined by *standard* Boolean-valued attribute grammars on ranked trees. We must extend this technique to unranked trees and automata, and must control the sizes of the NFAs involved in the transition function of the automaton. In particular, we control these sizes by first describing the transition function by nondeterministic two-way automata with a pebble which can be transformed into equivalent one-way nondeterministic automata with only an exponential size increase.

Theorem 4.36 *Deciding non-emptiness of extended AGs is EXPTIME-complete.*

Proof. EXPTIME-hardness has just been shown in Lemma 4.35, so it remains to show that non-emptiness is in EXPTIME.

Let $\mathcal{F}$ be an extended AG over the grammar $G = (N, T, P, U)$. Recall that all regular languages R_d are represented by NFAs. W.l.o.g., we assume that every grammar symbol has all attributes, i.e., for all $X \in N \cup T$, $\mathrm{Inh}(X) \cup \mathrm{Syn}(X) = A$. As mentioned above, we construct an NBTA $T_{\mathcal{F}}$ such that $L(T_{\mathcal{F}}) \neq \emptyset$ if and only if $\mathcal{F}$ is non-empty. The size of $T_{\mathcal{F}}$ will be exponential in the size of $\mathcal{F}$. That is, the set of states of $T_{\mathcal{F}}$ and the NFAs representing transition functions will be exponential in the size of $\mathcal{F}$. By Lemma 4.9, non-emptiness of $T_{\mathcal{F}}$ can be checked in time exponential in the size of $\mathcal{F}$. Hence, the theorem follows.

The automaton $T_{\mathcal{F}}$ essentially guesses the values of the attributes and then verifies whether they satisfy all semantic rules. Therefore, we use as states tuples (α, o, p, i) where $\alpha : A \to D$ is a function, $o \in \{0, 1\}$, $p \in P \cup T$ and $i \in \{1, \dots, s\}$, where $s = \max\{|r| \mid X \to r \in P\}$.

Before explaining the meaning of the states, we introduce the following notions. A *state assignment* for a tree $\mathbf{t}$ is a mapping ρ from the nodes of $\mathbf{t}$ to Q. A state assignment is *valid* if for every node $\mathbf{n}$ of $\mathbf{t}$ of arity n with children $\mathbf{n}_1, \dots, \mathbf{n}_n$, $\rho(\mathbf{n}_1) \cdots \rho(\mathbf{n}_n) \in \delta(\rho(\mathbf{n}), X)$, where $\mathbf{n}$ is labeled with X, and $\rho(\mathrm{root}(\mathbf{t})) \in F$. Clearly, a tree $\mathbf{t}$ is accepted by $T_{\mathcal{F}}$ if and only if there exists a valid state assignment for t.

If in a valid state assignment $T_{\mathcal{F}}$ assigns the state $q = (\alpha, o, p, i)$ to a node $\mathbf{n}$ of an input tree, then

- α represents the values of the attributes of $\mathbf{n}$; i.e., for all $a \in A$, $\mathcal{F}(\mathbf{t})(a(\mathbf{n})) = \alpha(a)$;
- $o = 1$ if and only if a node in the subtree rooted at $\mathbf{n}$ has been selected;
- if $\mathbf{n}$ is an internal node then $p \in P$ and $\mathbf{n}$ is derived by p; if $\mathbf{n}$ is a leaf then $p \in T$ and $\mathbf{n}$ is labeled by p; and
- if $\mathbf{n}$ is the root then $i = 1$; otherwise, if the parent $\mathbf{p}$ of $\mathbf{n}$ is derived by $p' \to r$, $\mathbf{n}$ is the j-th child of $\mathbf{p}$, and w is the string formed by the children of $\mathbf{p}$, then $\mathrm{pos}_r(j, w) = i$.

For a tuple $q = (\alpha, o, p, i) \in Q$, we denote o by $q.o$, α by $q.\alpha$, p by $q.p$, and i by $q.i$. If $p = X \to r \in P$ then we denote X by $p.X$ and r by $p.r$, and if $p \in T$ then we denote p also by $p.X$.

The set of final states F is defined as

$$\{q \in Q \mid q.o = 1,\ q.p.X = U \text{ and } q.i = 1\}.$$

As before, if $\alpha : A \to D$ is a function and $\sigma = a_1 \cdots a_n$ is a sequence of attributes then we denote the string $\alpha(a_1) \cdots \alpha(a_n)$ by $\alpha(\sigma)$.

We define the transition function. For all $\alpha \in A \to D$, $\sigma_1, \sigma_2 \in T$, $o \in \{0, 1\}$, and $i \in \{1, \ldots, s\}$, define

$$\delta\big((\alpha, o, \sigma_1, i), \sigma_2\big) := \begin{cases} \{\varepsilon\} & \text{if } \sigma_1 = \sigma_2 \text{ and } o = \alpha(\mathit{result}) \text{ ;} \\ \emptyset & \text{otherwise.} \end{cases}$$

For all $X \in N$ and $q \in Q$, if $q.X \neq X$ then $\delta(q, X) = \emptyset$; otherwise, if $q.X = X$ then $q_1 \cdots q_n \in \delta(q, X)$ iff

1. $q_1.X \cdots q_n.X \in L(q.r)$;

2. for all $j = 1, \ldots, n$, $\mathrm{pos}_{q.r}(j, q_1.X \cdots q_n.X) = q_j.i$;

3. for every synthesized attribute a of X, defined by the rule

$$\langle \sigma_0, \ldots, \sigma_{|q.r|}; (R_d)_{d \in D} \rangle,$$

   we must have:

$$q.\alpha(\sigma_0) \cdot q_1.\alpha(\sigma_{q_1.i}) \cdots q_n.\alpha(\sigma_{q_n.i}) \in R_{q.\alpha(a)};$$

4. for all $j = 1, \ldots, n$, for every inherited attribute a of $q_j.X$, defined by the rule

$$\langle \sigma_0, \ldots, \sigma_{|q.r|}; (R_d)_{d \in D} \rangle,$$

   we must have:

$$q.\alpha(\sigma_0) \cdot q_1.\alpha(\sigma_{q_1.i}) \cdots q_{j-1}.\alpha(\sigma_{q_{j-1}.i}) \# q_j.\alpha(\sigma_{q_j.i})$$
$$q_{j+1}.\alpha(\sigma_{q_{j+1}.i}) \cdots q_n.\alpha(\sigma_{q_n.i}) \in R_{q.\alpha(a)};$$

   and

5. $q.o = 1$ if and only if $q.\alpha(\mathit{result}) = 1$ or there exists a $j \in \{1, \ldots, n\}$ such that $q_j.o = 1$.

We now show that conditions (1–5) are regular. Moreover, they can be defined by NFAs whose size is exponential in the size of $\mathcal{F}$. The result then follows since the size of the NFA computing the intersection of a constant number of NFAs is polynomial in the sizes of those NFAs.

- (1) and (2) are checked by the the NFA $M_{q.r}$ obtained from $q.r$ as described in Lemma 4.3 whose size is linear in r.

- For (3), we describe a two-way nondeterministic automaton M_1. By Lemma 2.10, M_1 can be transformed into an equivalent one-way NFA whose size is exponential in M_1. M_1 makes one pass through the input string for every synthesized attribute a of X simulating the NFA for $R_{q.\alpha(a)}$. If the latter accepts then M_1 walks back to the beginning of the input string and treats the next synthesized attribute or accepts if all synthesized attributes have been accounted for; M_1 rejects if the NFA for $R_{q.\alpha(a)}$ rejects. This needs only a linear number of states in the sizes of the NFAs representing transition functions and the set of attributes.

- For (4), we describe a two-way nondeterministic automaton M_2 with a pebble. By Lemma 2.10, M_2 can be transformed into an equivalent one-way NFA whose size is exponential in M_2. M_2 now successively puts its pebble on each position of the input string. Suppose M_2 has just put the pebble on position j, then, for every inherited attribute a of $q_j.X$, M_2 walks back to the beginning of the input string and simulates the NFA for $R_{q_i.\alpha(a)}$, pretending to read # the moment it encounters the pebble. If the NFA for $R_{q_i.\alpha(a)}$ accepts, then M_2 walks back to the beginning of the input string and treats the next inherited attribute of $q_i.X$, or, if all inherited attributes of $q_i.X$ have been considered, moves the pebble to position $j+1$ and repeats the same procedure. If M_2 has put its pebble on all positions it accepts. This needs only a number of states linear in the size of the NFAs representing transition functions and the set of attributes.

- (5) is clearly regular.

This concludes the proof of the theorem. ■

Let us now turn to the equivalence problem. This problem is actually polynomial-time equivalent to the complement of the non-emptiness problem (i.e., the emptiness problem), and hence it is also EXPTIME-complete. Indeed, $\mathcal{F}$ expresses the constant empty query if and only if it is equivalent to a trivial extended AG that expresses this query, and conversely, we can easily test if $\mathcal{F}_1$ and $\mathcal{F}_2$ express the same query by constructing an extended AG that first runs $\mathcal{F}_1$ and $\mathcal{F}_2$ independently, and then defines the value of *result* of a node to be 0 iff the values of *result* for $\mathcal{F}_1$ and $\mathcal{F}_2$ on that node agree. This gives the following theorem.

Theorem 4.37 *Deciding equivalence of extended AGs is EXPTIME-complete.*

In Chapter 6 we use the above result to drastically improve the known upper bound on the complexity of the equivalence problem of Region Algebra expressions.

4.6 Relational extended AGs

In this section we define relational extended AGs which can be viewed as extensions of the relational BAGs studied in Chapter 3. The main difference with the extended AGs studied before (to which we refer by functional extended AGs) is that we now associate *one* regular language R with each production, rather than with each position in a production and for each attribute. Specifically, we show that relational extended AGs express the same class of unary queries as extended AGs.

An attribute grammar vocabulary is now just a tuple (D, A, Att), where D is a finite semantic domain, A is a finite set of attributes, and Att is a function from $N \cup T$ to the powerset of A assigning to each grammar symbol a set of attributes. A *relational extended AG* $\mathcal{F}$ now associates to each production $p = X \to r$ a semantic rule $\langle \sigma_0, \ldots, \sigma_{|r|}; R_p \rangle$, where for $i \in \{0, \ldots, |r|\}$, σ_i is a sequence of attributes of $p(i)$ and R_p is a regular language over D. Let $\mathbf{t}$ be a derivation tree, $\mathbf{n}$ a node of $\mathbf{t}$ with children $\mathbf{n}_1, \ldots, \mathbf{n}_m$ derived by p, and let for $l \in \{1, \ldots, m\}$, $j_l = \mathrm{pos}_r(l, w)$, where w is the string formed by the labels of the children of $\mathbf{n}$. Then define $W(\mathbf{n})$ as the sequence $\sigma_0(\mathbf{n}) \cdot \sigma_{j_1}(\mathbf{n}_1) \cdots \sigma_{j_m}(\mathbf{n}_m)$. A *valuation of* $\mathbf{t}$ is again a function that maps each pair $(\mathbf{n}, a)$, where $\mathbf{n}$ is a node in $\mathbf{t}$ and a is an attribute of the label of $\mathbf{n}$, to an element of D, and that maps for every $\mathbf{n}$, $v((lab, \mathbf{n}))$ to the label of $\mathbf{n}$. A valuation v of $\mathbf{t}$ is said to *satisfy* $\mathcal{F}$ if $v(W(\mathbf{n})) \in R_p$ for every $p \in P$ and every internal node $\mathbf{n}$ derived by p.

A relational extended AG $\mathcal{F}$ can express queries in various ways. We consider two natural ones.

Definition 4.38 (i) A query $\mathcal{Q}$ is expressed *existentially* by a relational extended AG $\mathcal{F}$ if for every $\mathbf{t}$

$$\mathcal{Q}(\mathbf{t}) = \{\mathbf{n} \mid \text{there exists a valuation } v \text{ that satisfies } \mathcal{F} \text{ such that } v(\mathit{result}(\mathbf{n})) = 1\};$$

(ii) A query $\mathcal{Q}$ is expressed *universally* by a relational extended AG $\mathcal{F}$ if for every $\mathbf{t}$

$$\mathcal{Q}(\mathbf{t}) = \{\mathbf{n} \mid \text{for every valuation } v \text{ that satisfies } \mathcal{F},\ v(\mathit{result}(\mathbf{n})) = 1\}.$$

We give an example of the just introduced notions.

Example 4.39 Consider the ECFG of Example 4.18. Let $\mathcal{Q}$ be the query that selects every other poem. The following relational extended AG expresses $\mathcal{Q}$ existentially and universally. If p is the production DB $\to$ Poem$^+$, then define its associated rule as

$$\langle \sigma_0 = \varepsilon, \sigma_1 = \mathit{result}; R_p = (10)^*(1 + \varepsilon) \rangle,$$

here $A = \{\mathit{result}, \mathrm{lab}\}$, $D = \{0, 1\}$ and *result* is an attribute of Poem.

Note that this query $\mathcal{Q}$ is also expressed by a functional extended AG with $A = \{\mathit{result}, \mathrm{lab}\}$, $D = \{0, 1, \mathrm{Poem}, \mathrm{DB}\}$, $\mathrm{Inh}(\mathrm{Poem}) = \{\mathit{result}\}$, and $\mathrm{Syn}(\mathrm{Poem}) = \emptyset$. Define in the context $(p, \mathit{result}, 1)$ the semantic rule

$$\mathit{result}(1) := \langle \sigma_0 = \varepsilon, \sigma_1 = \mathit{lab}; R_1 = (\mathrm{Poem}\mathrm{Poem})^* \# \mathrm{Poem}^* \rangle.$$

■

Functional and relational extended AGs are equally expressive as is shown in the next theorem. We just show that every query expressed existentially or universally by a relational extended AG can be defined in MSO and that every MSO definable query can be expressed both by an existential and a universal AG. The result then follows from Theorem 4.33.

Theorem 4.40 *The class of queries expressed existentially (universally) by relational extended AGs coincides with the class of queries expressed by extended AGs.*

Proof. The semantics of a relational extended AG can readily be defined in MSO. For ease of exposition we assume that all grammar symbols have all attributes, i.e., for every $X \in N \cup T$, $\mathrm{Inh}(X) \cup \mathrm{Syn}(X) = A$. We again use set variables Z_α, with α a function from A to D, to represent assignments of values to attributes. As in the proof of Lemma 4.32, we can construct an MSO formula $\psi((Z_\alpha)_{\alpha\in A\to D})$ such that whenever $\mathbf{t} \models \psi((Z_\alpha)_{\alpha\in A\to D})$ then

- the sets $(Z_\alpha)_{\alpha\in A\to D}$ are pairwise disjoint,
- $\bigcup_\alpha Z_\alpha = \mathrm{dom}(\mathbf{t})$, and
- the valuation v defined as, $v(a(\mathbf{n})) = \alpha(a)$ with $\mathbf{n} \in Z_\alpha$, satisfies $\mathcal{F}$.

The formula ψ just verifies the semantic rules of $\mathcal{F}$; and, since these are regular languages, this can be easily done in MSO. The following formulas then define the query expressed existentially respectively universally by $\mathcal{F}$

$$(\exists Z_\alpha)_{\alpha\in A\to D}\left(\psi((Z_\alpha)_{\alpha\in A\to D}) \wedge \bigvee \{Z_\alpha(x) \mid \alpha(\mathit{result}) = 1\}\right),$$

$$(\forall Z_\alpha)_{\alpha\in A\to D}\left(\psi((Z_\alpha)_{\alpha\in A\to D}) \to \bigvee \{Z_\alpha(x) \mid \alpha(\mathit{result}) = 1\}\right).$$

By Theorem 4.33, these MSO formulas can be transformed into an equivalent extended AG.

For the other direction, by Theorem 4.33, it suffices to show that any MSO definable query can be expressed by a relational extended AG under both the existential and the universal semantics. Let $\varphi(x)$ be an MSO formula of quantifier depth k. We now define a relational extended AG $\mathcal{F}$ that expresses φ under both the existential and the universal semantics. Again this relational extended AG just computes the $\equiv_k^{\mathrm{MSO}}$-type of the input tree. We write $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root})$ for $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$.

Define $A = \{\mathit{env}, \mathit{sub}, \mathit{result}, \mathit{lab}\}$ and $D = \Phi_k \cup \{0, 1\}$. To every production $p = X \to r$ we associate the rule

$$\langle \sigma_0 = (\mathit{env}, \mathit{sub}, \mathit{result}, \mathit{lab}), \dots, \sigma_{|r|} = (\mathit{env}, \mathit{sub}, \mathit{result}); R_p \rangle,$$

where the string language R_p is defined as follows: $\overline{\theta_0}\theta_0 o_0 \ell_0 \cdots \overline{\theta_n}\theta_n o_n \ell_n \in R_p$ if

1. $\overline{\theta_i}, \theta_i \in \Phi_k$, $o_i \in \{0, 1\}$, and $\ell_i \in N \cup T$ for $i = 1, \dots, n$;
2. for $i = 1, \dots, n$, if $\ell_i \in T$ then $\theta_i = \tau_k^{\mathrm{MSO}}(\mathbf{t}(\ell_i), \mathrm{root})$;

3. if $X = U$ then $\overline{\theta_0} = \tau_k^{\mathrm{MSO}}(\mathbf{t}(U), \mathrm{root})$;

4. there exists a tree $\mathbf{t}$ with a node $\mathbf{n}$ with children $\mathbf{n}_1, \ldots, \mathbf{n}_n$ such that $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathrm{root}) = \theta_0$, $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \overline{\theta_0}$, and for $i = 1, \ldots, n$, $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}_i}, \mathrm{root}) = \theta_i$ and $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}_i}}, \mathbf{n}) = \overline{\theta_i}$;

5. for $i = 0, \ldots, n$, $o_i = 1$ if and only if there exists a tree $\mathbf{t}$ with a node $\mathbf{n}$ such that $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \overline{\theta_i}$, $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathrm{root}) = \theta_i$, and $\mathbf{t} \models \varphi[\mathbf{n}]$;

We now construct a two-way deterministic string automaton B with one pebble accepting R_p. By Proposition 2.10, R_p is regular. For steps (1–3,5), B just makes one pass through the input string. For step (4), B first simulates the automaton $M = (Q, \Phi_k, \delta, s_0, F)$ for $\mathrm{SUB}(\ell_0, \theta_0)$ of Lemma 4.34 on $\theta_1 \cdots \theta_n$. Hereafter it checks the consistency of $\overline{\theta_0}$, and $\overline{\theta_1}, \ldots, \overline{\theta_n}$. Note that for every $i = 1, \ldots, n$, by Proposition 4.12(3), $\overline{\theta_i}$ depends only on $\overline{\theta_0}$, θ_i, $\delta^*(s_0, \theta_1 \cdots \theta_{i-1})$ and $\delta^*(s_0, \theta_{i+1} \cdots \theta_n)$, where δ is the transition function of M. Hence, B remembers $\overline{\theta_0}$ in its state and then successively puts its pebble on each input tuple. If the pebble lays on the i-th tuple then M computes $\delta^*(s_0, \theta_1 \cdots \theta_{i-1})$ and $\delta^*(s_0, \theta_{i+1} \cdots \theta_n)$, whereafter it returns to the pebble and checks whether $\xi_{\ell_0}(\overline{\theta_0}, \delta^*(s_0, \theta_1 \cdots \theta_{i-1}), \theta_i, \delta^*(s_0, \theta_{i+1} \cdots \theta_n)) = \overline{\theta_i}$ (ξ_{ℓ_0} is the function defined in the proof of Theorem 4.33).

It remains to show that for each tree only one valuation exists satisfying $\mathcal{F}$. Using Proposition 4.12(2), a simple induction on the height of nodes in the tree shows that $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n}) = \theta$ whenever $v(sub(\mathbf{n})) = \theta$ for a valuation v satisfying $\mathcal{F}$. An induction on the depth of nodes in the tree, using the above and Proposition 4.12(3), then shows that $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \theta$ whenever $v(env(\mathbf{n})) = \theta$ for a valuation v satisfying $\mathcal{F}$.

This concludes the proof of the theorem. ■

5

Query Automata

In this chapter we focus on the natural and well-studied computation model of tree automata to compute unary queries. Specifically, we define a query automaton (QA) as a deterministic two-way finite automaton over trees that has the ability to select nodes depending on the state and the label at those nodes. We study QAs over ranked as well as over unranked trees. First, we characterize the expressiveness of the different formalisms as the unary MSO definable queries. Surprisingly, in contrast to the ranked case, special stay transitions have to be added to QAs over unranked trees to capture MSO.

More concretely, we show that query automata can compute, for some fixed k, the type $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{n})$ for each node $\mathbf{n}$ of the input tree. In Chapter 3, we constructed a BAG $\mathcal{B}$ computing the $\equiv_k^{\mathrm{MSO}}$-types in the following way. In the first bottom-up pass $\mathcal{B}$ computes for every node $\mathbf{n}$ the type $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$ and *stores* it in the attributes at $\mathbf{n}$. In the next top-down pass, $\mathcal{B}$ uses this information to compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ for each $\mathbf{n}$, whereby $\mathcal{B}$ has enough information to compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{n})$. A query automaton, unfortunately, cannot mimic this procedure directly as it cannot store information at specific nodes. Nevertheless, by employing some kind of pebbling technique and a surprising lemma on two-way string automata due to Hopcroft and Ullman, we show that query automata in fact can compute all unary queries defined by MSO formulas.

Next, we establish the complexity of the non-emptiness and equivalence problem of query automata to be complete for EXPTIME. We conclude this chapter by considering nondeterministic one-way query automata which can express queries in an existentially and a universally manner. In particular, we show that both these semantics capture exactly the queries definable in MSO.

Proviso 5.1 In this chapter, whenever we say query we always mean “unary query”.

5.1 Query automata on strings

To warm up, we start with query automata on strings. These are simply two-way deterministic automata extended with a selection function. This approach allows us to introduce some important proof techniques in an easy setting which then later will be generalized to obtain our main results. In particular, we recall the important notion of behavior functions and reprove a surprising lemma on two-way automata by Hopcroft and Ullman.

We next define two-way automata over strings with begin- and endmarkers. Even though we already informally used their nondeterministic counterparts (even with a pebble) in previous chapters, we need a concrete definition in this chapter as we want to show that their behavior can be defined in MSO. In the rest of this section, when we feed a string $w \in \Sigma^*$ to a two-way automaton, and only then, we always assume the first symbol is the beginmarker b and the last symbol is the endmarker e. Moreover, these symbols do not occur in Σ. Hence, no intermediate position of any string carries a b or an e.

Definition 5.2 A *two-way deterministic finite automaton* (2DFA) is a tuple $M = (S, \Sigma, s_0, \delta, F, L, R)$, where

- S is a finite set of states;
- s_0 is the initial state;
- F is the set of final states,
- L and R are disjoint subsets of $S \times (\Sigma \cup \{b, e\})$ (left and right moves), such that for all $s \in S$, $(s, b) \notin L$ and $(s, e) \notin R$; and
- δ consists of the transition functions $\delta_{\leftarrow}$ and $\delta_{\rightarrow}$; in particular, $\delta_{\leftarrow} : L \to S$ is the transition function for left-moves and $\delta_{\rightarrow} : R \to S$ is the transition function for right-moves.

The conditions $(s, b) \notin L$ and $(s, e) \notin R$ merely state that the automaton cannot fall off the input string. In the following we will no longer mention explicitly the sets L and R in the definition of M.

A *configuration* of M is a member of $S \times \mathbb{N}$, i.e., a pair consisting of a state and a position. A *run* is a sequence of configurations, i.e., an element of $(S \times \mathbb{N})^*$. For a string w, the *run of M on w* is the sequence $(s_1, j_1) \ldots (s_m, j_m)$ such that for all $i = 1, \ldots, m$,

- $j_i \in \{1, \ldots, |w|\}$;
- if $(s_i, w_{j_i}) \in L$ then $s_{i+1} = \delta_{\leftarrow}(s_i, w_{j_i})$ and $j_{i+1} = j_i - 1$; and
- if $(s_i, w_{j_i}) \in R$ then $s_{i+1} = \delta_{\rightarrow}(s_i, w_{j_i})$ and $j_{i+1} = j_i + 1$.

The run is *accepting* if $j_1 = 1$, s_1 is the initial state, $s_m \in F$ and there is no transition possible from (s_m, w_{j_m}).

We will only consider 2DFAs that always halt. This is a decidable property. Indeed, as we will show in the proof of Theorem 5.8, the behavior of a 2DFA M can be defined in MSO. It is then not difficult to write an MSO sentence that is satisfiable iff M does not terminate on at least one input string. It is well known that satisfiability of MSO is decidable [Tho97b]. Clearly, any 2DFA can be modified such that it always halts at the endmarker. For convenience, we will assume each 2DFA is as such. A query automaton is now just a 2DFA extended with a selection function:

Definition 5.3 A *query automaton M on strings* (QA^{string}) is a tuple $(S, \Sigma, s_0, \delta, F, \lambda)$, where $(S, \Sigma, s_0, \delta, F)$ is a 2DFA, and λ is a mapping $\lambda : S \times \Sigma \to \{0, 1\}$.

We say that M *selects* position $i \in \{1, \dots, |w|\}$ if the run $(s_0, j_0), \dots, (s_m, j_m)$ of M on w is accepting and $\lambda(s_l, w_{j_l}) = 1$ for an $l \in \{0, \dots, m\}$ with $j_i = l$. That is, i is selected by M if M selects i at least once; M does not need to select i every time it visits this position. In particular, when the run is not accepting, no position is selected. The *query expressed* by M on w is defined as $M(w) := \{i \in \{1, \dots, |w|\} \mid M \text{ selects } i\}$.

Remark 5.4 Although 2DFAs are equivalent to one-way DFAs (see, e.g., Shepherdson [She59] or Hopcroft and Ullman [HU79]), not all QA^{string}s are equivalent to a QA^{string} that can move in only one direction. Consider for example queries of the following kind: *select the first and last symbol if the string contains the letter σ*. This query is not expressible by a QA^{string} that only moves in one direction. Indeed, when started on the beginmarker, the one-way query automaton already has to decide whether it should select without having seen the input. The same holds when it is started on the endmarker and it only can move from right to left. ■

We illustrate the previous definitions with an example.

Example 5.5 We give an example of a QA^{string} expressing the query: *select every position labeled with σ occurring on an odd position when counting from right to left starting at the right end of the input string.* Define $M = (S, \Sigma, s_0, \delta, F, \lambda)$ with

- $\Sigma = \{\sigma, \sigma'\}$;
- $S = \{s_0, s_1, s_2\}$;
- $F = \{s_1, s_2\}$;
- $R = \{s_0\} \times (\Sigma \cup \{b\})$;
- $L = \{s_1, s_2\} \times (\Sigma \cup \{e\})$;
- $\delta_\rightarrow(s_0, b) = \delta_\rightarrow(s_0, \sigma) = \delta_\rightarrow(s_0, \sigma') = s_0$;
- $\delta_\leftarrow(s_0, e) = s_1$; $\delta_\leftarrow(s_1, \sigma) = \delta_\leftarrow(s_1, \sigma') = s_2$; $\delta_\leftarrow(s_2, \sigma) = \delta_\leftarrow(s_2, \sigma') = s_1$; and
- for all $s \in S$ and $a \in \Sigma$, $\lambda(s, a) = 1$ iff $s = s_1$ and $a = \sigma$.

The automaton operates as follows. First it walks to the endmarker using state s_0. Hereafter, it returns to the beginmarker alternating between the states s_1 and s_2. A position is assigned the state s_1 (s_2) if it occurs on an odd (even) position when counting from the endmarker (endmarker not included). The run on input $w = b\sigma'\sigma\sigma\sigma' e$ is the sequence

$$(s_0,1)(s_0,2)(s_0,3)(s_0,4)(s_0,5)(s_0,6)(s_1,5)(s_2,4)(s_1,3)(s_2,2)(s_1,1).$$

Hence, only position 3 is selected.

This query automaton does not end at the endmarker. However, it can easily be modified to do so. ■

Before generalizing Büchi's Theorem to query automata, we define two-way deterministic finite automata that output at each position one symbol of a fixed alphabet rather than just 0 or 1 as is the case for query automata. Such automata will turn out useful in the proof of Theorem 5.8 and will be essential for the capturing of MSO by query automata on unranked trees in Section 5.3.

Definition 5.6 A *generalized string query automaton* (GSQA) M is a tuple

$$(S, \Sigma, s_0, \delta, F, \lambda, \Gamma),$$

where $(S, \Sigma, s_0, \delta, F)$ is a 2DFA, Γ is a finite output alphabet, and λ is a function from $S \times \Sigma$ to $\Gamma \cup \{0\}$. We always assume $0 \notin \Gamma$.

We will only consider GSQA that output at each position of the input string exactly one Γ-symbol different from 0 and which always halt. Therefore, for each position i of a string w, we denote by $M(w,i)$ the unique symbol output by M at position i. By $M(w)$ we denote the string $M(w,1)\cdots M(w,|w|)$. Let f be a length preserving function from Σ^* to Γ^*. We say that f is *computed* by a GSQA M if $M(w) = f(w)$ for all strings w.

The condition that a GSQA outputs exactly one Γ-symbol different from 0 at each position is not essential for the results in this paper. We could also just have taken $M(w,i)$ as the last Γ-symbol different from 0 output at position i. The former automata are just easier to work with.

Example 5.7 We modify the QA^{string} of Example 5.5 into a generalized query automaton. To this end, we redefine λ as the function $\lambda : S \times \Sigma \to \{\sigma, \sigma', *, 0\}$ as follows:

$$\begin{array}{ll} \lambda(s_0,\sigma) = 0; & \lambda(s_0,\sigma') = 0; \\ \lambda(s_1,\sigma) = *; & \lambda(s_1,\sigma') = \sigma'; \\ \lambda(s_2,\sigma) = \sigma; \text{ and} & \lambda(s_2,\sigma') = \sigma'. \end{array}$$

This automaton just copies the input string, but replaces every symbol σ with $*$ when it occurs on an odd position when counting from right to left from the endmarker. Thus, $M(b\sigma'\sigma\sigma\sigma' e) = b\sigma' * \sigma\sigma' e$. ■

We generalize Büchi's Theorem to query automata. In this proof we will introduce the concept of behavior function[1] which will play a major role in Sections 5.2, 5.3, and 5.4. Additionally, we use a remarkable lemma due to Hopcroft and Ullman [HU67] on two way automata which will turn out to be crucial in Sections 5.2 and 5.3.

Theorem 5.8 *A query is computable by a QA^{string} if and only it is definable in MSO.*

Proof. Let $M = (S, \Sigma, s_0, \delta, F, \lambda)$ be a QA^{string}. We will now construct an MSO formula $\varphi(x)$ that defines the query computed by M.

In the case of a one-way DFA M', the state assumed by M' at each position of the input string completely determined the behavior of M'. Accordingly, we simulated M' in the proof of Theorem 2.7, by simply guessing this state assignment. Now, we do not only have to describe the behavior of a two-way automaton, but we also have to know which positions it selects. Therefore, we define the following partial functions for M on a string w. If $i \in \{1, \ldots, |w|\}$ then the behavior function $f^{\leftarrow}_{w_1\cdots w_i} : S \to S$ is defined as

$$f^{\leftarrow}_{w_1\cdots w_i}(s) := \begin{cases} s & \text{if } (s, w_i) \in R; \\ s' & \text{if } (s, w_i) \in L \text{ and whenever } M \text{ starts} \\ & \text{its computation on } w \text{ at position } i \\ & \text{in state } s \text{ then } s' \text{ is the first state in} \\ & \text{which it returns at } i. \end{cases}$$

We need one more notion. For each $i = 1, \ldots, |w|$, the set of states *assumed* by M at i is defined as $\mathrm{Assumed}(w, i) := \{s_l \mid l \in \{1, \ldots, m\} \text{ and } j_l = i\}$ with $(s_1, j_1) \ldots (s_m, j_m)$ the run of M on w.

For each position i of the input string w the formula φ now guesses the function $f^{\leftarrow}_{w_1\cdots w_i}$, the set $\mathrm{Assumed}(w, i)$, and the first state in which M reaches i, denoted by $\mathrm{first}(w, i)$. Formally, the formula guesses sets $Z_{f,B,s}$ for all partial functions $f : S \to S$, sets $B \subseteq S$, and $s \in S$, with the intended meaning: $i \in Z_{f,B,s}$ iff $f = f^{\leftarrow}_{w_1\cdots w_i}$, $B = \mathrm{Assumed}(w, i)$ and $s = \mathrm{first}(w, i)$. Note that the number of sets $Z_{f,B,s}$ is bounded, independently of w. The correctness of these guesses is easily verified in FO since they are determined by local consistency checks only. To see this we introduce the following definitions. For each partial function $f : S \to S$ and state $s \in S$, $\mathrm{States}(f, s)$ is the smallest set containing s and if $s' \in \mathrm{States}(f, s)$ then $f(s') \in \mathrm{States}(f, s)$. That is, if M has reached position i in state s then $\mathrm{States}(f^{\leftarrow}_{w_1\cdots w_i}, s)$ is the set of states in which M visits position i before making a right move at i. There are now two possibilities. Either there exists a state $s' \in \mathrm{States}(f^{\leftarrow}_{w_1\cdots w_i}, s)$ with $f^{\leftarrow}_{w_1\cdots w_i}(s') = s'$ or there does not. The second case indicates that M starts to cycle, while the first case means that M makes its next right move at position i in state s'. Hence, we define $\mathrm{right}(f, s) = s'$ with $s' \in \mathrm{States}(f, s)$ and $f(s') = s'$ if such an s' exists. Otherwise, $\mathrm{right}(f, s)$ is undefined. We now have enough terminology to show that the consistency checks only depend on local information.

[1] We note that Shepherdson [She59] already used behavior functions to simulate two-way automata by one-way ones.

1. $\text{first}(w,1) = s_0$ and $f^{\leftarrow}_{w_1}(s) = s$ for all $s \in S$ (recall that w_1 is the beginmarker);
2. for $i = 1, \ldots, |w|-1$, $f^{\leftarrow}_{w_1\cdots w_{i+1}}$ only depends on $f^{\leftarrow}_{w_1\cdots w_i}$, w_i and w_{i+1}, and $\text{first}(w,i+1)$ only depends on $\text{first}(w,i)$, w_i and $f^{\leftarrow}_{w_1\cdots w_i}$. Specifically, for $i = 1, \ldots, |w|-1$, for all $s \in S$,
$$f^{\leftarrow}_{w_1\cdots w_{i+1}}(s) = \begin{cases} s & \text{if } (s, w_{i+1}) \in R \\ \delta_{\rightarrow}\left(\text{right}\left(f^{\leftarrow}_{w_1\cdots w_i}, \delta_{\leftarrow}(s, w_{i+1})\right), w_i\right) & \text{otherwise.} \end{cases}$$
We use the convention that $f^{\leftarrow}_{w_1\cdots w_{i+1}}(s)$ is undefined whenever $\text{right}(f^{\leftarrow}_{w_1\cdots w_i}, \delta_{\leftarrow}(s, w_{i+1}))$ or $\delta_{\rightarrow}$ is undefined. Further,
$$\text{first}(w, i+1) = \delta_{\rightarrow}(\text{right}(f^{\leftarrow}_{w_1\cdots w_i}, \text{first}(w,i), w_i);$$
3. $\text{Assumed}(w,|w|)$ only depends on $\text{first}(w,|w|)$ and $f^{\leftarrow}_{w_1\cdots w_{|w|}}$. Specifically,
$$\text{Assumed}(w,|w|) = \text{States}(f^{\leftarrow}_{w_1\cdots w_{|w|}}, \text{first}(w,|w|));$$
and
4. for $i = 1, \ldots, |w|-1$, $\text{Assumed}(w,i)$ only depends on $f^{\leftarrow}_{w_1\cdots w_i}$, w_{i+1}, $\text{first}(w,i)$, and $\text{Assumed}(w,i+1)$. Specifically, for $i = 1, \ldots, |w|-1$,
$$\begin{aligned}\text{Assumed}(w,i) = {} & \text{States}(f^{\leftarrow}_{w_1\cdots w_i}, \text{first}(w,i)) \\ & \cup \bigcup \{\text{States}(f^{\leftarrow}_{w_1\cdots w_i}, s) \mid \exists s' \in \text{Assumed}(w,i+1) \wedge \delta_{\leftarrow}(s', w_{i+1}) = s\}.\end{aligned}$$

The above conditions uniquely determine $\text{first}(w,i)$, $f^{\leftarrow}_{w_1\cdots w_i}$, and $\text{Assumed}(w,i)$, for each i. In particular, the states first and the functions $f^{\leftarrow}$ are fixed from left to right, whereafter the sets Assumed are fixed from right to left. Clearly, these conditions can be checked in FO. Finally, φ verifies whether M halts in an accepting state (this only depends on $\text{Assumed}(w,|w|)$) and, if so, selects those positions i that are selected by M which now only depend on the Bs.

Conversely, let $\varphi(x)$ be an MSO formula of quantifier depth k. We will describe an automaton N computing the query defined by φ. In particular, N computes $\tau_k^{\text{MSO}}(w,i)$ for every position i of the input string w, which, by Proposition 2.8, only depends on $\tau_k^{\text{MSO}}(w_1\cdots w_i, i)$ and $\tau_k^{\text{MSO}}(w_i \cdots w_{|w|}, 1)$.

We start with the (one-way) DFA M_1 of Lemma 2.9 to compute $\tau_k^{\text{MSO}}(w_1\cdots w_i, i)$ and its right-to-left variant M_2 to compute $\tau_k^{\text{MSO}}(w_i \cdots w_{|w|}, 1)$. A powerful and surprising lemma by Hopcroft and Ullman [HU67, AHU69] allows us to combine M_1 and M_2 into an automaton N that does exactly what we want. Adapted to our setting, the lemma says the following:

Lemma 5.9 *Let $M_1 = (P, \Sigma, \delta_1, p_0, F_1)$ be a left-to-right deterministic automaton on strings and let $M_2 = (Q, \Sigma, \delta_2, q_0, F_2)$ be a right-to-left one. There exists a generalized query automaton A that outputs, at each position of the input string, the pair (p,q) of states that M_1 and M_2 take at this position, respectively. That is, on input w, A outputs for position i the pair $(\delta_1^*(p_0, w_1\cdots w_i), \delta_2^*(q_0, w_{|w|}\cdots w_i))$.*

The result now readily follows since N can first simulate M_1 and M_2 as stated in Lemma 5.9 and then selects every position i for which it would output the pair (θ_1, θ_2) such that there exists a string v and a position j with $\tau_k^{\mathrm{MSO}}(v_1 \cdots v_j, j) = \theta_1$, $\tau_k^{\mathrm{MSO}}(v_j \cdots v_{|v|}, 1) = \theta_2$, and $v \models \varphi[j]$.

For the sake of completeness and since Lemma 5.9 will be used again later on, we sketch a proof of it based on the survey paper of Engelfriet [Eng79].

The automaton A first computes $\delta_1^*(p_0, w)$ by walking to the right end of w while simulating M_1. When it reaches the endmarker it outputs the pair

$$(\delta_1^*(p_0, w), \delta_2(q_0, w_{|w|})),$$

reverses direction and starts to walk back to the left endmarker of w while simulating M_2. The difficulty, however, is to maintain $\delta_1^*(p_0, w_1 \cdots w_i)$, for each position i. We will describe a general method for doing this. Therefore, let $p = \delta_1^*(p_0, w_1 \cdots w_i)$ and assume that A has output the pair (p, q) at the i-th position of w. We now show how A computes $\delta_1^*(p_0, w_1 \cdots w_{i-1})$ from p. Suppose, $\{p' \mid \delta_1(p', w_i) = p\} = \{p_1, \ldots, p_k\}$. That is, $\{p_1, \ldots, p_k\}$ is the set of states from which A could have reached p by reading w_i. If $k = 1$ then there is no problem. Hence, assume $k \geq 2$. Then A simulates M_1 backwards from each state in $\{p_1, \ldots, p_k\}$ simultaneously. That is, when it arrives at position j it knows for each $p_l \in \{p_1, \ldots, p_k\}$ the set of states $\gamma(p_l, w_{j+1} \cdots w_{i-1}) = \{p' \mid \delta_1^*(p', w_{j+1} \cdots w_{i-1}) = p_l\}$. Note that these γ-sets are pairwise disjoint.

This computation continues until one of the two following conditions occurs.

1. If at position j all γ-sets become empty except one (say $\gamma(p_l, w_{j+1} \cdots w_{i-1})$), then p_l is the state we were looking for.

2. If A arrives at the beginmarker then the required state is the one whose γ-set contains the start state.

Now A "knows" the correct state p_l of M_1 at position i. The only remaining problem is that it is now at some position j and has to find its way back to position i. By the construction above, one step before A found out about p_l (at position $j + 1$) there had been at least 2 different sets of states from which M_1 reaches p at position i (after reading w_i). The key idea is that i is exactly the position where two computations of M_1 that start at position $j + 1$ in two states from two of these sets flow together into the same state (in this case the state p). Hence, on its way to the left, A always remembers two states from different γ sets from the position before (right) and starts its way back to position i by simulating the behavior of M_1 from position $j + 1$ beginning with these two states. ■

We note that Lemma 5.9 is also used extensively by Engelfriet and Hoogeboom [EH99] to prove connections between MSO definable string transductions and deterministic two-way finite state transducers.

5.2 Query automata on ranked trees

After the excursion on strings, we turn to ranked trees. Specifically, we define query automata for ranked trees (QA^r) simply as two-way deterministic tree automata extended with a selection function. We will show that such automata express exactly the queries definable in MSO.

Before we start we make the following proviso.

Proviso 5.10 All definitions in this section are for Σ-trees with rank at most m, for some fixed natural number m.

We borrow some notation from Brüggemann-Klein, Murata and Wood [BKMW98] for the following definitions.

5.2.1 Two-way tree automata

We use the definition of a two-way tree automaton by Moriya [Mor94].

Definition 5.11 A *two-way deterministic tree automaton* ($2\mathrm{DTA}^r$) is a tuple

$$A = (Q, \Sigma, F, s, \delta),$$

where Q is a finite set of states, $F \subseteq Q$ is the set of final states and $s \in Q$ is the initial state. There are disjoint subsets U and D of $Q \times \Sigma$ (U corresponds to up transitions and D to down transitions) such that $\delta_{\mathrm{leaf}} : D \to Q$ is the transition function for leaves,[2] $\delta_{\mathrm{root}} : U \to Q$ is the transition function for the root, $\delta_{\uparrow} : U^* \to Q$ is the transition function for up transitions, and $\delta_{\downarrow} : D \times \{1, \ldots, m\} \to Q^*$ is the transition function for down-transitions. For each $i \leq m$, $\delta_{\downarrow}(q, a, i)$ is a string of length i.

We introduced the disjoint sets U and D to avoid collision between up and down transitions. We come back to this after having defined the computation of $2\mathrm{DTA}^r$s. To this end, we introduce the following notions. A *cut* of $\mathbf{t}$ is a subset of $\mathrm{Nodes}(\mathbf{t})$, that contains exactly one node of each path from the root to a leaf. A *configuration* of A on $\mathbf{t}$ is a mapping $c : C \to Q$ from a cut C of $\mathbf{t}$ to the set of states of A.

If $\mathbf{n}$ is a node of $\mathbf{t}$, then $\mathrm{children}(\mathbf{n})$ denotes the set of children of $\mathbf{n}$. Let $c : C \to Q$ be a configuration. If $\mathrm{children}(\mathbf{n}) \subseteq C$, then formally $c(\mathrm{children}(\mathbf{n}))$ is a subset of Q. We overload this notation so that $c(\mathrm{children}(\mathbf{n}))$ also denotes the sequence of states in Q which arises from the order of $\mathbf{n}$'s children in $\mathbf{t}$. If $\mathrm{children}(\mathbf{n}) = \mathbf{n}_1, \ldots, \mathbf{n}_n$ (in order) then define $\pi(c, \mathbf{n})$ as the sequence $(c(\mathbf{n}_1), \mathrm{lab}_{\mathbf{t}}(\mathbf{n}_1)) \cdots (c(\mathbf{n}_n), \mathrm{lab}_{\mathbf{t}}(\mathbf{n}_n))$.

The automaton A operating on $\mathbf{t}$ makes a *transition* between two configurations $c_1 : C_1 \to Q$ and $c_2 : C_2 \to Q$, denoted by $c_1 \to c_2$, iff it makes an up transition, a down transition, a leaf transition or a root transition:

1. A makes an *up transition* from c_1 to c_2 if there is a node $\mathbf{n}$ such that

   (i) $\mathrm{children}(\mathbf{n}) \subseteq C_1$,

[2] Note that leaves can also take part in up transitions.

(ii) $C_2 = (C_1 - \text{children}(\mathbf{n})) \cup \{\mathbf{n}\}$,

(iii) $\delta_\uparrow(\pi(c_1, \mathbf{n})) = c_2(\mathbf{n})$, and

(iv) c_1 is identical to c_2 on $C_1 \cap C_2$.

2. A makes a *down transition* from c_1 to c_2 if there is a node $\mathbf{n}$ such that

   (i) $\mathbf{n} \in C_1$,

   (ii) $C_2 = (C_1 - \{\mathbf{n}\}) \cup \text{children}(\mathbf{n})$,

   (iii) $\delta_\downarrow(c_1(\mathbf{n}), \text{lab}_\mathbf{t}(\mathbf{n}), \mathit{arity}(\mathbf{n})) = c_2(\text{children}(\mathbf{n}))$, and

   (iv) c_1 is identical to c_2 on $C_1 \cap C_2$.

3. A makes a *leaf transition* from c_1 to c_2 if there is a leaf node $\mathbf{n}$ such that

   (i) $\mathbf{n} \in C_1$,

   (ii) $C_2 = C_1$,

   (iii) $\delta_{\text{leaf}}(c_1(\mathbf{n}), \text{lab}_\mathbf{t}(\mathbf{n})) = c_2(\mathbf{n})$, and

   (iv) c_1 is identical to c_2 on $C_1 - \{v\}$.

4. A makes a *root transition* from c_1 to c_2 if

   (i) $C_1 = \{\text{root}(\mathbf{t})\}$,

   (ii) $C_2 = C_1$, and

   (iii) $\delta_{\text{root}}(c_1(\text{root}(\mathbf{t})), \text{lab}_\mathbf{t}(\text{root}(\mathbf{t}))) = c_2(\text{root}(\mathbf{t}))$.

The configuration $c : C \to Q$ with $c(\text{root}(\mathbf{t})) = s$ (and hence $C = \{\text{root}(\mathbf{t})\}$) is the *start configuration*. Any configuration with $c(\text{root}(\mathbf{t})) \in F$ is an *accepting configuration*. This means that a 2DTA^r starts at the root and returns there to accept the tree. A *run* is a sequence of configurations $c_1, \dots, c_n$, $n \geq 1$, such that $c_1 \to \cdots \to c_n$ and c_1 is the start configuration. A run is *maximal* if there does not exist a c such that $c_n \to c$. A run is *accepting* if it is maximal and if c_n is an accepting configuration.

It should be noted that, although there are usually many different runs for the same tree, for all nodes the sequence of states in which they are visited is the same in all these runs. Indeed, the disjointness of $Q \times \Sigma$ into U and D makes sure that a node labeled with a certain state cannot make an up transition in one run and a down transition in another run. Therefore it is justified to consider the behavior of these automata as deterministic. For this reason, we will also refer to *the* run of A on a tree rather then the more correct *a* run of A.

A 2DTA^r A now *accepts* a tree $\mathbf{t}$ if the run of A on $\mathbf{t}$ is accepting; *A accepts a tree language $\mathcal{T}$* if it accepts exactly every tree in $\mathcal{T}$.

Note that A can run forever on an input tree $\mathbf{t}$. In this case the run of A on $\mathbf{t}$ is infinite and therefore not accepting. We, however, will only consider automata that always terminate on every input. This is a decidable subclass. Indeed, later we show that the behavior of A can be defined in MSO. One then can construct an

MSO sentence that is satisfiable iff A does not terminate on at least one tree. Since satisfiability of MSO sentences is decidable [Tho97b], it follows that deciding whether a 2DTA^r halts on every input is also decidable.

We illustrate the above definitions with an example.

Example 5.12 Consider trees that represent Boolean circuits consisting of AND and OR gates having two inputs and one output. The represented Boolean function is evaluated from the leaves to the root. We now define a 2DTA^r accepting all trees that evaluate to a 1. For ease of exposition, we only consider full binary trees that indeed represent Boolean circuits. That is, internal nodes are labeled with AND and OR, and leaves are labeled with 0 and 1. Define the 2DTA^r

$$A = (Q, \Sigma = \{\mathrm{AND}, \mathrm{OR}, 0, 1\}, F, s, \delta),$$

with $Q = \{s, u\} \cup \{0,1\}^2$; $D = \{s\} \times \Sigma$; $U = \{u, (0,0), (0,1), (1,0), (1,1)\} \times \Sigma$; $F = \{1\}$; and for $\sigma \in \Sigma, i, j, i_1, j_1, i_2, j_2 \in \{0,1\}$, and $\mathrm{op}, \mathrm{op}_1, \mathrm{op}_2 \in \{\mathrm{AND}, \mathrm{OR}\}$, define

1. $\delta_\downarrow(s, \sigma, 2) = (s, s)$;
2. $\delta_{\mathrm{leaf}}(s, \sigma) = u$;
3. $\delta_\uparrow((u, i), (u, j)) = (i, j)$;
4. $\delta_\uparrow(((i_1, j_1), \mathrm{op}_1), ((i_2, j_2), \mathrm{op}_2)) = (i_1 \ \mathrm{op}_1 \ j_1, i_2 \ \mathrm{op}_2 \ j_2)$; and
5. $\delta_{\mathrm{root}}((i, j), \mathrm{op}) = i \ \mathrm{op} \ j$.

Here, i AND j and i OR j define the standard Boolean functions. The automaton first walks to the leaves (1); at the leaves it changes state s into state u (2); hereafter, A assigns the state (i, j) to nodes of height 1 where i is the label of their first child and j is the label of their second child (3); from then on, A assigns to each inner node the pair $(i, j) \in \{0,1\}^2$, where i and j are the result of the evaluation of the left and right subtree of this node (4); finally, the root is assigned the value of the tree (5). ■

To obtain a more uniform two-way tree automaton we have let all transitions depend on the state and the label of the nodes from where this transition originates. That is, up transitions depend on the labels and the states of the children of the node the automaton heads to, while a down transition depends on the state and the label of the parent node. Up transitions of the one-way tree automata defined in Chapter 2 differ from these in that they depend on the states at the children and the label of the parent. Each two-way tree automaton can readily simulate a one-way one. Indeed, let $B = (Q_B, \Sigma, \delta_B, F_B)$ be a bottom-up deterministic tree automaton. For ease of exposition assume all transitions of B are defined. Then define the two-way automaton A simulating B as follows. First, A runs to the leaves of the input tree $\mathbf{t}$. From thereon it uses functions $f : \Sigma \to Q_B$ as states with the following intended meaning: A assigns f to a node $\mathbf{n}$ such that $\delta_B^*(\mathbf{t_n}) = f(\sigma)$ whenever $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}) = \sigma$. Thus to each leaf A

assigns the state f with $f(\sigma) = \delta_B(\sigma)$ for each $\sigma \in \Sigma$, and up transitions are defined as follows: $\delta(f_1, \sigma_1, \ldots, f_n, \sigma_n) = f$ with $f(\sigma) = \delta_B(f_1(\sigma_1), \ldots, f_n(\sigma_n), \sigma)$ for every $\sigma \in \Sigma$. Furthermore, A accepts when $f(\mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))) \in F_B$ where f is the state assigned to the root.

5.2.2 Query automata

A ranked query automaton is simply a two-way deterministic tree automaton over ranked trees extended with a selection function.

Definition 5.13 A *query automaton* (QA^r) is a tuple $A = (Q, \Sigma, F, s, \delta, \lambda)$, where $(Q, \Sigma, F, s, \delta)$ is a $\mathrm{2DTA}^r$ and λ is a function from $Q \times \Sigma$ to $\{0, 1\}$; λ is the *selection function*.

We define the semantics of a QA^r A. If $\mathbf{t}$ is a tree and $\mathbf{n}$ is a node of $\mathbf{t}$, then A *selects* $\mathbf{n}$ *in configuration* $c : C \to Q$, if $\mathbf{n} \in C$ and $\lambda(c(\mathbf{n}), \mathrm{lab}_{\mathbf{t}}(\mathbf{n})) = 1$. A *selects* $\mathbf{n}$ if the run $c_1, \ldots, c_n$ of A on $\mathbf{t}$ is accepting and if there is an $i \in \{1, \ldots, n\}$ such that $\mathbf{n}$ is selected by A in c_i. The *query expressed by* A is defined as $A(\mathbf{t}) := \{\mathbf{n} \in \mathrm{Nodes}(\mathbf{t}) \mid A \text{ selects } \mathbf{n}\}$. *$A$ accepts the tree language* that is accepted by the underlying tree automaton.

Example 5.14 An automaton selecting all nodes evaluating to 1 in a Boolean circuit, is obtained from the automaton of Example 5.12 by changing F to Q and adding the selection function λ defined by, for $i, j \in \{0, 1\}$ and $\mathrm{op} \in \{\mathrm{AND}, \mathrm{OR}\}$, $\lambda((i, j), \mathrm{op}) := 1$ iff i op $j = 1$. ∎

Remark 5.15 Although two-way deterministic tree automata are equivalent to deterministic bottom-up tree automata (see, e.g., Moriya [Mor94]), not all *query* automata are equivalent to deterministic *query* automata that are only top-down or only bottom-up. Consider for example queries of the following kind: *select the root if there is a leaf labeled with σ* and *select all leaves if the root is labeled with σ*. In Section 5.5, we show that adding nondeterminism to bottom-up or top-down query automata suffices to capture the expressiveness of two-way query automata.

In other words: two-way and one-way query automata are equivalent with respect to defining tree languages but not with respect to expressing queries. ∎

In preparation of the proof of Lemma 5.17, we extend the notion of a behavior function used in the proof of Theorem 5.8 to two-way tree automata.

Definition 5.16 Let A be a QA^r with state set Q. The *behavior function* $f_{\mathbf{t}}^A : Q \to Q$ of A on a tree $\mathbf{t}$ is the partial function defined as follows

$$f_{\mathbf{t}}^A(q) := \begin{cases} q & \text{if } (q, \mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))) \text{ is in } U \\ q' & \text{if } (q, \mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))) \text{ is in } D \text{ and} \\ & \text{whenever } A \text{ starts its computation} \\ & \text{on } \mathbf{t} \text{ in state } q \text{ then } q' \text{ is the first} \\ & \text{state in which it returns at } \mathrm{root}(\mathbf{t}). \end{cases}$$

If $c_1, \ldots, c_n$ is the run of A on $\mathbf{t}$, then the set of states A *assumes* at a node $\mathbf{n}$ is defined as $\mathrm{Assumed}^A(\mathbf{t}, \mathbf{n}) := \{c_i(\mathbf{n}) \mid i \in \{1, \ldots, n\}$ and $\mathbf{n}$ belongs to the cut of $c_i\}$.

Sometimes, when A is clear from the context we just write $f_{\mathbf{t}}$ and $\mathrm{Assumed}(\mathbf{t}, \mathbf{n})$ rather than $f_{\mathbf{t}}^A$ and $\mathrm{Assumed}^A(\mathbf{t}, \mathbf{n})$.

We introduce some more notation. If $f_1, \ldots, f_n$ are partial functions from Q to Q and $q \in Q$, then the set of states reachable from q by using the functions $f_1, \ldots, f_n$, denoted by $\mathrm{States}(f_1, \ldots, f_n, q)$, is the smallest set of states containing q and closed under applications of every f_i. We define $\mathrm{up}(f, q)$ as the unique state q' in $\mathrm{States}(f, q)$ for which $f(q') = q'$. If there is no such state, then $\mathrm{up}(f, q)$ is undefined. Intuitively, when f corresponds to the behavior function $f_{\mathbf{t_n}}^A$, then $\mathrm{up}(f, q)$ is the state in which A makes an up transition at $\mathbf{n}$ when started at $\mathbf{n}$ in state q.

5.2.3 Expressiveness

We characterize the expressiveness of ranked query automata in terms of MSO. First, we show how the query computed by a ranked query automaton can be defined in MSO.

Lemma 5.17 *Every query computed by a ranked query automaton can be defined in MSO.*

Proof. Let $A = (Q, \Sigma, F, s, \delta, \lambda)$ be a QA^r. Like in the proof of Theorem 5.8, we will construct an MSO formula that guesses sets and then verifies the consistency of these sets. We make use of the sets $Z_{f,B}$, where f is a partial mapping $f : Q \to Q$ and $B \subseteq Q$. On input $\mathbf{t}$ they have the following intended meaning: a node $\mathbf{n} \in Z_{f,B}$ iff $f = f_{\mathbf{t_n}}^A$, and $B = \mathrm{Assumed}^A(\mathbf{t}, \mathbf{n})$. Again, like in the proof of Theorem 5.8, the correctness of these guesses is easily verified in FO since they are determined by local conditions only. Indeed,

1. the behavior function of every leaf node only depends on its label;

2. the behavior function of every non-leaf node $\mathbf{n}$ with n children only depends on $f_{\mathbf{t_{n1}}}^A, \ldots, f_{\mathbf{t_{nn}}}^A$, $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}1), \ldots, \mathrm{lab}_{\mathbf{t}}(\mathbf{n}n)$, and $\mathrm{lab}_{\mathbf{t}}(\mathbf{n})$. Specifically, let $\mathbf{n}$ be a node of $\mathbf{t}$ of arity n. Then, for every $q \in Q$,

$$f_{\mathbf{t_n}}^A(q) := \begin{cases} q & \text{if } (q, \mathrm{lab}_{\mathbf{t}}(\mathbf{n})) \in U \\ q' & \text{if } (q, \mathrm{lab}_{\mathbf{t}}(\mathbf{n})) \notin U,\ \delta_{\downarrow}(q, \mathrm{lab}_{\mathbf{t}}(\mathbf{n})) = (q_1, \ldots, q_n) \text{ and} \\ & \delta_{\uparrow}(\mathrm{up}(f_{\mathbf{t_{n1}}}^A, q_1), \mathrm{lab}_{\mathbf{t}}(\mathbf{n}1), \ldots, \mathrm{up}(f_{\mathbf{t_{nn}}}^A, q_n), \mathrm{lab}_{\mathbf{t}}(\mathbf{n}n)) = q'; \end{cases} \quad (*)$$

   We use the convention that $f_{\mathbf{t_n}}^A(q)$ is undefined whenever in $(*)$, $\delta_{\downarrow}$, $\delta_{\uparrow}$, or one of the $\mathrm{up}(f_{\mathbf{t_{ni}}}^A, q_i)$ is undefined;

3. $\mathrm{Assumed}^A(\mathbf{t}, \mathrm{root}(\mathbf{t}))$ only depends on $f_{\mathbf{t}}^A$, the label of the root, and the start state. Specifically, $\mathrm{Assumed}^A(\mathbf{t}, \mathrm{root}(\mathbf{t})) = \mathrm{States}(f_{\mathbf{t}}^A, \delta_{\mathrm{root}}(\cdot, \mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))), s)$;[3]

[3] Here, for each $\sigma \in \Sigma$, $\delta_{\mathrm{root}}(\cdot, \sigma)$ is the function mapping each q to $\delta_{\mathrm{root}}(q, \sigma)$.

4. for every non-root node $\mathbf{n}i$, $\text{Assumed}^A(\mathbf{t}, \mathbf{n}i)$ only depends on $\text{Assumed}^A(\mathbf{t}, \mathbf{n})$, the label of $\mathbf{n}$, and the behavior function of $\mathbf{n}i$. Specifically, let $\mathbf{n}$ be a node with n children. Then[4]

$$\text{Assumed}^A(\mathbf{t}, \mathbf{n}i) = \bigcup\{\text{States}(f^A_{\mathbf{t}_{\mathbf{n}i}}, q) \mid \exists q' \in \text{Assumed}^A(\mathbf{t}, \mathbf{n}) \wedge \delta_\downarrow(q, \text{lab}_{\mathbf{t}}(\mathbf{n})).i = q\}.$$

The above conditions uniquely determine the behavior functions and the sets Assumed. In particular, the behavior functions are fixed bottom-up, whereafter the sets Assumed are fixed top-down. Furthermore, the above conditions can clearly be expressed in FO. Together with the verification of these conditions, the formula verifies whether A halts in an accepting state (this only depends on $f^A_{\mathbf{t}}$) and, if so, selects those nodes that are visited in a selecting state, which now only depends on the B's. ■

For the proof of the other direction we construct an automaton computing, for some fixed k, the type $\tau^{\text{MSO}}_k(\mathbf{t}, \mathbf{n})$ for each node $\mathbf{n}$ of the input tree. Recall that, in Chapter 3, we constructed a BAG $\mathcal{B}$ computing the $\equiv^{\text{MSO}}_k$-types in the following way. In the first bottom-up pass $\mathcal{B}$ computes for every node $\mathbf{n}$ the type $\tau^{\text{MSO}}_k(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$ and *stores* it in the attributes at $\mathbf{n}$. In the next top-down pass, $\mathcal{B}$ uses this information to compute $\tau^{\text{MSO}}_k(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ for each $\mathbf{n}$, whereby $\mathcal{B}$ has enough information to compute $\tau^{\text{MSO}}_k(\mathbf{t}, \mathbf{n})$. A query automaton, unfortunately, cannot mimic this procedure directly as it cannot store information at specific nodes. Nevertheless, by employing some kind of pebbling technique and Lemma 5.9, we will show in the next theorem that query automata in fact can compute each query expressed by an MSO formula.

Theorem 5.18 *A query is expressible by a* QA^r *if and only if it is definable in MSO.*

Proof. The only-if direction is given in Lemma 5.17.

For notational simplicity we describe the proof of the other direction only for trees of rank 2. The proof of the general case is a straightforward generalization.

Let $\varphi(x)$ be an MSO-formula of quantifier depth k. We describe a QA^r that computes the query which is defined by φ. The automaton has to find out, for each vertex $\mathbf{n}$ of a tree $\mathbf{t}$, whether $\mathbf{t} \models \varphi(\mathbf{n})$. This depends only on $\tau^{\text{MSO}}_k(\mathbf{t}, \mathbf{n})$, the $\equiv^{\text{MSO}}_k$-type of the structure $(\mathbf{t}, \mathbf{n})$. By Proposition 2.14(1), $\tau^{\text{MSO}}_k(\mathbf{t}, \mathbf{n})$ is uniquely determined by $\tau^{\text{MSO}}_k(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$ and $\tau^{\text{MSO}}_k(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$. Hence, the QA^r only has to compute $\tau^{\text{MSO}}_k(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$ and $\tau^{\text{MSO}}_k(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ to decide whether $\mathbf{n}$ should be selected. We first describe an algorithm that computes these $\equiv^{\text{MSO}}_k$-types for every node of a *complete* binary tree. Next we explain how this algorithm can be translated to a QA^r. Finally, we sketch how the QA^r has to be modified to deal also with possibly non-complete trees.

(i) The underlying algorithm of the QA^r for complete binary trees is outlined in Figure 5.1.

(ii) All objects computed by the algorithm in Figure 5.1 are of bounded size, depending only on φ and not on the size of $\mathbf{t}$. Hence, a QA^r can store them in its

[4]We denote by $\delta_\downarrow(q, \text{lab}_{\mathbf{t}}(\mathbf{n})).i$ the i-th entry of $\delta_\downarrow(q, \text{lab}_{\mathbf{t}}(\mathbf{n}))$.

Input: $\mathbf{t}$
Compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathrm{root}(\mathbf{t})}}, \mathrm{root}(\mathbf{t}))$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$,
for $i := 0$ **to** depth of $\mathbf{t}$ **do**
begin
  for all vertices $\mathbf{n}$ of level i **do**
  begin
    % the root is level 0
    % $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ *has already been computed*
    % now compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$
    1. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}1}, \mathbf{n}1)$
       Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}2}, \mathbf{n}2)$
    2. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$ from $\mathrm{lab}_{\mathbf{t}}(\mathbf{n})$, $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}1}, \mathbf{n}1)$,
       and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}2}, \mathbf{n}2)$
    3. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{n})$ from $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$
    4. Deduce from $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{n})$ whether $\mathbf{t} \models \varphi(\mathbf{n})$ holds
       If so, select $\mathbf{n}$
    5. Compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}1}}, \mathbf{n}1)$ and $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}2}}, \mathbf{n}2)$ from
         $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$, $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}1}, \mathbf{n}1)$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}2}, \mathbf{n}2)$
  end
end

Figure 5.1: The algorithm for computing the query defined by $\varphi(x)$ over complete binary trees.

state. To simulate the outer loop of the algorithm a QA^rcan proceed in cuts that consist of all vertices of level i.[5] It follows from Proposition 2.14 that steps 2, 3 and 5 only involve the application of fixed finite functions. Hence steps 2–5 can be performed in parallel at all vertices of the same depth i. Step 1 is the only one that involves non-local computation. We next discuss this step.

The type $\tau_k^{\mathrm{MSO}}(\mathbf{t_m},\mathbf{m})$ can be computed in a bottom-up fashion for a subtree $\mathbf{t_m}$ of $\mathbf{t}$. Indeed, we just use the automaton of Lemma 2.15. As discussed at the end of Section 5.2 this automaton can be readily simulated by a two-way automaton that now starts at $\mathbf{m}$. The problem, however, is to detect when the root of the subtree $\mathbf{t_m}$, i.e., the starting point, is reached.

Our QA^r remembers this starting point by a kind of pebbling trick. To compute $\tau_k^{\mathrm{MSO}}(\mathbf{t_{n1}},\mathbf{n}1)$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t_{n2}},\mathbf{n}2)$ it first makes a down transition: to $\mathbf{n}2$ the automaton assigns a U-state which keeps $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}},\mathbf{n})$ in mind and waits until the computation in the left subtree has finished. The QA^r then goes down to the leaves of $\mathbf{t_{n1}}$ and computes $\tau_k^{\mathrm{MSO}}(\mathbf{t_{n1}},\mathbf{n}1)$ in one bottom-up traversal as described above. It "recognizes" that the subtree-evaluation is finished by meeting the U-state at $v2$. Next it makes an up transition, followed by a down transition. Hereafter, $v1$ has a U-state which contains $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}},\mathbf{n})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t_{n1}},\mathbf{n}1)$, and waits for the termination of the evaluation of the right subtree which is done analogously to the case of the left subtree. This finishes the description of the QA^r for the case of complete binary trees.

(*iii*) We now explain how a QA^r can deal with non-complete binary trees. We cannot use the above described pebbling trick when a node $\mathbf{n}$ has only one child. To remedy this we make use of Lemma 5.9. If a node $\mathbf{n}$ has only one child (while its parent node $\mathbf{p}$ has at least two children), then we view the part of the tree between $\mathbf{n}$ and the first descendant $\mathbf{m}$ of $\mathbf{n}$ with more than one child (or no child) as a string, where $\mathbf{p}$ and $\mathbf{m}$ play the role of begin and endmarker, respectively. Since $\mathbf{m}$ has more than one child or is a leaf, we can compute $\tau_k^{\mathrm{MSO}}(\mathbf{t_m},\mathbf{m})$ as described in (*ii*).

Consider the deterministic string automata M_1 and M_2, where, for all vertices $\mathbf{c}$ between $\mathbf{p}$ and $\mathbf{m}$, M_1 computes $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_c}},\mathbf{c})$ starting from $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}},\mathbf{n})$ and M_2 computes $\tau_k^{\mathrm{MSO}}(\mathbf{t_c},\mathbf{c})$ starting from $\tau_k^{\mathrm{MSO}}(\mathbf{t_m},\mathbf{m})$. On the string between $\mathbf{p}$ and $\mathbf{m}$ the QA^r then behaves as the two-way string automaton that combines the two automata M_1 and M_2 as specified in Lemma 5.9.

Hereafter, the automaton walks to $\mathbf{m}$ arriving there in state $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_m}},\mathbf{m})$ and continues.

If we consider m-ary trees, then in step (1) we just need to compute $\tau_k^{\mathrm{MSO}}(\mathbf{t_{n1}})$, ... , $\tau_k^{\mathrm{MSO}}(\mathbf{t_{n\mathit{n}}})$, where n is the arity of $\mathbf{n}$. Because $n \leq m$ and m is fixed, we can compute these one after the other. Steps (2–5) again consist of the application of fixed finite functions. ■

[5]It should be noted that we are describing here only one special run of the automaton. But, as mentioned before, all possible runs are equivalent.

5.3 Query automata on unranked trees

We next turn to query automata over unranked trees. Surprisingly, the equivalence with MSO obtained in the previous section does not generalize smoothly to unranked trees. Indeed, to obtain the expressiveness of MSO we have to add so-called "stay transitions" to our model. We start with the obvious generalization of query automata to unranked trees.

5.3.1 First approach

A first approach to define query automata for unranked trees is to add a selection function to the two-way deterministic automata for unranked trees as defined by Brüggemann-Klein, Murata and Wood [BKMW98]. However, it will turn out that these automata cannot even express all first-order logic definable queries.

Definition 5.19 [BKMW98] A *two-way deterministic tree automaton over unranked trees* (2DTA^u) is a tuple $A = (Q, \Sigma, F, s, \delta)$, where Q, F, s, U, D, δ_{leaf} and δ_{root} are as in Definition 5.11. The transition function for up transitions is now of the form $\delta_\uparrow : U^* \to Q$, and the transition function for down transitions is of the form $\delta_\downarrow : D \times \mathbb{N} \to Q^*$. For each $(q, a) \in D$, $L_\downarrow(q, a) := \{\delta_\downarrow(q, a, i) \mid i \in \mathbb{N}\}$ is regular; for each $j \in \mathbb{N}$, $\delta_\downarrow(q, a, j)$ must be a string of length j; and for each $q \in Q$ the language $L_\uparrow(q) := \{w \in U^* \mid \delta_\uparrow(w) = q\}$ must be regular. To assure determinism, we require that $L_\uparrow(q) \cap L_\uparrow(q') = \emptyset$ for all $q \neq q'$.

The definitions of configuration, leaf, root, up and down transitions,[6] run, and accepting run carry over from QA^{r}s.

We argue that each transition in a run of the automaton takes linear time. To this end we elaborate on the structure of the regular languages $L_\downarrow(q, a)$. Each such language contains for each $n \in \mathbb{N}$, at most one string of length n. Shallit [Sha92] has shown that such languages can be described by finite unions of regular expressions of the form xy^*z, where x, y, and z are strings. Hence, we can assume all languages $L_\downarrow(q, a)$ are represented by such languages. Suppose the automaton makes a down transition in state q at a node $\mathbf{n}$ with label a and arity n. Then all we have to do is look up in $L_\downarrow(q, a)$ the string of length n, if it exists. This can clearly be done in time linear in the size of the input tree when $L_\downarrow(q, a)$ is represented by finite unions of regular expressions of the above simple form. We represent all regular languages $L_\uparrow(q)$ by deterministic finite acceptors. Suppose in a configuration c the automaton makes an up transition at the children of a node $\mathbf{n}$. Then we just have to check for each q whether $\pi(c, \mathbf{n})$ (cf. Section 5.2.1) belongs to $L_\uparrow(q)$. This can also be done in time linear in the size of the input tree.

Each two-way tree automaton can readily simulate a one-way one. Indeed, let $B = (Q_B, \Sigma, \delta_B, F_B)$ be a bottom-up deterministic tree automaton over unranked trees. For ease of exposition assume all transitions of B are defined, that is, for each

[6] Note that $\delta_\downarrow$ is uniquely determined by the regular languages $L_\downarrow(q, a)$.

$q_1 \cdots q_n \in Q_B^*$ there exists $q \in Q_B$ and $\sigma \in \Sigma$ such that $q_1 \cdots q_n \in \delta_B(q, \sigma)$. Then define the two-way automaton $A = (Q, \Sigma, F, s, \delta)$ simulating B as follows. First, A runs to the leaves of the input tree $\mathbf{t}$. From thereon it uses functions $f : \Sigma \to Q_B$ as states with the following intended meaning: A assigns f to a node $\mathbf{n}$ such that $\delta_B^*(\mathbf{t_n}) = f(\sigma)$ whenever $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}) = \sigma$. Thus to each leaf A assigns the state f with $f(\sigma) = q$ such that $\varepsilon \in \delta_B(q, \sigma)$ for each $\sigma \in \Sigma$. Up transitions are defined as follows: $(f_1, \sigma_1) \cdots (f_n, \sigma_n) \in L_\uparrow(f)$ whenever for every $\sigma \in \Sigma$ we have that $f_1(\sigma_1) \cdots f_n(\sigma_n) \in \delta_B(q, \sigma)$, where $f(\sigma) = q$. Clearly, each $L_\uparrow(f)$ is regular. Furthermore, A accepts when $f(\mathrm{lab}_{\mathbf{t}}(\mathrm{root}(\mathbf{t}))) \in F_B$ where f is the state assigned to the root.

Definition 5.20 A *query automaton* (QA^u) is a tuple $A = (Q, \Sigma, F, s, \delta, \lambda)$, where $(Q, \Sigma, F, s, \delta)$ is a 2DTA^u and λ is a mapping $\lambda : Q \times \Sigma \to \{0, 1\}$.

The query expressed by a QA^u and the tree language defined by a QA^u are defined analogously to QA^{r}s.

Example 5.21 Consider Boolean circuits consisting of AND and OR gates that have one output but can have an arbitrary number of inputs. The following query automaton selects all nodes of the input tree that evaluate to a 1. Again, we only consider trees as inputs that represent Boolean circuits.

Define the QA$^u A = (Q, \Sigma = \{\mathrm{AND}, \mathrm{OR}, 0, 1\}, F, s, \delta)$, with $Q = \{s, u, \mathit{all\_one}, \mathit{all\_zero}, \mathit{mixed}\}$, $D = \{s\} \times \Sigma$, $U = \{u, \mathit{all\_one}, \mathit{all\_zero}, \mathit{mixed}\} \times \Sigma$, and $F = Q$. Define

1. for any natural number n and $\sigma \in \Sigma$, $\delta_\downarrow(s, \sigma, n) = s \cdots s$ (n times);
2. for all $\sigma \in \Sigma$, $\delta_{\mathrm{leaf}}(s, \sigma) = u$;
3. $(q_1, \sigma_1) \cdots (q_n, \sigma_n) \in L_\uparrow(\mathit{all\_one})$ iff for all $i \in \{1, \ldots, n\}$
   - if $q_i = u$ then $\sigma_i = 1$;
   - if $\sigma_i = \mathrm{AND}$ then $q_i = \mathit{all\_one}$; and
   - if $\sigma_i = \mathrm{OR}$ then $q_i = \mathit{mixed}$ or $q_i = \mathit{all\_one}$.
4. $(q_1, \sigma_1) \cdots (q_n, \sigma_n) \in L_\uparrow(\mathit{all\_zero})$ iff for all $i \in \{1, \ldots, n\}$
   - if $q_i = u$ then $\sigma_i = 0$;
   - if $\sigma_i = \mathrm{AND}$ then $q_i = \mathit{all\_zero}$ or $q_i = \mathit{mixed}$; and
   - if $\sigma_i = \mathrm{OR}$ then $q_i = \mathit{all\_zero}$.
5. $L_\uparrow(\mathit{mixed}) := U^* - (L_\uparrow(\mathit{all\_one}) \cup L_\uparrow(\mathit{all\_zero}))$.

The automaton first walks to the leaves (1) and then changes state s into state u (2). Hereafter, it walks back up again assigning to each inner node the state *all_one*, *all_zero* or *mixed*, depending on whether the evaluation of the subtrees of this node returns only ones, only zeros, or both ones and zeros, respectively (3–5). Consider for example (3): an internal node is assigned *all_one* if all the trees rooted at its children evaluate to 1. That is, first, if a child is a leaf then it should be labeled with a 1. Next, if a child is labeled with AND then it should be assigned the state *all_one*, as all his children in turn should evaluate to 1. Finally, if a child is labeled with OR then it should be assigned the state *all_one* or *mixed*, as at least one of this node's children should evaluate to 1.

The selection function is now defined as follows: for all $q \in Q$ and $\mathrm{op} \in \Sigma$, $\lambda(q, \mathrm{op}) = 1$ if and only if $q = \mathit{all\_one}$ and $\mathrm{op} \in \{\mathrm{AND}, \mathrm{OR}\}$, or $q = \mathit{mixed}$ and $\mathrm{op} = \mathrm{OR}$. ■

Even though QA^{u}s can accept all recognizable tree languages, they cannot even express all first-order logic definable queries.

Proposition 5.22 QA^u*s cannot express all queries definable in first-order logic.*

Proof. Let Σ be the alphabet $\{0, 1\}$. Consider the query "select all 1-labeled leaves for which there is no node among their left siblings that is labeled with a 1". Towards a contradiction, suppose there exists a QA^u A that expresses the above query. Let Q be the set of states of A and let $m = |Q|$. The crucial observation is that there exist at most $m!$ different sequences of states that A can take at the root of a tree. We set $n := m!$. Define for $i = 0, \ldots, n$, $\mathbf{t}_i$ as the tree consisting of a root (say, labeled 0) with $n + 1$ children, where the first i children are labeled with 0 and the others are labeled with 1. There now exist $j, j' \in \{0, \ldots, n\}$ such that $j < j'$ and A goes through the same sequence of root states for $\mathbf{t}_j$ and $\mathbf{t}_{j'}$. Since for each state and for each arity there is only one string of states that can be assigned to the children, the set of states assumed by A at the $(j' + 1)$-th leaf of $\mathbf{t}_j$ is the same as the set of states assumed by A at the $(j' + 1)$-th leaf of $\mathbf{t}_{j'}$. Since both leaves carry a 1, A selects them both or does not select them at all. This leads to the desired contradiction. ■

QA^{u}s cannot express the query in the proof of Proposition 5.22 because they cannot pass enough information from one sibling to another. Indeed, when the automaton makes a down transition at some node $\mathbf{n}$, it assigns a state to every child of $\mathbf{n}$; even though every child knows its own state, it cannot know in general which states are assigned to its siblings. To resolve this, we introduce in the next section query automata with "stay transitions". Such a transition consists of a two-way string-automaton which processes the string formed by the states and the labels of the children of a certain node, and then outputs for each child a new state.

5.3.2 Strong query automata

Tree automata with stay transitions are defined next.

Definition 5.23 A *generalized two-way deterministic tree automaton* (G2DTA^u) is a tuple $A = (Q, \Sigma, F, s, \delta)$, where Q, F, s, U, D, δ_{leaf}, δ_{root} and $\delta_\downarrow$ are defined as in Definition 5.19. Let U_{up} and U_{stay} be two disjoint regular subsets of U^*. Then $\delta_\uparrow$ is a function $\delta_\uparrow : U_{\mathrm{up}} \to Q$ (here the same conditions apply as in Definition 5.2), and

$\delta_- : U_{\text{stay}} \to Q^*$ is the transition function for stay transitions. We require that this function is computed by a generalized string query automaton (cf. Definition 5.6).

Most definitions remain the same as for QA^{u}s. Only, we now also have stay transitions: A makes a *stay transition* from a configuration $c_1 : C_1 \to Q$ to a configuration $c_2 : C_2 \to Q$ if there is a node $\mathbf{n}$ in $\mathbf{t}$ such that

(*i*) $\text{children}(\mathbf{n}) \subseteq C_1$,

(*ii*) $C_2 = C_1$,

(*iii*) $\delta_-(\pi(c_1, \mathbf{n})) = c_2(\text{children}(\mathbf{n}))$, and

(*iv*) c_1 is identical to c_2 on $C_1 \cap C_2$.

The above defined automata are much more expressive than MSO. Indeed, they can for instance simulate linear space Turing Machines on trees of depth one. Therefore, we restrict them in the following way:

Definition 5.24 A *strong* two-way deterministic tree automaton (S2DTA^u) is a G2DTA^u that makes at most one stay transition for the children of each node.

In Lemma 5.28 we show that the behavior of a S2DTA^u can be defined in MSO. It is then not difficult to construct an MSO sentence asserting that a particular G2DTA^u makes two stay transitions at the children of a particular node. Since satisfiability of MSO sentences on graphs of bounded tree-width, and consequently also on unranked trees, is decidable [Cou90], it is decidable whether a G2DTA^u is a S2DTA^u.

A strong query automaton is an S2DTA^u extended with a selection function.

Definition 5.25 A *strong query automaton* (SQA^u) is a tuple $A = (Q, \Sigma, F, s, \delta, \lambda)$, where $(Q, \Sigma, F, s, \delta)$ is a S2DTA^u and λ is a function from $Q \times \Sigma$ to $\{0, 1\}$.

We illustrate the above with an example.

Example 5.26 Recall the query of the proof of Proposition 5.22, *select all 1-labeled leaves for which there is no node among their left siblings labeled with a 1.* This query can be expressed by a SQA^u. Indeed, let $A = (Q, \Sigma, F, s, \delta, \lambda)$ be the SQA^u with $Q = F = \{s, \textit{stay}, \textit{up}, 1\}$, $D = \{s\} \times \Sigma$, $U = \{\textit{stay}, \textit{up}, 1\} \times \Sigma$, and where

- $U_{\text{up}} = \{\textit{up}\} \times \Sigma$,
- $U_{\text{stay}} = U^* - U_{\text{up}}$,
- for each natural number n and $\sigma \in \Sigma$, $\delta_\downarrow(s, \sigma, n) = s \cdots s$ (n times),
- for each $\sigma \in \Sigma$, $\delta_{\text{leaf}}(s, \sigma) = \textit{stay}$;
- δ_- is computed by the GSQA that assigns 1 to the first 1-labeled node and *up* to the others, and

- $L_\uparrow(up) = up^*1up^* + up^*$.

The automaton walks to the leaves, makes one stay transition, and then walks back to the root. The selection function is defined as follows: for each $\sigma \in \Sigma$ and $q \in Q$, $\lambda(q, \sigma) = 1$ iff $q = 1$. ∎

5.3.3 Expressiveness

We next prove that a query is expressible by a SQA^u if and only if it is definable in MSO. But first, we emphasize the remarkable difference between tree automata and query automata over unranked trees. Indeed, as shown in the next proposition, stay transitions do not increase the expressiveness with respect to defining tree languages. However, stay transitions do make a difference with respect to expressing queries, as was shown by Proposition 5.22 and Example 5.26.

Proposition 5.27 *Every* S2DTA^u *is equivalent to a* 2DTA^u *accepting the same tree language.*

Proof. This follows directly from the proof of Theorem 5.29, below, as the automaton that evaluates an MSO formula does not need any stay transitions before it decides whether to select the root of the tree. ∎

We first generalize Lemma 5.17 to query automata over unranked trees.

Lemma 5.28 *Every query computed by an unranked query automaton can be defined in MSO.*

Proof. The proof is similar to the proof of Lemma 5.17. We use some of the notation introduced there. Given an SQA^u $A = (Q, \Sigma, F, s, \delta)$, we again guess sets $Z_{f,B}$ and check their consistency. On input $\mathbf{t}$ these sets have the following intended meaning: a node $\mathbf{n} \in Z_{f,B}$ iff $f = f^A_{\mathbf{t_n}}$ and $B = \text{Assumed}^A(\mathbf{t}, \mathbf{n})$. As opposed to the proof of Lemma 5.17, the consistency check can no longer be specified in first-order logic because the correctness of the guesses depends on the transition functions $\delta_\uparrow$, $\delta_\downarrow$ and δ_- which are no longer finite functions, but are given by regular languages and by a GSQA. However, the correctness can easily be verified in MSO because, by Theorem 2.6 and Theorem 2.12, regular languages and GSQAs can be defined in MSO. Further, the correctness of the behavior functions crucially depends on our assumption that at most one stay transition can occur at the children of each node. Indeed, suppose a node $\mathbf{n}$ is labeled with transition function f and its n children are labeled with $f_1, \ldots, f_n$. Then we have to check for all states q and q' with $f(q) = q'$ that

1. $q = q'$ if $(q, \text{lab}_\mathbf{t}(\mathbf{n})) \in U$;
2. if $(q, \text{lab}_\mathbf{t}(\mathbf{n})) \in D$ then there exist states $q_1, \ldots, q_n$ such that $\delta_\downarrow(q, \sigma, n) = q_1 \cdots q_n$; there are now two possibilities:

(a) for each $i \in \{1, \dots, n\}$, $\text{up}(f_i, q_i) \in U$: in this case we should have

$$\delta_\uparrow((\text{up}(f_1, q_1), \sigma_1) \cdots (\text{up}(f_n, q_n), \sigma_n)) = q';$$

or

(b) for each $i \in \{1, \dots, n\}$, $\text{up}(f_i, q_i) \in U_{\text{stay}}$: in this case there should exist $q'_1, \dots, q'_n$ with

$$\delta_-((\text{up}(f_1, q_1), \sigma_1) \cdots (\text{up}(f_n, q_n), \sigma_n)) = q'_1 \cdots q'_n,$$

and

$$\delta_\uparrow(\text{up}(f_1, q'_1), \sigma_1) \cdots (\text{up}(f_n, q'_n), \sigma_n)) = q'.$$

Finally, if $f(q)$ is undefined then $\delta_\downarrow(q, \sigma, n)$ should be undefined; or in case (2a) $\delta_\uparrow((\text{up}(f_1, q_1), \sigma_1) \cdots (\text{up}(f_n, q_n), \sigma_n))$ or one of the $\text{up}(f_i, q_i)$ should be undefined; in case (2b)

$$\delta_-((\text{up}(f_1, q_1), \sigma_1) \cdots (\text{up}(f_n, q_n), \sigma_n)),$$

$\delta_\uparrow(\text{up}(f_1, q'_1), \sigma_1) \cdots (\text{up}(f_n, q'_n), \sigma_n))$, or one of the $\text{up}(f_i, q_i)$ or $\text{up}(f_i, q'_i)$ should be undefined.

From our assumption that an SQA^u can make at most one stay transition at the children of each node, it follows that the case distinctions (2a) and (2b) suffice. ∎

We are now ready to prove the main result of this section.

Theorem 5.29 *A query is expressible by a* SQA^u *if and only if it is definable in MSO.*

Proof. The only-if direction is given in Lemma 5.28.

Let $\varphi(x)$ be an MSO-formula of quantifier depth k. We describe an SQA^u that computes the query which is defined by φ. This automaton has to find out, for each node $\mathbf{n}$ of a tree $\mathbf{t}$, whether $\mathbf{t} \models \varphi[\mathbf{n}]$. This depends only on $\tau_k^{\text{MSO}}(\mathbf{t}, \mathbf{n})$, which in turn, by Proposition 4.12(1), depends only on $\tau_k^{\text{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$ and $\tau_k^{\text{MSO}}(\mathbf{t_n}, \mathbf{n})$. The case where trees can also have nodes with one child can be treated as in the proof of Theorem 5.18; hence, we can assume that all inner nodes have more than one child. In Figure 5.2, we describe an algorithm that evaluates φ.

The $\equiv_k^{\text{MSO}}$-type of a subtree $(\mathbf{t_m}, \mathbf{m})$ can be computed in a bottom-up manner by the automaton of Lemma 4.13. This automaton can be transformed to an equivalent two-way automaton as discussed at the end of Section 5.3.1. Note that the two-way automaton starts at $\mathbf{m}$.

Step 1 is now done in two phases. We re-use the pebbling idea from the proof of Theorem 5.18. First, the automaton makes a down transition. All children of $\mathbf{n}$, besides $\mathbf{n}1$ enter a U-state which remembers $\tau_k^{\text{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$ and waits until the computation in the subtree $\mathbf{t_{n1}}$ has finished. The $\equiv_k^{\text{MSO}}$-type of this subtree is computed bottom-up. The automaton "recognizes" that the subtree-evaluation is finished by meeting the U-states at the siblings of $\mathbf{n}1$. Next it makes an up transition, followed by a down transition. After this, $\mathbf{n}1$ has a U-state which remembers

Input: $\mathbf{t}$
Compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathrm{root}(\mathbf{t})}}, \mathrm{root}(\mathbf{t}))$, $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t}))$
for $i := 0$ **to** depth of $\mathbf{t}$ **do**
begin
  for all vertices $\mathbf{n}$ of level i **do**
  begin
    % the root is level 0
    % the type of $(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ *has already been computed*
    % now compute the type of $(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$
    1. **for** $j = 1, \ldots, \mathit{arity}(\mathbf{n})$ **do** compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}j}, \mathbf{n}j)$
    2. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$ from $\mathrm{lab}_{\mathbf{t}}(\mathbf{n})$ and the $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}j}, \mathbf{n}j)$
    3. Compute $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{n})$ from $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}}, \mathbf{n})$
    4. Deduce from $\tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathbf{n})$ whether $\varphi(\mathbf{n})$ holds
       If so, select $\mathbf{n}$
    5. **for** $j = 1, \ldots, \mathit{arity}(\mathbf{n})$ **do**
         Compute $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}j}}, \mathbf{n}j)$ from $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ and the $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}j'}, \mathbf{n}j')$
  end
end

Figure 5.2: The algorithm for computing the query defined by $\varphi(x)$ over unranked trees.

$\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}1}, \mathbf{n}1)$ and waits for the termination of the evaluation of the other subtrees which are computed in parallel. This evaluation simultaneously computes $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}j}, \mathbf{n}j)$, for each $j > 1$.

Step 2 is just a special case of step 1. Indeed, since the types of the subtrees of $\mathbf{t}_{\mathbf{n}}$ are present at the children of $\mathbf{n}$, they can be combined to the type of $\mathbf{t}_{\mathbf{n}}$ by making an up transition. Step 3 and 4 only involve information that is available at vertex $\mathbf{n}$.

It then only remains to show how step 5 can be done by an SQA^u. It should be noted that after the up transition of step 2 the information about the types of the subtrees of $\mathbf{n}$ is lost. Therefore the SQA^u first recomputes the $\equiv_k^{\mathrm{MSO}}$-types $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}j}, \mathbf{n}j)$, as described above (also keeping $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$ in mind).

We now show that there is a GSQA B computing the sequence $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}1}}, \mathbf{n}1) \cdots \tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}n}}, \mathbf{n}n)$ on input

$$(\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n}), \tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}1}, \mathbf{n}1)) \cdots (\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n}), \tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}n}, \mathbf{n}n)),$$

for a tree $\mathbf{t}$ and $\mathbf{n}$ a node of $\mathbf{t}$ of arity n. Let σ be the label of $\mathbf{n}$. Then, by Proposition 4.12(3), for each $i = 1, \ldots, n$, $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}i}}, \mathbf{n}i)$ only depends on $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathbf{n}}}, \mathbf{n})$, $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_{\mathbf{n}1}, \ldots, \mathbf{t}_{\mathbf{n}(i-1)}), \mathbf{n})$, $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}i)$ (which depends only on $\tau_k^{\mathrm{MSO}}(\mathbf{t}_{\mathbf{n}i}, \mathbf{n}i)$), and $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_{\mathbf{n}(i+1)}, \ldots, \mathbf{t}_{\mathbf{n}n}), \mathbf{n})$.

Now, B is defined as the automaton combining, as specified in Lemma 5.9, the automata B_1 and B_2, where B_1 computes $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_{\mathbf{n}1} \ldots \mathbf{t}_{\mathbf{n}(i-1)}), \mathbf{n})$ and B_2 computes $\tau_k^{\mathrm{MSO}}(\sigma(\mathbf{t}_{\mathbf{n}(i+1)} \ldots \mathbf{t}_{\mathbf{n}n}), \mathbf{n})$. So, at each position i the automaton B has enough information to output $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}_i}, \mathbf{n}i)$.

Hence, step 5 can be done by recomputing the $\equiv_k^{\mathrm{MSO}}$-types $\tau_k(\mathbf{t_n}_j, \mathbf{n})$ (in the same way as in step 2) and making one stay transition. ■

Remark 5.30 Allowing an SQA^u to make any constant number of stay transitions at the children of each node does not increase the expressiveness of the formalism. Indeed, like in the proof of Theorem 5.29 such an automaton can be simulated in MSO.

5.4 Optimization

In this section we establish the complexity of the non-emptiness and the equivalence problem for the query automata defined in the previous sections. These problems are defined as follows:

Non-emptiness: Given a query automaton A, does there exist a tree $\mathbf{t}$ such that $A(\mathbf{t}) \neq \emptyset$?

Equivalence: Given two query automata A_1 and A_2, do they express the same query? That is, is $A_1(\mathbf{t}) = A_2(\mathbf{t})$ for all trees $\mathbf{t}$?

We show that these problems are EXPTIME-complete for QA^rs, QA^us and SQA^us. EXPTIME-hardness of all these problems follows from EXPTIME-hardness of the non-emptiness problem for QA^rs. EXPTIME-membership follows from EXPTIME-membership of the non-emptiness problem for SQA^us, since we will show that equivalence can be reduced to non-emptiness in polynomial time and query automata on ranked trees are special cases of query automata on unranked trees.

The *size* of an SQA^u is the sum of the sizes of the DFAs representing the up transitions, the sizes of the automata for the stay transitions, the sizes of the regular expressions representing the down transitions, and the size of the set of states of the SQA^u. We point out in the proof of Theorem 5.32, why we need DFAs, as opposed to NFAs, for the representation of up transitions.

We start by observing that deciding whether the tree language defined by a $2\mathrm{DTA}^r$ is non-empty is EXPTIME-hard. We use a reduction from the problem TWO PERSON CORRIDOR TILING which is defined in Section 4.5.

Proposition 5.31 *Deciding whether a* $2\mathrm{DTA}^r$ *accepts any tree is EXPTIME-hard.*

Proof. We reduce TWO PERSON CORRIDOR TILING to non-emptiness of $2\mathrm{DTA}^r$. Recall that this is the problem to decide, given a set of tiles T, $H, V \subseteq T \times T$, two sequences of tiles $\bar{b} = b_1, \dots, b_n$ and $\bar{t} = t_1, \dots, t_n \in T^n$, whether player one wins the corridor game.

A strategy for player one can be represented by a tree where the nodes are labeled with tiles. Indeed, if we put the rows of a tiling next to each other rather than on top of each other, then every branch, i.e., the sequence of labels from the root to a leaf, of a tree represents a possible tiling. If we forget about the start row $\bar{b}$ for a moment, then the odd depth nodes have no siblings and represent moves of player one and the even depth nodes do have siblings and represent all the choices of player two. A strategy is then winning when every branch is either a corridor tiling or is a tiling where player two made a false move. The 2DTA^r A we construct will only accept trees that correspond to winning strategies for player one. The automaton essentially only has to check the horizontal and vertical constraints. The vertical constraints at a node $\mathbf{n}$ of the input tree can be checked by moving up n nodes (the width of the corridor), while the horizontal constraints can be checked by looking at the tile carried by the parent of $\mathbf{n}$.

We now formally define when a tree represents a winning strategy. Take Σ as $\{0,1,2\} \times \{1,\dots,n\} \times T$. If a node is labeled with (i,j,t) and $i \neq 0$, then this means that player i places tile t on the j-th position of the current row. The case $i = 0$ is just to define the first n nodes which are labeled with $b_1, \dots, b_n$. We say that a Σ-tree $\mathbf{t}$ represents a winning strategy for the first player if the following holds:

1. $\mathbf{t}$ starts with a monadic tree labeled with $\bar{b}$. That is, the root carries the label $(0,1,b_1)$, the only child of the root carries the label $(0,2,b_2)$, and so on.
2. If there exists a node of depth n (recall that the root has depth 0) then there exists only one such node (say $\mathbf{n}$) and additionally $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}) = (1,1,t)$ for some $t \in T$: player one places the tile on the first column of the second row whenever there is a second row; indeed, $\bar{b}$ and $\bar{t}$ can already form a corridor tiling.
3. For every internal node $\mathbf{n}$ of $\mathbf{t}$, if $\mathrm{lab}_{\mathbf{t}}(\mathbf{p}) = (i,j,t)$ with $i \in \{1,2\}$ and $\mathbf{p}$ the parent of $\mathbf{n}$, then $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}) = ((i \bmod 2)+1, (j+1) \bmod n, t')$ for some $t' \in T$; this means that players one and two place tiles on turn.
4. No two siblings are labeled with the same label; and, nodes corresponding to moves of player one have no siblings.
5. Every alternative of player two should be present: for every node $\mathbf{n}$ with $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}) = (1,j,t)$ and for every $t' \in T$ there is a child $\mathbf{m}$ of $\mathbf{n}$ labeled with $(2,(j+1) \bmod n, t')$.
6. Each branch extended with $\bar{t}$ corresponds to a corridor tiling from $\bar{b}$ to $\bar{t}$ or should contain a false move by player two.

The 2DTA^r A works on trees of rank n. Let N be $|T|+|H|+|V|+n$. Clearly (1)–(5) can be checked by A by using a number of states linear in N. We now consider (6). Suppose A arrives at $\mathbf{n}$. In order to check the horizontal constraints at $\mathbf{n}$, A already remembered the tile of the parent of $\mathbf{n}$ in its state when it moved down. To check the vertical constraints, A just has to move up n nodes to get the tile that is placed immediately below the square corresponding to $\mathbf{n}$. However, moving up requires the cooperation of all siblings. To this end, A moves through the tree level by level, and for each level makes n up transitions to get the required tile. Again, only a number of states that is linear in N is needed. Moreover, A can clearly be computed in LOGSPACE. ■

We next show how to test non-emptiness for SQA^{u}s in EXPTIME.

Theorem 5.32 *Non-emptiness of* SQA^u*s is in EXPTIME.*

Proof. We describe an EXPTIME algorithm which decides whether the SQA^u A is non-empty. The proof consists of two parts. First, we define a two-way deterministic tree automaton A' such that A is non-empty iff A' accepts at least one tree (we then also say that A' is non-empty). Moreover, the size of A' is linear in the size of A. Subsequently, we show that testing non-emptiness of two-way deterministic tree automata is in EXPTIME. This then implies that non-emptiness of SQA^{u}s is in EXPTIME.

Construction of A'. The two-way deterministic automaton A' works over the alphabet $\Sigma \cup (\Sigma \times \{1\})$. On input $\mathbf{t}$, it first checks whether there is exactly one node with a label in $\Sigma \times \{1\}$. This can be done by one traversal of the tree from the root to the leaves. If there is more than one such node or none at all, then A' rejects. Otherwise, A' walks back to the root and starts simulating A, that is, it just behaves like A would but without actually selecting nodes. Let $\mathbf{n}$ be the unique node with a label in $\Sigma \times \{1\}$. Then A' accepts when A does and when, additionally, A selects $\mathbf{n}$. The latter can be achieved by keeping a flag in the state of A' from the moment A selects $\mathbf{n}$. Clearly, the size of A' is linear in the size of A.

Testing non-emptiness of two-way deterministic tree automata on unranked trees is in EXPTIME. Let $A' = (Q, \Gamma, F, s_0, \delta)$ be such an automaton. We construct a nondeterministic bottom-up automaton (NBTA) $B = (Q_B, \Gamma, F_B, \delta_B)$ (cf. Section 4.1.3) whose size is exponential in the size of A' with the additional property that B is non-empty iff A' is non-empty. By Lemma 4.9 we know that testing non-emptiness of NBTAs is in PTIME. Hence, testing non-emptiness of two-way deterministic automata is in EXPTIME.

The set of states Q_B consists of all tuples of the form (f, d, s, σ) where

- $f : Q \to Q$ is a partial function;
- $d : Q \to Q$ and $s : Q \to Q$ are total functions; and
- $\sigma \in \Gamma$.

To describe the intuition behind the components in the states of Q_B we introduce the following notion. A *state assignment* for a tree $\mathbf{t}$ is a mapping $\rho : \text{Nodes}(\mathbf{t}) \to Q$. A state assignment ρ for $\mathbf{t}$ is *semi-valid* if for every node $\mathbf{n}$ of $\mathbf{t}$ of arity n, $\rho(\mathbf{n}1) \cdots \rho(\mathbf{n}n) \in \delta(\rho(\mathbf{n}), \text{lab}_{\mathbf{t}}(\mathbf{n}))$, and for every leaf node $\mathbf{n}$, $\varepsilon \in \delta(\rho(\mathbf{n}), \text{lab}_{\mathbf{t}}(\mathbf{n}))$. We say that a state assignment ρ for a tree $\mathbf{t}$ is *valid* iff it is *semi-valid* and $\rho(\text{root}(\mathbf{t})) \in F$. Clearly, a tree $\mathbf{t}$ is *accepted* by T if there exists a valid state assignment for it.

The intuition behind the states in Q_B is that for each semi-valid state assignment ρ for a tree $\mathbf{t}$, if $\rho(\mathbf{n}) = (f, d, s, \sigma)$ then $f^A_{\mathbf{t_n}} = f$ and $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}) = \sigma$. The functions d and s are just to facilitate the definition of the transition function of A' which we define next.

For all $n \geq 1$, $\sigma \in \Gamma$, and every state $(f, d, s, \sigma') \in Q_B$,

$$w = (f_1, d_1, s_1, \sigma_1) \cdots (f_n, d_n, s_n, \sigma_n) \in \delta_B((f, d, s, \sigma'), \sigma)$$

iff

1. $\sigma' = \sigma$;
2. $f(q) = q$ for each $q \in Q$ with $(q, \sigma) \in U$;
3. $\delta_\downarrow(q, \sigma, n) = d_1(q) \cdots d_n(q)$ for each $q \in Q$ with $(q, \sigma) \in D$ and for which $f(q)$ is defined; there are now two possibilities:
   (a) for each $i \in \{1, \ldots, n\}$, $(f_i(d_i(q)), \sigma_i) \in U$: in this case we should have
   $$\delta_\uparrow((f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n)) = f(q);$$
   or
   (b) for each $i \in \{1, \ldots, n\}$, $(f_i(d_i(q)), \sigma_i) \in U_{\mathrm{stay}}$: in this case we should have
   $$\delta_-((f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n)) = s_1(q) \cdots s_n(q),$$
   and
   $$\delta_\uparrow((f_1(s_1(q)), \sigma_1) \cdots (f_n(s_n(q))), \sigma_n) = f(q).$$

   If $f(q)$ is undefined then $\delta_\downarrow(q, \sigma, n)$ should be undefined; or in case (3a) one of the $f_i(d_i(q))$ or $\delta_\uparrow(f_1(d_1(q)) \cdots f_n(d_n(q)))$ should be undefined; or in case (3b) $\delta_-(f_1(d_1(q)) \cdots f_n(d_n(q)))$, $\delta_\uparrow((f_1(s_1(q)), \sigma_1) \cdots (f_n(s_n(q)), \sigma_n))$, or one of the $f_i(d_i(q))$ or $f_i(s_i(q))$ should be undefined.

   From our assumption that an SQA^u can make at most one stay transition at the children of each node, it follows that the case distinctions (3a) and (3b) suffice.

Further, $\varepsilon \in \delta_B((f, d, s, \sigma'), \sigma)$ iff $f = f^A_{\mathbf{t}(\sigma)}$ and $\sigma = \sigma'$. Finally, define $F_B = \{(f, d, s, \sigma) \mid \mathrm{States}(f, \delta_{\mathrm{root}}(\cdot, \sigma), s_0) \cap F \neq \emptyset\}$.[7] It is now readily checked that for each semi-valid state assignment ρ for a tree $\mathbf{t}$, $\rho(\mathbf{n}) = (f, d, s, \sigma)$ iff $f^A_{\mathbf{t_n}} = f$ and $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}) = \sigma$. By definition of F_B, we have that A' accepts $\mathbf{t}$ iff there exists a valid state assignment for $\mathbf{t}$. Consequently, A' is non-empty iff B is non-empty.

It remains to show that each $\delta((f, d, s, \sigma'), \sigma)$ can be accepted by an NFA whose size is exponential in the size of A'. We will define a two-way deterministic automaton M whose size is polynomial in A' that accepts $\delta((f, d, s, \sigma'), \sigma)$. By Lemma 2.10, M is equivalent to an NFA whose size is at most exponential in A'.

[7] Here, for each $\sigma \in \Gamma$, $\delta_{\mathrm{root}}(\cdot, \sigma)$ is the function mapping each q to $\delta_{\mathrm{root}}(q, \sigma)$.

(1) does not depend on the input;

(2) does not depend on the input;

(3) For each $q \in Q$ with $(q, \sigma) \in D$, we do the following. We only describe the case where $f(q)$ is defined, the converse case is similar. To test whether

$$\delta_\downarrow(q, \sigma, n) = d_1(q) \cdots d_n(q);$$

we just simulate the finite union of regular expressions representing $L_\downarrow(q, \sigma)$ on the string $d_1(q) \cdots d_n(q)$. This can be done by subsequently trying to match each regular expression in this union. Due to the very simple form of these regular expressions (namely xy^*z) this only needs a number of states linear in the size of the expressions. Hereafter, M tests whether $(f_i(d_i(q)), \sigma_i) \in U$ for each $i \in \{1, \ldots, n\}$, or whether $(f_i(d_i(q)), \sigma_i) \in U_{\text{stay}}$ for each $i \in \{1, \ldots, n\}$. This test is performed by another sweep through the input string w. Depending on this test M does the following.

   (a) M simply simulates the DFA for $L_\uparrow(q)$; that is, M tests by another sweep through w whether

   $$(f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n) \in L_\uparrow(q);$$

   This only needs a number of states linear in the size of the automaton for $L_\uparrow(q)$.

   (b) In the second case, M verifies that

   $$\delta_-((f_1(d_1(q)), \sigma_1) \cdots (f_n(d_n(q)), \sigma_n)) = s_1(q) \cdots s_n(q),$$

   and that

   $$((f_1(s_1(q)), \sigma_1) \cdots (f_n(s_n(q))), \sigma_n)) \in L_\uparrow(q).$$

   The former can be done by simulating the GSQA for δ_-. Recall our convention that each GSQA only outputs one symbol at each position. The latter can be done by one sweep through the input string simulating the DFA for $L_\uparrow(q)$. Again only a linear number of states in the size of A is needed.

We briefly come back to why we need DFAs rather than NFAs: in the case where $f(q)$ is undefined we must check that an up transition is undefined for a certain sequence of states. When up transitions are represented by DFAs this is easy: we just simulate the automaton and see whether it gets stuck. For an NFA, however, this is much harder as we have to check that all computations are undefined.

■

We conclude this section by showing how to reduce equivalence to non-emptiness. Let A_1 and A_2 be two SQA^{u}s working over Σ-trees. Define the SQA^u B working over trees labeled with symbols from the alphabet $\Sigma \cup (\Sigma \times \{1\})$, as follows. On input $\mathbf{t}$, B first checks whether there is exactly one node with a label in $\Sigma \times \{1\}$. This can be done by one traversal of the tree from the root to the leaves. If there is more than one such node or none at all, then B rejects. Otherwise, let $\mathbf{n}$ be the unique node with a label in $\Sigma \times \{1\}$. Then B walks back to the root, first simulates A_1 and then A_2, and remembers which automaton selects $\mathbf{n}$. Recall our convention that we only consider automata that terminate on every input. If A_1 and A_2 both select $\mathbf{n}$ or both do not select $\mathbf{n}$, then B rejects. Otherwise, B selects the root. Hence, B is non-empty iff A_1 and A_2 are not equivalent. As the size of B is linear in the sizes of A_1 and A_2, and non-emptiness is in EXPTIME, it follows that equivalence is in EXPTIME. Hence, we have the following theorem.

Theorem 5.33 *Equivalence of* QA^r*s,* QA^u*s, and* SQA^u*s is in EXPTIME.*

5.5 Nondeterministic query automata

In classical automata theory on strings and trees, determinism, nondeterminism, and two-wayness are equally powerful w.r.t. defining languages.[8] In the previous sections, we restricted ourselves to deterministic query automata only and pointed out the difference between the one and two-way ones. In the last section of this chapter, we want to complete the picture and therefore shift attention to nondeterministic query automata. First of all it is not clear how such automata can express queries. As we will see, this can happen in various ways. Further, in strong contrast to the deterministic case, bottom-up nondeterministic query automata already suffice to capture all unary queries definable in MSO. This means, in particular, that no stay transitions or down transitions are needed.

We will restrict attention to unranked trees only, because there will be no fundamental difference between the ranked and the unranked ones.

Let $B = (Q, \Sigma, F, \delta)$ be a nondeterministic bottom-up automaton as defined in Definition 4.6. A *state assignment* for a tree $\mathbf{t}$ is a mapping $\rho : \text{Nodes}(\mathbf{t}) \to Q$. A state assignment ρ for $\mathbf{t}$ is *valid* if $\rho(\text{root}(\mathbf{t})) \in F$; $\rho(\mathbf{n}1) \cdots \rho(\mathbf{n}n) \in \delta(\rho(\mathbf{n}), \text{lab}_{\mathbf{t}}(\mathbf{n}))$, for every node $\mathbf{n}$ of $\mathbf{t}$ of arity n; and, $\varepsilon \in \delta(\rho(\mathbf{n}), \text{lab}_{\mathbf{t}}(\mathbf{n}))$, for every leaf node $\mathbf{n}$. A tree $\mathbf{t}$ is *accepted* by T if there exists a valid state assignment for it.

Definition 5.34 A *nondeterministic bottom-up query automaton* (NBQA) is a tuple $T = (Q, \Sigma, F, \delta, \lambda)$, where (Q, Σ, F, δ) is an NBTA and $\lambda : Q \times \Sigma \to \{0, 1\}$ is a function.

There are various possibilities to define the query expressed by a nondeterministic query automaton. We consider two of them. In the existential semantics a node is selected if it selected by at least one accepting computation. In the universal

[8] Although, they may be more succinct then one another [GH96].

semantics, on the other hand, a node is selected if it is selected by every accepting computation. We define this formally:

The query expressed *existentially* by T on a tree $\mathbf{t}$ is defined as

$$T(\mathbf{t})^{\exists} := \{\mathbf{n} \mid \text{there exists a valid state assignment } \rho \text{ for } \mathbf{t} \text{ such that } \lambda(\rho(\mathbf{n}), \mathrm{lab}_{\mathbf{t}}(\mathbf{n})) = 1\}.$$

The query expressed *universally* by T on a tree $\mathbf{t}$ is defined as

$$T(\mathbf{t})^{\forall} := \{\mathbf{n} \mid \text{for every valid state assignment } \rho \text{ for } \mathbf{t},\ \lambda(\rho(\mathbf{n}), \mathrm{lab}_{\mathbf{t}}(\mathbf{n})) = 1\}.$$

As already indicated above, we show that nondeterministic query automata capture MSO:

Theorem 5.35 *Both under the existential and the universal semantics, nondeterministic query automata express exactly the queries definable in MSO.*

Proof. Let T be an NBQA. The query expressed existentially (universally) by T can be defined in MSO by quantifying existentially (universally) over all valid state assignments. Checking whether a state assignment is valid only depends on membership of strings in regular languages and can, hence, be done in MSO.

For the proof of the other direction, let $\varphi(x)$ be an MSO formula of quantifier depth k. We now define an NBQA $T = (Q, \Sigma, F, \delta)$ that expresses both existentially and universally the query defined by φ. We construct T in such a way that for any node $\mathbf{n}$ of a tree $\mathbf{t}$ and for any valid state assignment ρ we have $\rho(\mathbf{n}) = (\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n}), \tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}))$. This clearly allows to deduce whether $\mathbf{t} \models \varphi[\mathbf{n}]$. The nondeterministic nature of T allows it to compute $\varphi(x)$ in a bottom-up way. Indeed, the equivalence types of the subtrees can be computed in a bottom-up fashion like before, while the equivalence types of the envelopes can be simply guessed. The deterministic query automata considered in the previous sections did not have this ability and therefore needed two-way movements to simulate all of MSO.

There will be exactly one valid state assignment for every tree. Hence, the existential and universal semantics of T coincide.

Define $Q = \Phi_k \times \Phi_k$;

$$F = \{(\theta, \overline{\theta}) \mid \text{there exists a tree } \mathbf{t} \text{ such that } \tau_k^{\mathrm{MSO}}(\overline{\mathbf{t}_{\mathrm{root}(\mathbf{t})}}, \mathrm{root}(\mathbf{t})) = \overline{\theta}, \text{ and } \tau_k^{\mathrm{MSO}}(\mathbf{t}, \mathrm{root}(\mathbf{t})) = \theta\};$$

and for every $(\theta, \overline{\theta}) \in Q$ and $\sigma \in \Sigma$,

$$(\theta_1, \overline{\theta_1}) \cdots (\theta_n, \overline{\theta_n}) \in \delta((\theta, \overline{\theta}), \sigma)$$

whenever there exists a tree $\mathbf{t}$ with a node $\mathbf{n}$ of arity n such that

- for $i = 1, \ldots, n$, $\tau_k^{\mathrm{MSO}}(\mathbf{t_{n\mathit{i}}}, \mathbf{n}i) = \theta_i$ and $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_{n\mathit{i}}}}, \mathbf{n}i) = \overline{\theta_i}$;
- $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n}) = \theta$;

- $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \overline{\theta}$; and
- $\mathrm{lab}_{\mathbf{t}}(\mathbf{n}) = \sigma$.

Note that the last requirement only depends on θ and $\overline{\theta}$.

Further, define $\lambda((\theta, \overline{\theta}), \sigma) = 1$ if there exists a tree $\mathbf{t}$ and a node $\mathbf{n}$ such that $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n}) = \theta$, $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n}) = \overline{\theta}$, and $\mathbf{t} \models \varphi[\mathbf{n}]$.

A simple induction on the height of the nodes of an input tree $\mathbf{t}$, using Proposition 2.14(2), shows that if $\rho(\mathbf{n}) = (\theta_{\mathbf{n}}, \overline{\theta_{\mathbf{n}}})$, then $\theta_{\mathbf{n}} = \tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n})$. An induction on the depth of nodes, using Proposition 2.14(3), then shows that for every node $\mathbf{n}$, $\overline{\theta_{\mathbf{n}}} = \tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$. We can conclude that T expresses the query defined by φ.

It remains to show that $\delta((\theta, \overline{\theta}), \sigma)$ is regular. This is indeed the case. as θ is uniquely determined by $\theta_1 \cdots \theta_n$ and this can be checked by an automaton like in the proof of Lemma 4.13. Further, for each i, $\overline{\theta}_i$ is uniquely determined by $\overline{\theta}$, $\theta_1 \cdots \theta_{i-1}$, θ_i, and $\theta_{i+1} \cdots \theta_n$. These dependencies can be checked by an automaton similar to the GSQA B in the proof of Theorem 5.29. ■

6

Applications and related work

In this chapter we apply the techniques developed in this dissertation and discuss some related work. First, we drastically improve the upper bound on the complexity of the equivalence test of Region Algebra expressions from iterated exponential to EXPTIME. The Region Algebra is a query language for manipulating text regions introduced by Consens and Milo [CM98a] as a formal model for the PAT algebra. We obtain our result by an efficient translation of Region Algebra expressions into extended AGs and by then applying our algorithm for testing equivalence of the latter. Further, we apply the techniques used to obtain the expressiveness results in previous chapters to the actual XML transformation language XSLT [Cla99]. Specifically, we show that already a very restricted subset of XSLT has the ability to issue any MSO pattern at any node in the document. That is, when XSLT arrives at a node it can decide for any unary MSO formula $\varphi(x)$ whether this formula holds at that node and use this information for further processing of the document. Hereby, on the one hand, we reveal that core XSLT has a very powerful pattern language at its disposal, and, on the other hand, provide evidence for the robustness of the language. Next, we compare MSO, and therefore query automata and extended AGs, with the selective power of languages for semi-structured data and XML. The navigational mechanism of such languages is usually based on regular path expressions [ABS99]. It is thus justified to model the selective power of these languages by first-order logic extended with regular path expressions. That is, for each regular expression r we add the predicate $r(x, y)$ expressing that x is an ancestor of y and that the string consisting of the symbols on the path from x to y belongs to the language defined by r. We then show that this logic, and therefore the current query languages for semi-structured data, in strong contrast with MSO, lack the ability to perceive the input tree as a whole. In particular, we formally prove that the above logic cannot define the class of trees representing Boolean circuits evaluating to true. Finally, we elaborate on the connection between ranked and unranked trees.

6.1 Optimization of Region Algebra expressions

The region algebra introduced by Consens and Milo [CM98a, CM94] is a set-at-a-time algebra, based on the PAT algebra [ST92], for manipulating text regions. In this section we show that any Region Algebra expression can be simulated by an extended AG of polynomial size. This then leads to an EXPTIME algorithm for the equivalence and emptiness test of Region Algebra expressions. The algorithm of Consens and Milo is based on the equivalence test for first-order logic formulas over trees which has a non-elementary lower bound. Our algorithm therefore drastically improves the complexity of the equivalence test for the Region Algebra and matches more closely the coNP lower bound [CM98a].

It should be pointed out that our definition differs slightly from the one in [CM98a]. Indeed, we restrict ourselves to regular languages as patterns, while Consens and Milo do not use a particular pattern language. This is no loss of generality since

- on the one hand, regular languages are the most commonly used pattern language in the context of document databases; and,
- on the other hand, the huge complexity of the algorithm of [CM98a] is not due to the pattern language at hand, but is due to quantifier alternation of the resulting first-order logic formula, induced by combinations of the operators '$-$' (difference) and $<$, $>$, $\subset$, and $\supset$.

Definition 6.1 A *region index schema* $\mathcal{I} = (S_1, \ldots, S_n, \Sigma)$ consists of a set of region names $S_1, \ldots, S_n$ and a finite alphabet Σ.

If N is a natural number, then a *region over* N is a pair (i, j) with $i \leq j$ and $i, j \in \{1, \ldots, N\}$.

An *instance* I of a region index schema $\mathcal{I}$ consists of a string $I(\omega) = a_1 \ldots a_{N_I} \in \Sigma^*$ with $N_I > 0$, and a mapping associating to each region name S a set of regions over N_I.

We abbreviate $r \in \bigcup_{i=1}^{n} I(S_i)$ by $r \in I$. We use the notation $L(r)$ (respectively $R(r)$) to denote the location of the left (respectively right) endpoint of a region r and denote by $\omega(r)$ the string $a_{L(r)} \ldots a_{R(r)}$.

Example 6.2 Consider the region index schema $\mathcal{I} = (\mathbf{Proc}, \mathbf{Func}, \mathbf{Var}, \Sigma)$. In Figure 6.1 an example of an instance over $\mathcal{I}$ is depicted. Here, $N_I = 16$, $I(\omega) =$ abcdefghijklmnop, $I(\mathbf{Proc}) = \{(1, 16), (6, 10)\}$, $I(\mathbf{Func}) = \{(12, 16)\}$ and $I(\mathbf{Var}) = \{(2, 3), (6, 7), (12, 13)\}$. ■

For two regions r and s in I define:

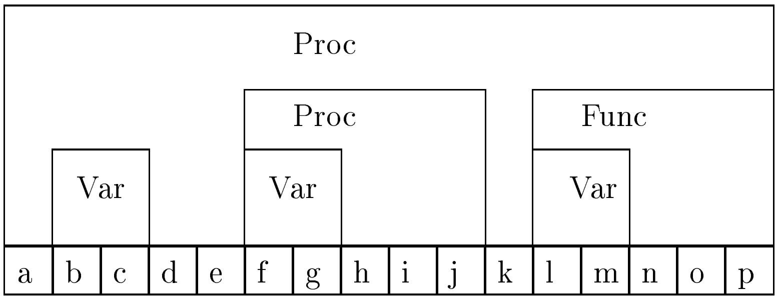

Figure 6.1: An instance I over the region index schema of Example 6.2

- $r < s$ if $R(r) < L(s)$ (r *precedes* s); and
- $r \subset s$ if $L(s) < L(r)$ and $R(r) \leq R(s)$, or $L(s) \leq L(r)$ and $R(r) < R(s)$ (r is *included in* s).

We also allow the dual operators $r > s$ and $r \supset s$ which have the obvious meaning.

Definition 6.3 An instance I is *hierarchical* if

- $I(S) \cap I(S') = \emptyset$ for all region names S and S' in $\mathcal{I}$, and
- for all $r, s \in I$, one of the following holds: $r < s$, $s < r$, $r \subset s$ or $s \subset r$.

The last condition simply says that if two regions overlap then one is strictly contained in the other.

The instance in Figure 6.1 is hierarchical. Like in [CM98a], we only consider hierarchical instances. We now define the Region Algebra.

Definition 6.4 *Region Algebra expressions over* $\mathcal{I} = (S_1, \ldots, S_n, \Sigma)$ are inductively defined as follows:

- every region name of $\mathcal{I}$ is a Region Algebra expression;
- if e_1 and e_2 are Region Algebra expressions then $e_1 \cup e_2$, $e_1 - e_2$, $e_1 \subset e_2$, $e_1 < e_2$, $e_1 \supset e_2$, and $e_1 > e_2$ are also Region Algebra expressions;
- if e is a Region Algebra expression and R is a regular language then $\sigma_R(e)$ is a Region Algebra expression.

The semantics of a Region Algebra expression on an instance I is defined as follows:

$$\begin{aligned}
[\![S]\!]^I &:= \{r \mid r \in I(S)\};\\
[\![\sigma_R(e)]\!]^I &:= \{r \mid r \in [\![e]\!]^I \text{ and } \omega(r) \in R\};\\
[\![e_1 \cup e_2]\!]^I &:= [\![e_1]\!]^I \cup [\![e_2]\!]^I;\\
[\![e_1 - e_2]\!]^I &:= [\![e_1]\!]^I - [\![e_2]\!]^I;
\end{aligned}$$

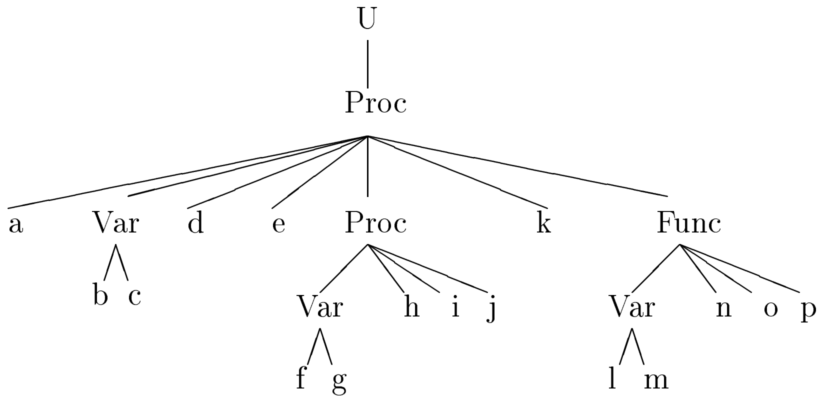

Figure 6.2: The tree $\mathbf{t}_I$ corresponding to the instance I of Figure 6.1.

and for $\star \in \{<, >, \subset, \supset\}$:

$$[\![e_1 \star e_2]\!]^I \quad := \quad \{r \mid r \in [\![e_1]\!]^I \text{ and } \exists s \in [\![e_2]\!]^I \text{ such that } r \star s\}.$$

Example 6.5 The Region Algebra expression **Proc** $\supset \sigma_{\Sigma^*\mathrm{start}\Sigma^*}($**Proc**$)$ defines all the **Proc** regions which contain a **Proc** region that contains the string start. ■

The important observation is that for any region index schema $\mathcal{I} = (S_1, \ldots, S_n, \Sigma)$ there exists an ECFG $G_\mathcal{I}$ such that any hierarchical instance of $\mathcal{I}$ 'corresponds' to a derivation tree of $G_\mathcal{I}$. This ECFG is now defined as follows: $G_\mathcal{I} = (N, T, P, U)$, with $N = \{S_1, \ldots, S_n\}$, $T = \Sigma$, and where P consists of the rules

$$\begin{array}{rcll} p_0 & := & U & \to & (S_1 + \ldots + S_n + \Sigma)^+; \\ p_1 & := & S_1 & \to & (S_1 + \ldots + S_n + \Sigma)^+; \\ & & & \vdots & \\ p_n & := & S_n & \to & (S_1 + \ldots + S_n + \Sigma)^+. \end{array}$$

For example, the derivation tree $\mathbf{t}_I$ of $G_\mathcal{I}$ representing the instance I of Figure 6.1 is depicted in Figure 6.2. Regions in I then correspond to nodes in $\mathbf{t}_I$ in the obvious way. We denote the node in $\mathbf{t}_I$ that corresponds to the region r by $\mathbf{n}_r$.

Since extended AGs can store results of subcomputations in their attributes, they are naturally closed under composition. It is, hence, no surprise that the translation of Region Algebra expressions into extended AGs proceeds by induction on the structure of the former.

Theorem 6.6 *For every Region Algebra expression e over $\mathcal{I}$ there exists an extended AG $\mathcal{F}_e$ over $G_\mathcal{I}$ such that for every hierarchical instance I and region $r \in I$, $r \in [\![e]\!]^I$ if and only if $\mathcal{F}_e(\mathbf{t}_I)(result_e(\mathbf{n}_r)) = 1$. Moreover, $\mathcal{F}_e$ can be constructed in time polynomial in the size of e.*

Proof. The proof proceeds by induction on the structure of Region Algebra expressions. We represent the regular languages occurring as patterns in Region Algebra expressions by DFAs. The extended AG $\mathcal{F}_e$ will always contain the attribute $result_e$ which is synthesized for all region names. As before the R_d's that are not specified are assumed to be empty. Region Algebra expressions can only select regions, therefore, no attributes are defined for terminals.

1. $e = S_i$: $A_e = \{result_e, lab\}$; $D_e = \{0, 1, S_1, \ldots, S_n\} \cup \Sigma$; for $i = 1, \ldots, n$, define in the context $(p_i, result_e, 0)$ the rule

$$result_e(0) := \langle \sigma_0 = lab, \sigma_1 = \varepsilon, \ldots, \sigma_{n+1} = \varepsilon; R_1 = \{S_i\}, R_0 = D_e^* - R_1 \rangle.$$

2. $e = \sigma_R(e_1)$: Let $M = (S, \Sigma, \delta, s_0, F)$ be the DFA accepting R with $S = \{s_0, \ldots, s_m\}$. Define $A_e = A_{e_1} \cup S$ and $D_e = D_{e_1} \cup S$. W.l.o.g, we assume $S \cap A_{e_1} = \emptyset$ and $S \cap D_{e_1} = \emptyset$. We now define the semantic rules of $\mathcal{F}_e$ as the semantic rules of $\mathcal{F}_{e_1}$ extended with the ones we describe next.

   Each non-terminal has the synthesized attributes $s_0, \ldots, s_m$. They are defined in $\mathcal{F}_e$ such that for a region instance I and region $r \in I$, $\mathcal{F}_e(\mathbf{t}_I)(s(\mathbf{n}_r)) = s'$ if and only if $\delta^*(s, \omega(r)) = s'$. So, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, define in the context $(p_i, s_j, 0)$ the rule

$$s_j(0) := \langle \sigma_0 = \varepsilon, \sigma_1 = (s_0, \ldots, s_m), \ldots, \sigma_n = (s_0, \ldots, s_m), \sigma_{n+1} = \text{lab}; \\ (R_s^j)_{s \in S} \rangle.$$

   Note that the input strings for each R_s^j are of the form $\bar{w} = w_1 \cdots w_k$, where for $l = 1, \ldots, k$, $w_l \in S^{m+1}$ or $w_l \in \Sigma^*$. The DFA $M_{s,j}$ accepting R_s^j now works as follows: it starts in state s_j, if $w_1 \in S^{m+1}$ then $M_{s,j}$ continues in state s' where s' occurs on the $(j+1)$-th position of w_1 (this is the value of the attribute s_j); otherwise; if $w_1 \in \Sigma^*$ then $M_{s,j}$ continues in state $\delta^*(s_j, w_1)$. Formally, $M_{s,j}$ accepts $\bar{w}$ if there exists $j_1, \ldots, j_k \in \{0, \ldots, m\}$ such that

   - if $w_1 \in S^{m+1}$ then $s_{j_1} = w_1(j+1)$; if $w_1 \in \Sigma^*$ then $s_{j_1} = \delta^*(s_j, w_1)$;
   - for $l = 2, \ldots, k$, if $w_l \in S^{m+1}$ then $s_{j_l} = w_l(j_{l-1} + 1)$; if $w_l \in \Sigma^*$ then $s_{j_l} = \delta^*(s_{j_{l-1}}, w_l)$; and
   - $s_{j_k} = s$.

   Clearly, $M_{s,j}$ can be defined using a number of states polynomial in the size of S. The attribute $result_e$ then becomes true for a node $\mathbf{n}$, when $\mathcal{F}_e(\mathbf{t}_I)(s_0(\mathbf{n})) \in F$ and $\mathcal{F}_e(\mathbf{t}_I)(result_{e_1}(\mathbf{n})) = 1$. So, for $i = 1, \ldots, n$, define in the context $(p_i, result_e, 0)$ the rule

$$result_e(0) := \langle \sigma_0 = (s_0, result_{e_1}), \sigma_1 = \varepsilon, \ldots, \sigma_{n+1} = \varepsilon; \\ R_1 = \{s1 \mid s \in F\}, R_0 = D_e^* - R_1 \rangle.$$

In the following e will always depend on subexpressions e_1 and e_2. Hence, $\mathcal{F}_e$ always consist of $\mathcal{F}_{e_1}$ and $\mathcal{F}_{e_2}$ extended with rules for the new attributes. We will always assume that (apart from the attribute lab) A_{e_1}, A_{e_2} and the set of new attributes are disjoint.

3. $e = e_1 \cup e_2$: A node $\mathbf{n}$ is now selected when

$$\mathcal{F}_e(\mathbf{t}_I)(\mathit{result}_{e_1}(\mathbf{n})) = 1 \text{ or } \mathcal{F}_e(\mathbf{t}_I)(\mathit{result}_{e_2}(\mathbf{n})) = 1.$$

Define $A_e = A_{e_1} \cup A_{e_2} \cup \{\mathit{result}_e\}$ and $D_e = D_{e_1} \cup D_{e_2}$. So, for $i = 1, \ldots, n$, define in the context $(p_i, \mathit{result}_e, 0)$ the rule

$$\mathit{result}_e(0) := \langle \sigma_0 = (\mathit{result}_{e_1}, \mathit{result}_{e_2}), \sigma_1 = \varepsilon, \ldots, \sigma_{n+1} = \varepsilon; \; R_1 = \{01, 10\}, R_0 = D_e^* - R_1 \rangle.$$

4. $e = e_1 - e_2$: A node $\mathbf{n}$ is now selected when

$$\mathcal{F}_e(\mathbf{t}_I)(\mathit{result}_{e_1}(\mathbf{n})) = 1 \text{ and } \mathcal{F}_e(\mathbf{t}_I)(\mathit{result}_{e_2}(\mathbf{n})) = 0.$$

Define $A_e = A_{e_1} \cup A_{e_2} \cup \{\mathit{result}_e\}$ and $D_e = D_{e_1} \cup D_{e_2}$. So, for $i = 1, \ldots, n$, define in the context $(p_i, \mathit{result}_e, 0)$ the rule

$$\mathit{result}_e(0) := \langle \sigma_0 = (\mathit{result}_{e_1}, \mathit{result}_{e_2}), \sigma_1 = \varepsilon, \ldots, \sigma_{n+1} = \varepsilon; \; R_1 = \{10\}, R_0 = D_e^* - R_1 \rangle.$$

5. $e = e_1 < e_2$: Define $A_e = A_{e_1} \cup A_{e_2} \cup \{\mathit{right}, \mathit{result}_e\}$ and $D_e = D_{e_1} \cup D_{e_2}$. Each non-terminal has the inherited attribute *right* such that for a region instance I and a region r, $\mathcal{F}_e(\mathbf{t}_I)(\mathit{right}(\mathbf{n}_r)) = 1$ if there exists a region s such that $r < s$ and $s \in [\![e_2]\!]^I$. Thus, for $j = 1, \ldots, n$, define in the context (p_0, right, j) the rule

$$\mathit{right}(j) := \langle \sigma_0 = \varepsilon, \sigma_1 = \mathit{result}_{e_2}, \ldots, \sigma_n = \mathit{result}_{e_2}, \sigma_{n+1} = \varepsilon; \; R_1, R_0 = (D_e \cup \{\#\})^* - R_1 \rangle,$$

where R_1 is the regular language that contains a string $w_1\#aw_2$, with $w_1, w_2 \in \{0,1\}^*$ and $a \in \{0,1\}$, if w_2 contains a 1. For $i = 1, \ldots, n$ and $j = 1, \ldots, n$, define in the context (p_i, right, j) the rule

$$\mathit{right}(j) := \langle \sigma_0 = \mathit{right}, \sigma_1 = \mathit{result}_{e_2}, \ldots, \sigma_n = \mathit{result}_{e_2}, \sigma_{n+1} = \varepsilon; \; R_1, R_0 = (D_e \cup \{\#\})^* - R_1 \rangle,$$

where R_1 is the regular language that contains a string $aw_1\#bw_2$, with $w_1, w_2 \in \{0,1\}^*$ and $a, b \in \{0,1\}$, if w_2 contains a 1 or $a = 1$. A node $\mathbf{n}$ is then selected when $\mathcal{F}_e(\mathbf{t}_I)(\mathit{result}_{e_1}(\mathbf{n})) = 1$ and $\mathcal{F}_e(\mathbf{t}_I)(\mathit{right}(\mathbf{n})) = 1$. So, for $i = 1, \ldots, n$, define in the context $(p_i, \mathit{result}_e, 0)$ the rule

$$\mathit{result}_e(0) := \langle \sigma_0 = (\mathit{result}_{e_1}, \mathit{right}), \sigma_1 = \varepsilon, \ldots, \sigma_{n+1} = \varepsilon; \; R_1 = \{11\}, R_0 = D_e^* - R_1 \rangle.$$

6. $e = e_1 > e_2$: Similar to the previous case;

7. $e = e_1 \supset e_2$: Define $A_e = A_{e_1} \cup A_{e_2} \cup \{down, result_e\}$ and $D_e = D_{e_1} \cup D_{e_2}$. Each region name has the synthesized attribute *down* such that for a region instance I and a region r, $\mathcal{F}_e(\mathbf{t}_I)(down(\mathbf{n}_r)) = 1$ if there exists a region s such that $r \supset s$ and $s \in [\![e_2]\!]^I$. So, for $i = 1, \dots, n$, define in the context $(p_i, down, 0)$ the rule

$$down(0) := \langle \sigma_0 = \varepsilon, \sigma_1 = (result_{e_2}, down), \dots, \sigma_n = (result_{e_2}, down), \\ \sigma_{n+1} = \varepsilon; R_1, R_0 = D_e^* - R_1 \rangle,$$

where R_1 is the regular language that contains all strings containing at least one 1. A node $\mathbf{n}$ is then selected when

$$\mathcal{F}_e(\mathbf{t}_I)(result_{e_1}(\mathbf{n})) = 1$$

and

$$\mathcal{F}_e(\mathbf{t}_I)(down(\mathbf{n})) = 1.$$

So, for $j = 1, \dots, n$, define in the context $(p_i, result_e, 0)$ the rule

$$result_e(0) := \langle \sigma_0 = (result_{e_1}, down), \sigma_1 = \varepsilon, \dots, \sigma_{n+1} = \varepsilon; \\ R_1 = \{11\}, R_0 = D_e^* - R_1 \rangle.$$

8. $e = e_1 \subset e_2$: Define $A_e = A_{e_1} \cup A_{e_2} \cup \{up, result_e\}$ and $D_e = D_{e_1} \cup D_{e_2}$. Each region name has the inherited attribute *up* such that for a region instance I and a region r, $\mathcal{F}_e(\mathbf{t}_I)(up(\mathbf{n}_r)) = 1$ if there exists a region s such that $r \subset s$ and $s \in [\![e_2]\!]^I$. So, for $j = 1, \dots, n$, define in the context (p_0, up, j) the rule

$$up(j) := \langle \sigma_0 = \varepsilon, \sigma_1 = \varepsilon, \dots, \sigma_{n+1} = \varepsilon; R_1 = \emptyset, R_0 = (D_e \cup \{\#\})^* \rangle.$$

For $i = 1, \dots, n$ and $j = 1, \dots, n$, define in the context (p_i, up, j) the rule

$$up(j) := \langle \sigma_0 = (up, result_{e_2}), \sigma_1 = \varepsilon, \dots, \sigma_{n+1} = \varepsilon; \\ R_1 = \{11\#, 10\#, 01\#\}, R_0 = (D_e \cup \{\#\})^* - R_1 \rangle.$$

A node $\mathbf{n}$ is then selected when $\mathcal{F}_e(\mathbf{t}_I)(result_{e_1}(\mathbf{n})) = 1$ and $\mathcal{F}_e(\mathbf{t}_I)(up(\mathbf{n})) = 1$. So, for $i = 1, \dots, n$, define in the context $(p_i, result_e, 0)$ the rule

$$result_e(0) := \langle \sigma_0 = (result_{e_1}, up), \sigma_1 = \varepsilon, \dots, \sigma_{n+1} = \varepsilon; \\ R_1 = \{11\}, R_0 = D_e^* - R_1 \rangle.$$

■

We need the following definition to state the main result of this section.

Definition 6.7 A Region Algebra expression e over $\mathcal{I}$ is *empty* if for every hierarchical instance I over $\mathcal{I}$, $[\![e]\!]^I = \emptyset$. Two Region Algebra expressions e_1 and e_2 over $\mathcal{I}$ are *equivalent* if for every hierarchical instance I over $\mathcal{I}$, $[\![e_1]\!]^I = [\![e_2]\!]^I$.

Theorem 6.8 *Testing emptiness and equivalence of Region Algebra expressions is in* EXPTIME.

Proof. Although every hierarchical instance of $\mathcal{I} = (S_1, \dots, S_n, \Sigma)$ can be represented as a derivation tree of $G_\mathcal{I}$, not every derivation tree of $G_\mathcal{I}$ is an hierarchical instance. Indeed, if an internal node has no siblings then it represents the same region as it parent. For example, the instance corresponding to the derivation tree

$$\begin{array}{c} U \\ \downarrow \\ \mathbf{Proc} \\ \downarrow \\ \mathbf{Func} \\ \downarrow \\ a \end{array}$$

is not hierarchical because **Proc** and **Func** represent the same region. An extended AG can easily check this condition by making one bottom-up pass through the tree. Another top-down pass then informs all nodes in the tree whether the tree represents an hierarchical instance.

If e is a Region Algebra expression, then we define $\mathcal{F}(e)$ as the extension of the extended AG $\mathcal{F}_e$, given by Theorem 6.6, that simulates e on all hierarchical instances and assigns false to the result attribute of any node of a non-hierarchical instance. Hence, $\mathcal{F}(e)$ is empty if and only if e is empty. Further, if e_1 and e_2 are Region Algebra expressions, then, obviously, $\mathcal{F}(e_1)$ and $\mathcal{F}(e_2)$ are equivalent if and only if e_1 and e_2 are equivalent. Hence, the result follows by Theorem 4.36.

We now describe the construction of $\mathcal{F}(e)$ in more detail. Define $A = A_e \cup \{subhier, hier, result\}$ and $D = D_e$, where A_e and D_e are the attribute set and the semantic domain of $\mathcal{F}_e$, respectively. The semantic rules of $\mathcal{F}(e)$ consists of those of $\mathcal{F}_e$ extended with the rules defining *subhier*, *hier*, and *result*.

Each region name has the synthesized attribute *subhier* such that for a region instance I and a region r, $\mathcal{F}_e(\mathbf{t}_I)(subhier(\mathbf{n}_r)) = 1$ if $\mathbf{t}_{\mathbf{n}_r}$ represents an hierarchical instance. So, for $i = 1, \dots, n$, define in the context $(p_i, subhier, 0)$ the rule

$$subhier(0) := \langle \sigma_0 = \varepsilon, \sigma_1 = (lab, subhier), \dots, \sigma_n = (lab, subhier), \sigma_{n+1} = \text{lab}; \\ R_0, R_1 = D^* - R_0, \rangle,$$

where R_0 is the regular language consisting of all strings containing at least one 0 and all the strings $\{S_1 1, \dots, S_n 1\}$. This rule is correct, since $\mathbf{t}_{\mathbf{n}_r}$ does not represent an hierarchical instance only when at least one of its subtrees does not represent an hierarchical instance, or when $\mathbf{n}_r$ has just one child that, additionally, is labeled with a region name.

Each region name has the inherited attribute *hier* such that for a region instance I and a region r, $\mathcal{F}_e(\mathbf{t}_I)(hier(\mathbf{n}_r)) = 1$ if $\mathbf{t}_I$ represents an hierarchical instance. So, for $j = 1, \ldots, n$, define in the context $(p_0, hier, j)$ the rule

$$hier(j) := \langle \sigma_0 = \varepsilon, \sigma_1 = subhier, \ldots, \sigma_n = subhier, \sigma_{n+1} = \varepsilon; \\ R_1 = 1^*, R_0 = (D \cup \{\#\})^* - R_1 \rangle.$$

For $i = 1, \ldots, n$ and $j = 1, \ldots, n$, define in the context $(p_i, hier, j)$ the rule

$$hier(j) := \langle \sigma_0 = hier, \sigma_1 = \varepsilon, \ldots, \sigma_{n+1} = \varepsilon; R_1 = \{1\#\}, R_0 = (D \cup \{\#\})^* - R_1 \rangle.$$

A node $\mathbf{n}$ is then selected when $\mathcal{F}_e(\mathbf{t}_I)(result_e(\mathbf{n})) = 1$ and $\mathcal{F}_e(\mathbf{t}_I)(hier(\mathbf{n})) = 1$. So, for $j = 1, \ldots, n$, define in the context $(p_i, result, 0)$ the rule

$$result(0) := \langle \sigma_0 = (result_e, hier), \sigma_1 = \varepsilon, \ldots, \sigma_n = \varepsilon; R_1 = \{11\}, R_0 = D^* - R_1 \rangle.$$

■

6.2 Expressiveness of XSLT as a pattern language

In this section we apply the techniques developed to study the expressiveness of query automata and extended AGs to the actual document transformation language XSLT [Cla99]. In particular, we show that XSLT has the ability to issue any MSO pattern at any node in the document. That is, when XSLT arrives at a node it can decide for any unary MSO formula $\varphi(x)$ whether this formula holds at that node and use this information for further processing of the document. Stated as such the result is hardly surprising since full-fledged XSLT allows to call arbitrary Java programs and, therefore, can express all computable document transformations. Our aim, however, is to stress that the navigational mechanism together with a restricted use of variables already suffices to capture the expressiveness of MSO. Hereby, on the one hand, we reveal that core XSLT has a very powerful pattern language at its disposal, and, on the other hand, provide evidence for the robustness of the language.

6.2.1 XSLT

XSL [CD] is a current W3C [Con] proposal for an XML transformation language. Its original primary role was to allow users to write transformations from XML to HTML, thus describing the presentation of the XML document. However, recently, a new working draft emerged, describing XSLT as an extension of XSL for transformation of XML documents into other XML documents [Cla99]. Although XSLT is not intended as a general purpose query language for XML and its definition is still unstable, the variety of questions on the XML newsgroups indicate that already many people use the lotusxsl implementation of XSLT by IBM [IBM99] for their day to day manipulation of XML documents.

Specifically, an XSLT program is a collection of *template rules* where each such rule consist of a pattern and a template (see, for example, the program in Figure 6.4). An important concept regarding the actual computation of XSLT programs is the *current node list*. This list contains all the nodes that still have to be processed by the XSLT program. Hence, at the start of the computation the current node list only contains the root element of the document. The computation proceeds as follows. XSLT removes the first node from the current node list, to which we refer as the *current node*, and tries to apply a pattern to that node (usually such a pattern only refers to the label of the current node). If it succeeds it executes the corresponding template. The latter usually instructs XSLT to produce some XML result and to select a list of nodes for further processing. This list is then prepended to the current node list and the computation proceeds in the same manner with the new list. An XSLT transformation, hence, is a recursive process guided by the structure of the input document.

Next, we introduce by means of an example the important programming constructs of XSLT needed to obtain our main result in the next section. That is, we start by giving an XSLT program simulating a deterministic bottom-up tree automaton T that works on binary trees. For convenience, we restrict T to complete binary trees only. We define $T = (Q, \Sigma, \delta, 0, F)$ as follows: $Q = \{0, 1\}$, $\Sigma = \{a, b\}$, $F = \{0\}$, and

$$\begin{array}{ll}
\delta(a) = 0 & \delta(b) = 1 \\
\delta(0,0,a) = 0 & \delta(0,0,b) = 1 \\
\delta(0,1,a) = 1 & \delta(0,1,b) = 0 \\
\delta(1,0,a) = 1 & \delta(1,0,b) = 0 \\
\delta(1,1,a) = 0 & \delta(1,1,b) = 1.
\end{array}$$

Here, a represents the logical XOR function and b represents the operator $\leftrightarrow$. We use the DTD in Figure 6.3 for representing trees. This figure also displays an XML document corresponding to the tree $a(a(b,b), b(b,a))$. Here, is the usual shorthand for . Note that we do not consider the element mytree as part of the tree $a(a(b,b), b(b,a))$ that it models.

Consider the XSLT program in Figure 6.4. On input $\mathbf{t}$, this program simply outputs the state T assumes at the root of $\mathbf{t}$, that is, the state $\delta^*(\mathbf{t})$. We describe this program in some more detail. The first statement just defines the xsl namespace. It is an obligatory statement and we do not discuss it any further. The XSLT program uses various patterns which we discuss next. The pattern '.' stands for the current node, while '/' means *child of*; in particular, './*' stands for all children of the current node and, for each natural number i, './*[i]' means the i-th child of the current node. The total program consists of three template rules, even though we omitted the template rule

```
<xsl:template match="b" mode="A">  ...  </xsl:template>
```

in Figure 6.4. This rule takes care of the b-labeled nodes and is completely similar to the template rule for the a-labeled nodes, as will become clear after the following discussion.

```
<!ELEMENT tree   (a | b)>

<!ELEMENT a      ((a | b)(a | b)) | ε >

<!ELEMENT b      ((a | b)(a | b)) | ε >
```

```
<tree>
      <a>
            <a>
                  <b/>
                  <b/>
            </a>
            <b>
                  <b/>
                  <a/>
            </b>
      </a>
</tree>
```

Figure 6.3: A DTD for trees over $\{a, b\}$ and an XML document conform to it.

The first template just starts up the process. That is, the pattern `mytree` matches the `<mytree>` node of the input tree; the template itself then outputs the XML tree obtained by applying the template rules in mode `A` on all children of `<mytree>`. Since each `<mytree>` node has only one child, we could also have used the pattern '`./*[1]`'. In this example program, modes play no particular role. In brief, modes are the analogue of states in formal language theory: they allow to handle same nodes in different ways. Their importance will become apparent later on when we have to combine different XSLT subprograms.

The second template rule in our program encodes the transition function of T when the current node carries the label a. More precisely, the first part of the first `choose` statement returns the state 0 when the current node happens to be a leaf (recall that $\delta(a) = 0$). Outputting the value `0` is accomplished by the statement `<xsl:text>0</xsl:text>`; testing for a leaf is done by checking whether the current node has no children. When the current node $\mathbf{n}$ is an internal node, and thus has exactly two children, the program first computes $\delta^*(\mathbf{t_{n1}})$ in the variable `subtree1` by invoking the template rules on $\mathbf{n}1$ in mode A. That is, the output generated by application of the template rules to the first child of $\mathbf{n}$ is assigned to the variable `subtree1`. We point out that all output in this program is restricted to the values `0` and `1`. Hereafter, in the same manner, the program computes $\delta^*(\mathbf{t_{n2}})$ in variable `subtree2`. Depending on the values of `subtree1` and `subtree2` the program outputs the correct state for $\mathbf{n}$. For example, when $\delta^*(\mathbf{t_{n1}}) = 0$ and $\delta^*(\mathbf{t_{n2}}) = 0$ then the program outputs the state $\delta(0, 0, a) = 0$. By outputting the correct state, the program either defines the required state $\delta^*(\mathbf{t})$ if the current node is the child of the `mytree` node, or defines the value of a higher level variable otherwise. From this discussion it should be clear to the reader how to define the template rule for b-labeled nodes and how to simulate tree automata with an arbitrary number of states.

Before continuing, we elaborate on the use of variables in XSLT. In general, the

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

<xsl:template match="mytree" mode="A">
    <xsl:apply-templates select="./*" mode="A"/>
</xsl:template>

<xsl:template match="a" mode="A">
    <xsl:choose>
        <xsl:when test="count(./*) = 0">
            <xsl:text>0</xsl:text>
        </xsl:when>
        <xsl:otherwise>
            <xsl:variable name="subtree1">
                <xsl:apply-templates select="./*[1]" mode="A"/>
            </xsl:variable>
            <xsl:variable name="subtree2">
                <xsl:apply-templates select="./*[2]" mode="A"/>
            </xsl:variable>
            <xsl:choose>
                <xsl:when test="$subtree1 = 0 and $subtree2 = 0">
                    <xsl:text>0</xsl:text>
                </xsl:when>
                <xsl:when test="$subtree1 = 0 and $subtree2 = 1">
                    <xsl:text>1</xsl:text>
                </xsl:when>
                <xsl:when test="$subtree1 = 1 and $subtree2 = 0">
                    <xsl:text>1</xsl:text>
                </xsl:when>
                <xsl:when test="$subtree1 = 0 and $subtree2 = 1">
                    <xsl:text>1</xsl:text>
                </xsl:when>
            </xsl:choose>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Figure 6.4: An example of an XSLT program simulating T.

value of an XSLT variable can be an arbitrary string.[1] However, in our programs, we use variables in a very restricted way: the values of the variables always come from a fixed finite set (in this particular program they are 0 or 1). Therefore, we prefer to see our restricted use of variables as the analogue of the finite state look-ahead as often used in formal language theory [RS97] and not as a dirty programming trick.

In summary, the most important features of the XSLT program in Figure 6.4 are the following: easy navigation through the input document and the use of variables to simulate bottom-up computations.

6.2.2 Main result

In the present section we study the expressiveness of a core subset of the XSLT language that only uses the following features:

1. modes;
2. testing whether the current node is a leaf, the root, the left most or the right most child of its parent;
3. restricted navigation: only moves to direct neighbors (parent, child, sibling); and
4. variables that can only be instantiated by values coming from a fixed finite set.

We refer to this subset as *core XSLT*. Specifically, we will show the following.

Theorem 6.9 *For each MSO formula $\varphi(x)$, there exists a core XSLT program that, for any tree* $\mathbf{t}$ *and node* $\mathbf{n}$*, when applied at* $\mathbf{n}$*, outputs* `true` *if* $\mathbf{t} \models \varphi[\mathbf{n}]$ *and outputs* `false` *otherwise.*

Of course, each above XSLT program can be used as a subroutine by any other XSLT program by capturing the output of the former in some variable. Hence, the pattern language of core XSLT is as expressive as MSO. As an immediate consequence, we can allow XSLT templates of the form

`<xsl:template match="`$\varphi(x)$`" mode=" ... "> ... </xsl:template>`,

where $\varphi(x)$ is an arbitrary MSO formula. We stress that the simulation we are about to describe is by no means an efficient one and is infeasible in practice due to the high complexity of transforming MSO formulas into automata. The only aim of our result is to demonstrate the surprising expressiveness of a small core subset of XSLT. Actually, in addition to formalize a core fragment of XSL corresponding to the working draft of December 1998, Maneth and the present author [MN99] already proposed to extend the pattern language of XSL with MSO formulas.[2]

[1] The XSLT working draft [Cla99] explicitly stresses the conversion of each XML document to a string when they are passed through as values of a variable.

[2] We still like to think they read our paper and only then decided to add those variables.

To prove the above theorem, we could try to simulate query automata or extended AGs in XSLT. However, this is not so straightforward. Indeed, query automata constitute a parallel computation device where different parallel processes have to be combined when making up and down transitions. XSLT can also invoke parallel processes but it is not obvious how they can exchange the information needed for the up and down transitions required to simulate query automata. Extended AGs, on the other hand, have a distributed memory in terms of the attributes. Similarly, it is not clear how this can be simulated directly in XSLT.

Therefore, we proceed as follows. Let $\varphi(x)$ be an MSO formula of quantifier depth k. Then, by Proposition 4.12(1), for any tree $\mathbf{t}$ and node $\mathbf{n}$, $\mathbf{t} \models \varphi[\mathbf{n}]$ only depends on $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$ and $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n})$. By Lemma 4.13, there exists a deterministic bottom-up tree automata (DBTA) computing $\tau_k^{\mathrm{MSO}}(\mathbf{t_n}, \mathbf{n})$ on input $\mathbf{t_n}$. We invite the reader to check the existence of a DBTA computing $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$ on input $(\overline{\mathbf{t_n}}, \mathbf{n})$, where $(\overline{\mathbf{t_n}}, \mathbf{n})$ is the tree $\overline{\mathbf{t_n}}$ with $\mathbf{n}$ as a distinguished node. We could for example distinguish a node by the symbol $*$. That is, let $\mathbf{t}$ be the tree $a(b(b), a)$ and let $\mathbf{n}$ be the first child of the root. Then $(\overline{\mathbf{t_n}}, \mathbf{n})$ is represented by $a(b*, a)$. In outline, the construction of the DBTA computing $\tau_k^{\mathrm{MSO}}(\overline{\mathbf{t_n}}, \mathbf{n})$ on input $(\overline{\mathbf{t_n}}, \mathbf{n})$ is a modification of the construction preceding Theorem 4.11. Only, here, when the automaton reads a leaf labeled $\sigma*$ it assigns the state $\tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma), \mathrm{root}(\mathbf{t}(\sigma)))$ rather than just $\tau_k^{\mathrm{MSO}}(\mathbf{t}(\sigma))$ as for non-distinguished nodes.

It, thus, suffices to show that upon arriving at a node $\mathbf{n}$, XSLT can compute the states $\delta_1^*(\mathbf{t_n})$ and $\delta_2^*((\overline{\mathbf{t_n}}, \mathbf{n}))$, for DBTAs $B_1 = (Q_1, \Sigma, \delta_1, F_1)$ and $B_2 = (Q_2, \Sigma, \delta_2, F_2)$ working over unranked trees. In the following we will outline an XSLT program for computing this subtree type and envelope type with respect to a node $\mathbf{n}$. In particular, the statements

```
<xsl:apply-templates select="." mode="subtree">
```

and

```
<xsl:apply-templates select="." mode="envelope">,
```

issued at a node $\mathbf{n}$ will return the states $\delta_1^*(\mathbf{t_n})$ and $\delta_2^*((\overline{\mathbf{t_n}}, \mathbf{n}))$, respectively. Depending on these values the XSLT program knows whether the current node satisfies the pattern $\varphi(x)$ and can output `true` or `false` accordingly. For expository purposes, we blur the distinction between B_1 and B_2 and talk about an automaton B.

We will use the following small DBTA $B = (Q, \Sigma, \delta, F)$ over unranked trees defined as follows: $Q = \{0, 1\}$, $\Sigma = \{a, b\}$, $F = \{0\}$, $\delta(0, a) = L(M)$, $\delta(1, a) = \{0, 1\}^* - L(M)$, $\delta(0, b) = \{0, 1\}^* - L(M)$, and $\delta(1, b) = L(M)$. Where M is the DFA $(S = \{s_0, \ldots, s_n\}, \{0, 1\}, \delta_M, s_0, F_M)$. Note that B is special in the sense that all regular sets $\delta(0, a)$, $\delta(1, a)$, $\delta(0, b)$, and $\delta(1, b)$ are determined by the DFA M. We chose this DBTA to keep the resulting XSLT program simple. At the end, we indicate how the construction can be generalized for arbitrary DBTAs.

Subtree

To compute $\delta^*(\mathbf{t_n})$ at some node $\mathbf{n}$, we cannot use the same idea as in the simulation of T in Section 6.2.1, for the simple reason that trees are now unranked and we do not have an unbounded number of variables at our disposal (one for each possible subtree). However, the solution to this is quite simple. Suppose the XSLT program arrives at a node $\mathbf{n}$ that has n children and is labeled with an a. Basically, all the XSLT program has to do is simulate M on the string $\delta^*(\mathbf{t_{n1}}) \cdots \delta^*(\mathbf{t_{n\mathit{n}}})$. When this string is accepted the program has to output `0`, otherwise it has to output `1`. To this end, the XSLT program at $\mathbf{n}$ starts up a subproces to define the value of the variable `endstate` as the state $\delta_M(s_0, \delta^*(\mathbf{t_{n1}}) \cdots \delta^*(\mathbf{t_{n\mathit{n}}}))$. Depending whether this state is a final state the program knows whether to output 0 or 1. The value of `endstate` is computed by first jumping to the first child of $\mathbf{n}$ in mode s_0 (recall that s_0 is the start state of M). And then, whenever the program arrives at a node $\mathbf{n}j$ in mode s, it first computes $q = \delta^*(\mathbf{t_{n\mathit{j}}})$ by applying the template rules on itself in mode `subtree`. If $\mathbf{n}j$ is the last child, then it outputs the state $\delta_M(s, q)$ (which will become the value of `endstate`). Conversely, if $\mathbf{n}j$ is not the last child, then the XSLT program jumps to $\mathbf{n}(j+1)$ in mode $\delta_M(s, q)$.

The XSLT program outlined above is depicted in Figure 6.5 and Figure 6.6. We only give a fragment of the needed template rules. The construction of the remaining template rules is similar. We elaborate on some of the details. The test `position() = last()` determines whether the current node is the last child of its parent. The meaning of the functions `position()` and `last()` are the obvious ones. Further, the pattern `following-sibling::*[1]` selects the first right sibling of the current node; `*` indicates that this node may have any label.

We briefly discuss how this construction can be extended to more general DBTAs. Therefore, let B' be a DBTA over the alphabet Σ with state set Q, and let for each $\sigma \in \Sigma$, $\delta(q, \sigma)$ be defined by the automaton $M_{q,\sigma}$. W.l.o.g., we can assume that the state sets of all the DFAs are mutually disjoint. If the XSLT program arrives at a node $\mathbf{n}$ that has n children and is labeled with an a, then it has to simulate every automaton $M_{q,a}$ on the string $\delta^*(\mathbf{t_{n1}}) \cdots \delta^*(\mathbf{t_{n\mathit{n}}})$. This can easily be achieved by defining a variable $\texttt{endstate}_q$, for every q, that starts up a subproces computing the state $\delta_{M_{q,a}}(s_0^{q,a}, \delta^*(\mathbf{t_{n1}}) \cdots \delta^*(\mathbf{t_{n\mathit{n}}}))$, where $\delta_{M_{q,a}}$ is the transition function of $M_{q,a}$ and $s_0^{q,a}$ is the start state of $M_{q,a}$. In the end, only one $\texttt{endstate}_q$ can be assigned a state that is a final state of its automaton $M_{q,a}$. The XSLT program then outputs this q.

Envelope

It now remains to show that an XSLT program arriving at a node $\mathbf{n}$ can compute $\delta^*((\overline{\mathbf{t_n}}, \mathbf{n}))$. For ease of reading, from now on we write $\overline{\mathbf{t_n}}$ for $(\overline{\mathbf{t_n}}, \mathbf{n})$. The program starts by selecting $\mathbf{n}$ in mode envelope-$\delta(\mathrm{lab}_{\mathbf{t}}(\mathbf{n}))$. Suppose during its computation it arrives at some node $\mathbf{m}$ on the path from $\mathbf{n}$ to the root in mode envelope-ℓ where

```
<xsl:template match="a" mode="subtree">
    <xsl:variable name="endstate">
        <xsl:apply-templates select="./*[1]" mode="s_0">
    </xsl:variable>
<xsl:choose>
    <xsl:when test="$endstate = s_0">
        <xsl:text>out</xsl:text>
        % here out is 0 if s_0 ∈ F_M, and is 1 otherwise
        % as δ(a,0) = L(M) and δ(a,1) = {0,1}* − L(M)
    </xsl:when>
          ⋮
    <xsl:when test="$endstate = s_n">
        <xsl:text>out</xsl:text>
        % here out is 0 if s_n ∈ F_M, and is 1 otherwise
        % as δ(a,0) = L(M) and δ(a,1) = {0,1}* − L(M)
    </xsl:when>
</xsl:template>
```

Figure 6.5: An illustrating fragment of an XSLT program computing $\delta^*(\mathbf{t_n})$.

ℓ is 0 or 1. The intention is that $\delta^*((\overline{\mathbf{t_n}})_{\mathbf{m}}) = \ell$.[3] Note that this holds for $\mathbf{m} = \mathbf{n}$. If $\mathbf{m}$ is the root then $\delta^*(\overline{\mathbf{t_n}}) = \ell$ and the XSLT program just outputs ℓ. If not, then it computes $\delta^*((\overline{\mathbf{t_n}})_{\mathbf{p}})$ where $\mathbf{p}$ is the parent of $\mathbf{m}$. We next show how to do this. First, let $\mathbf{p}$ have n children, let a be the label of $\mathbf{p}$, and let $\mathbf{m}$ be the i-th child of $\mathbf{p}$. To obtain the state $\delta^*((\overline{\mathbf{t_n}})_{\mathbf{p}})$, the XSLT program just simulates the automaton M on the string

$$\delta^*(\mathbf{t_{p1}}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}(i-1)}) \cdot \ell \cdot \delta^*(\mathbf{t}_{\mathbf{p}(i+1)}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}n}).$$

Recall that the XSLT program starts from $\mathbf{m}$ and that $\ell = \delta^*((\overline{\mathbf{t_n}})_{\mathbf{m}})$. So, it first has to determine the state $\delta^*_M(s_0, \delta^*(\mathbf{t_{p1}}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}(i-1)}))$. However, it cannot simply jump to the first child of $\mathbf{p}$ and from there on process the string $\delta^*(\mathbf{t_{p1}}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}(i-1)})$, since it cannot remember that $\mathbf{m}$ is the i-th child of $\mathbf{p}$.[4] Therefore, the XSLT program simulates M reversely from $\delta^*(\mathbf{t}_{\mathbf{p}(i-1)})$ till $\delta^*(\mathbf{t_{p1}})$ computing the sets

$$S_s = \{s' \mid \delta^*_M(s', \delta^*(\mathbf{t_{p1}}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}(i-1)})) = s\},$$

for each $s \in S$. Since M is deterministic only one of those sets can contain the start state s_0. If the set S_s contains s_0 then we know that

$$\delta^*_M(s_0, \delta^*(\mathbf{t_{p1}}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}(i-1)})) = s.$$

[3] Here, $(\overline{\mathbf{t_n}})_{\mathbf{m}}$ denotes the subtree of $\overline{\mathbf{t_n}}$ rooted at $\mathbf{m}$.

[4] Of course, this could easily be done by employing XSLT parameters (not discussed anywhere in this section). We, however, want a *core* XSLT program.

```
<xsl:template match="a" mode="s">
    <xsl:variable name="state">
        <xsl:apply-templates select="." mode="subtree">
    </xsl:variable>
    <xsl:choose>
        <xsl:when test="position() = last()">
            <xsl:choose>
                <xsl:when test="$state = 0">
                    <xsl:text>δ_M(s,0)</xsl:text>
                </xsl:when>
                <xsl:when test="$state = 1">
                    <xsl:text>δ_M(s,1)</xsl:text>
                </xsl:when>
            </xsl:choose>
        </xsl:when>
        <xsl:otherwise>
            <xsl:choose>
                <xsl:when test="$state = 0">
                    <xsl:apply-templates select=
                        "following-sibling::*[1]" mode="δ_M(s,0)">
                </xsl:when>
                <xsl:when test="$state = 1">
                    <xsl:apply-templates select=
                        "following-sibling::*[1]" mode="δ_M(s,1)">
                </xsl:when>
            </xsl:choose>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Figure 6.6: An illustrating fragment of an XSLT program computing $\delta^*(\mathbf{t_n})$ (continued).

Specifically, the XSLT program uses the variables left-s, for each $s \in S$, such that left-s is assigned ok (not_ok) whenever $s_0 \in S_s$ ($s_0 \notin S_s$). Concretely, this happens as follows. The program uses all subsets of S as modes and starts with selecting $\mathbf{p}(i-1)$ in mode $\{s\}$, for each $s \in S$, to compute the value of left-s. Suppose a $\mathbf{p}j$ is selected in mode S_j. If $\mathbf{p}j$ is the first child then the XSLT program outputs ok if $\delta_M(s_0, \delta^*(\mathbf{t}_{\mathbf{p}j})) \cap S_j \neq \emptyset$ and outputs not_ok otherwise. If $\mathbf{p}j$ is not the first child then the program selects $\mathbf{p}(j-1)$ in mode S_{j-1} where $S_{j-1} = \{s' \mid \delta_M(s', \delta^*(\mathbf{t}_{\mathbf{p}j})) \cap S_j \neq \emptyset\}$, i.e., all the states in which M can reach a state in S_j by reading the symbol $\delta^*(\mathbf{t}_{\mathbf{p}j})$.

Let s_k be the state such that $s_0 \in S_{s_k}$. Then the XSLT program computes the state $\delta^*_M(s_k, \ell \cdot \delta^*(\mathbf{t}_{\mathbf{p}(i+1)}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}n}))$ by jumping from one sibling to another in the same way as the XSLT program of the previous subsection, but now using the modes env-s for each $s \in S$. The XSLT program acts differently when it reaches the last node: it jumps to $\mathbf{p}$ in mode endstate-s' where $s' = \delta^*_M(s_k, \delta^*(\mathbf{t}_{\mathbf{p}1}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}(i-1)}) \cdot \ell \cdot \delta^*(\mathbf{t}_{\mathbf{p}(i+1)}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}n}))$. The XSLT program then changes to mode envelope-0 or envelope-1 depending on the label of $\mathbf{p}$ and on whether s' is a final state. That is, the program changes to mode envelope-0 if the label of $\mathbf{p}$ is a and $s' \in F_M$, or the label of $\mathbf{p}$ is b and $s' \notin F_M$; conversely, it changes to mode envelope-1 if the label of $\mathbf{p}$ is a and $s' \notin F_M$, or the label of $\mathbf{p}$ is b and $s' \in F_M$. The computation then proceeds from thereon in a similar way until the root is reached.

The strategy outlined above is implemented in Figures 6.7–6.10. Figure 6.7 contains the controlling template rules. The computation is started by selecting a node $\mathbf{n}$ in mode envelope, indicating that the envelope of this node should be computed. Suppose $\mathbf{m}$, a node occurring on the path from $\mathbf{n}$ to the root, is selected in mode envelope-ℓ. If $\mathbf{m}$ is the root node then the program outputs ℓ. Testing whether the current node the root of the modeled tree, is done by checking whether its parent node is labeled by mytree. Otherwise, we start by assuming that $\mathbf{m}$ is always a middle node: that is, $\mathbf{m}$ is never the first or the last child of its parent. This is just a convenient restriction to simplify the presentation of the XSLT program.

The program first determines the state $\delta^*_M(s_0, \delta^*(\mathbf{t}_{\mathbf{p}1}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}(i-1)}))$ by the reverse computation mentioned above. An example template rule for mode S', with $S' \subseteq S$, is given in Figure 6.8. Hereafter, it computes $\delta^*_M(s, \ell \cdot \delta^*(\mathbf{t}_{\mathbf{p}(i+1)}) \cdots \delta^*(\mathbf{t}_{\mathbf{p}n}))$ as explained above. An example of a template rule accomplishing the latter is given in Figure 6.9. Here, .. is the pattern that selects the parent of the current node. Finally, an example template rule for the mode endstate-s is given in Figure 6.10.

The cases where $\mathbf{m}$ can be the first or the last child are straightforward modifications of the presented program. Indeed, only the template rule in Figure 6.7 has to be extended with some consistency checks: (i) if the current node is the left most sibling then the part selecting the first left sibling in the modes $\{s_0\}, \ldots, \{s_n\}$, can be skipped; and (ii) if the current node is the right most sibling then its parent should be selected as in the first part of the template rule in Figure 6.9.

We briefly discuss how this construction can be extended for more general DBTAs. Therefore, let B' be a DBTA over the alphabet Σ with state set Q, and let for each $\sigma \in \Sigma$, $\delta(q, \sigma)$ be defined by the automaton $M_{q,\sigma}$. W.l.o.g., we can assume that the state sets of all the DFAs are mutually disjoint. Suppose the XSLT program arrives

```
<xsl:template match="a" mode="envelope">
    <xsl:apply:templates select="." mode="envelope-$\delta^*(a)$">
</xsl:template>

<xsl:template match="a" mode="envelope-$\ell$">
    <xsl:choose>
        <xsl:when test="count(parent::mytree) > 0">
            <xsl:text>$\ell$</xsl:text>
        </xsl:when>
        <xsl:otherwise>
            <xsl:variable name="left-$s_0$">
                <xsl:apply-templates select=
                    "preceding-sibling::*[1]" mode="{$s_0$}">
            </xsl:variable>
                    ⋮
            <xsl:variable name="left-$s_n$">
                <xsl:apply-templates select=
                    "preceding-sibling::*[1]" mode="{$s_n$}">
            </xsl:variable>
            <xsl:choose>
                <xsl:when test="$left-$s_0$ = ok">
                    <xsl:apply-templates select=
                        "following-sibling::*[1]" mode="$\delta_M(s_0,\ell)$">
                </xsl:when>
                        ⋮
                <xsl:when test="$left-$s_n$ = ok">
                    <xsl:apply-templates select=
                        "following-sibling::*[1]" mode="$\delta_M(s_n,\ell)$">
                </xsl:when>
            </xsl:choose>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Figure 6.7: Template rules computing $\delta^*(\overline{\mathbf{t_n}})$.

```
<xsl:template match="a" mode="$S'$">
    <xsl:variable name="state">
        <xsl:apply-templates select="." mode=subtree>
    </xsl:variable>
    <xsl:choose>
        <xsl:when test="position() = 1">
            <xsl:choose>
                <xsl:when test="$state=0">
                    <xsl:text>out</xsl:text>
                    % here out is ok if $\delta_M(s_0, 0) \cap S' \neq \emptyset$
                    % not_ok otherwise
                </xsl:when>
                <xsl:when test="$state=1">
                    <xsl:text>out</xsl:text>
                    % here out is ok if $\delta_M(s_0, 1) \cap S' \neq \emptyset$
                    % not_ok otherwise
                </xsl:when>
            </xsl:choose>
        </xsl:when>
        <xsl:otherwise>
            <xsl:choose>
                <xsl:when test="$state=0">
                    <xsl:apply-templates select=
                        "preceding-sibling::*[1]"
                               mode="$\{s' \mid \delta_M(s', 0) \cap S' \neq \emptyset\}$">
                </xsl:when>
                <xsl:when test="$state=1">
                    <xsl:apply-templates select=
                        "preceding-sibling::*[1]"
                               mode="$\{s' \mid \delta_M(s', 1) \cap S' \neq \emptyset\}$">
                </xsl:when>
            </xsl:choose>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Figure 6.8: Template rules computing $\delta^*(\overline{\mathbf{t_n}})$ (continued).

```
<xsl:template match="a" mode="env-$s$">
    <xsl:variable name="state">
        <xsl:apply-templates select="." mode="subtree">
    </xsl:variable>
    <xsl:choose>
        <xsl:when test="position() = last()">
            <xsl:choose>
                <xsl:when test="$state = 0">
                    <xsl:apply-templates select=".."
                                         mode="endstate-$\delta_M(s,0)$"/>
                </xsl:when>
                <xsl:when test="$state = 1">
                    <xsl:apply-templates select=".."
                                         mode="endstate-$\delta_M(s,1)$"/>
                </xsl:when>
            </xsl:choose>
        </xsl:when>
        <xsl:otherwise>
            <xsl:choose>
                <xsl:when test="$state = 0">
                    <xsl:apply-templates select=
                        "following-sibling::*[1]" mode="env-$\delta_M(s,0)$">
                </xsl:when>
                <xsl:when test="$state = 1">
                    <xsl:apply-templates select=
                        "following-sibling::*[1]" mode="env-$\delta_M(s,1)$">
                    </xsl:when>
                </xsl:choose>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
```

Figure 6.9: Template rules computing $\delta^*(\overline{\mathbf{t_n}})$ (continued).

```
<xsl:template match="a" mode="endstate-$s$">
    <xsl:apply:templates select="." mode="envelope-$\ell$">
    % $\ell = 0$ iff $s \in F_M$, as $\delta(a,0) = L(M)$ and $\delta(a,1) = \{0,1\}^* - L(M)$
</xsl:template>
```

Figure 6.10: Template rules computing $\delta^*(\overline{\mathbf{t_n}})$ (continued).

in state ℓ at a node $\mathbf{m}$ that is the i-th child of its parent $\mathbf{p}$. Let $\mathbf{p}$ have n children. Then we have to compute the state

$$\delta^*_{M_{q,\sigma}}(s_0^{q,\sigma}, \delta^*(\mathbf{t_{p1}}) \cdots \delta^*(\mathbf{t_{p(\mathit{i}-1)}}) \cdot \ell \cdot \delta^*(\mathbf{t_{p(\mathit{i}+1)}}) \cdots \delta^*(\mathbf{t_{p\mathit{n}}})),$$

for each q and σ. Here, $\delta^*_{M_{q,\sigma}}$ is the transition function of $M_{q,\sigma}$ and $s_0^{q,\sigma}$ is the start state of $M_{q,\sigma}$. For each σ there is only one q_σ such that $M_{q_\sigma,\sigma}$ will accept this string. Remember that our goal is to reach $\mathbf{p}$ in mode $q_{\mathrm{lab}_{\mathbf{t}}(\mathbf{q})}$. So we are done if we can compute all these q_σ's, gather them in a mode, and jump to $\mathbf{p}$. Indeed, upon arriving at $\mathbf{p}$, the program can then determine the state $q_{\mathrm{lab}_{\mathbf{t}}(\mathbf{q})}$ from this mode. For each automaton $M_{q,\sigma}$ the XSLT program finds out the state

$$s^{q,\sigma} = \delta^*_{M_{q,\sigma}}(s_0^{q,\sigma}, \delta^*(\mathbf{t_{p1}}) \cdots \delta^*(\mathbf{t_{p(\mathit{i}-1)}}))$$

by a reverse computation guided by variables like in the previous case. Note that by this reverse computation all started subcomputations end again at $\mathbf{m}$. Hereafter, for each q and σ, the program starts up a subproces computing the state $\delta^*_{M_{q,\sigma}}(s^{q,\sigma}, \ell \cdot \delta^*(\mathbf{t_{p(\mathit{i}+1)}}) \cdots \delta^*(\mathbf{t_{p\mathit{n}}}))$ in a variable $\mathtt{endstate}_{q,\sigma}$. It then can verify which ones are final states, and, hence, can obtain all states q_σ.

6.3 A comparison with other query languages

We now compare the expressiveness of MSO, or query automata and extended AGs for that matter, with FO extended with regular path expressions (denoted by $\mathrm{FO}^{\mathrm{reg}}$). As mentioned in the introduction, we use the latter as an abstraction of the selective power embodied by most of the current query languages for structured documents and semi-structured data [ABS99]. In fact, we show that no $\mathrm{FO}^{\mathrm{reg}}$ sentence can define the class of all trees representing Boolean circuits evaluating to true. This class can be computed by query automata as is shown in Example 5.21. We chose this query in the first place to facilitate the inexpressibility proof. Nevertheless, this and such like queries can be relevant when considering queries themselves as documents [VVV96, NVVV98]. Then, for optimization purposes, one might be interested in those queries containing a sub query that always evaluates to true independent of the input database.

We define $\mathrm{FO}^{\mathrm{reg}}$ in the following way. For each regular expression r we add the predicate $r(x,y)$ to FO with the following semantics. For each $\mathbf{t}$ and nodes $\mathbf{n}$ and $\mathbf{m}$ of $\mathbf{t}$, $\mathbf{t} \models r[\mathbf{n},\mathbf{m}]$ iff $\mathbf{n}$ is an ancestor of $\mathbf{m}$ and the string formed by concatenating the labels on the path from $\mathbf{n}$ to $\mathbf{m}$ (labels of $\mathbf{n}$ and $\mathbf{m}$ included) belongs to the language defined by r. We denote the latter string by $\mathrm{path}_{\mathbf{t}}(\mathbf{n},\mathbf{m})$.

We start by adding the ancestor relation $\prec$ to the logical structure representing trees. Note that this relation is readily definable in $\mathrm{FO}^{\mathrm{reg}}$. Indeed, $x \prec y$ is equivalent to $r(x,y)$ where r is a regular expression denoting the set of all strings.

Proviso 6.10 In the rest of this section we always assume the relation $\prec$ to be available in the logical structures representing trees.

We define a pebble game for the logic $\mathrm{FO}^{\mathrm{reg}}$. The k-round $\mathrm{FO}^{\mathrm{reg}}$ game on trees $(\mathbf{t}, \mathbf{c}_1, \ldots, \mathbf{c}_n)$ and $(\mathbf{s}, \mathbf{d}_1, \ldots, \mathbf{d}_n)$ is played exactly as the k-round FO game with the addition that, if after k rounds the nodes $\mathbf{n}_1, \ldots, \mathbf{n}_k$ and $\mathbf{m}_1, \ldots, \mathbf{m}_k$ are chosen in $\mathbf{t}$ and $\mathbf{s}$, respectively, then not only should $\bar{\mathbf{n}} \to \bar{\mathbf{m}}$ be a partial isomorphism between $(\mathbf{t}, \bar{\mathbf{c}})$ and $(\mathbf{s}, \bar{\mathbf{d}})$ (also taking $\prec$ into account), but also for all $j, l \in \{1, \ldots, k\}$, and $i \in \{1, \ldots, n\}$, the following should hold:

- if $\mathbf{n}_l \prec \mathbf{n}_j$ then $\mathrm{path}_{\mathbf{t}}(\mathbf{n}_l, \mathbf{n}_j) = \mathrm{path}_{\mathbf{s}}(\mathbf{m}_l, \mathbf{m}_j)$;
- if $\mathbf{c}_i \prec \mathbf{n}_j$ then $\mathrm{path}_{\mathbf{t}}(\mathbf{c}_i, \mathbf{n}_j) = \mathrm{path}_{\mathbf{s}}(\mathbf{d}_i, \mathbf{m}_j)$; and
- if $\mathbf{n}_j \prec \mathbf{c}_i$ then $\mathrm{path}_{\mathbf{t}}(\mathbf{n}_j, \mathbf{c}_i) = \mathrm{path}_{\mathbf{s}}(\mathbf{m}_j, \mathbf{d}_i)$.

By adapting the proof of Proposition 2.4, it readily follows that if the duplicator wins the k-round $\mathrm{FO}^{\mathrm{reg}}$ game on $(\mathbf{t}, \bar{\mathbf{c}})$ and $(\mathbf{s}, \bar{\mathbf{d}})$, then for all $\mathrm{FO}^{\mathrm{reg}}$ formulas φ of quantifier depth k, $\mathbf{t} \models \varphi[\bar{\mathbf{c}}]$ iff $\mathbf{s} \models \varphi[\bar{\mathbf{d}}]$. We denote the latter by: $(\mathbf{t}, \bar{\mathbf{c}}) \equiv_k^{\mathrm{reg}} (\mathbf{s}, \bar{\mathbf{d}})$. Note that the pebble game is much too strong for $\mathrm{FO}^{\mathrm{reg}}$ since $(\mathbf{t}, \bar{\mathbf{c}}) \not\equiv_k^{\mathrm{reg}} (\mathbf{s}, \bar{\mathbf{d}})$ does not imply that the spoiler has a winning strategy. Nevertheless, the game will serve our purpose.

We will show that no $\mathrm{FO}^{\mathrm{reg}}$ formula defines the set of trees representing Boolean circuits that evaluate to true. Assume towards a contradiction that there exists such a formula φ of quantifier depth k. We play the $\mathrm{FO}^{\mathrm{reg}}$ game on the trees $\mathrm{AND}(0, c, h)$ and $\mathrm{AND}(1, c, h)$, and $\mathrm{OR}(0, c, h)$ and $\mathrm{OR}(1, c, h)$ defined as follows. For all $c \geq 1$ and $i \in \{0, 1\}$, define $\mathrm{AND}(i, c, 0) = \mathrm{OR}(i, c, 0) = i$. For all $c, h \geq 1$, define

$$
\begin{aligned}
\mathrm{AND}(0,c,h) &:= \mathrm{AND}(\mathrm{OR}(1,c,h-1)^{\times c}, \mathrm{OR}(0,c,h-1), \mathrm{OR}(1,c,h-1)^{\times c}),\\
\mathrm{OR}(0,c,h) &:= \mathrm{OR}(\mathrm{AND}(0,c,h-1)^{\times 2c+1}),\\
\mathrm{AND}(1,c,h) &:= \mathrm{AND}(\mathrm{OR}(1,c,h-1)^{\times 2c+1}), \text{ and}\\
\mathrm{OR}(1,c,h) &:= \mathrm{OR}(\mathrm{AND}(0,c,h-1)^{\times c}, \mathrm{AND}(1,c,h-1),\\
&\qquad\qquad\qquad\qquad \mathrm{AND}(0,c,h-1)^{\times c}).
\end{aligned}
$$

Here, for a tree $\mathbf{t}$, $\mathbf{t}^{\times i}$ denotes the sequence $\mathbf{t}, \ldots, \mathbf{t}$ (i times). See Figure 6.11 for a graphical representation. We start with some observations concerning these trees. Each of the above trees is of height h (cf. definition of height in Section 2.4), and all internal nodes have exactly $2c+1$ children. Further, all nodes occurring on the same height are labeled with the same label and the labels of the levels alternate between AND and OR. The root of each $\mathrm{AND}(i, c, h)$ is labeled with AND, while the root of each $\mathrm{OR}(i, c, h)$ is labeled with OR. We invite the reader to verify that all trees $\mathrm{AND}(0, c, h)$ and $\mathrm{OR}(0, c, h)$ evaluate to 0, while all trees $\mathrm{AND}(1, c, h)$ and $\mathrm{OR}(1, c, h)$ evaluate to 1. We refer to the former as 0-trees and to the latter as 1-trees.

We will show that for $\sigma \in \{\mathrm{AND}, \mathrm{OR}\}$, $\sigma(0, c, h) \equiv_k^{\mathrm{reg}} \sigma(1, c, h)$ for sufficiently large values c and h. However, rather than playing the k-round $\mathrm{FO}^{\mathrm{reg}}$ game on these structures, we just play the ordinary k-round FO game, but require the duplicator to make *depth preserving* moves. That is, whenever the spoiler picks a node, the duplicator is required to pick a node of the same depth (cf. definition of depth in Section 2.4). Due to the structure of the above defined trees it readily follows that,

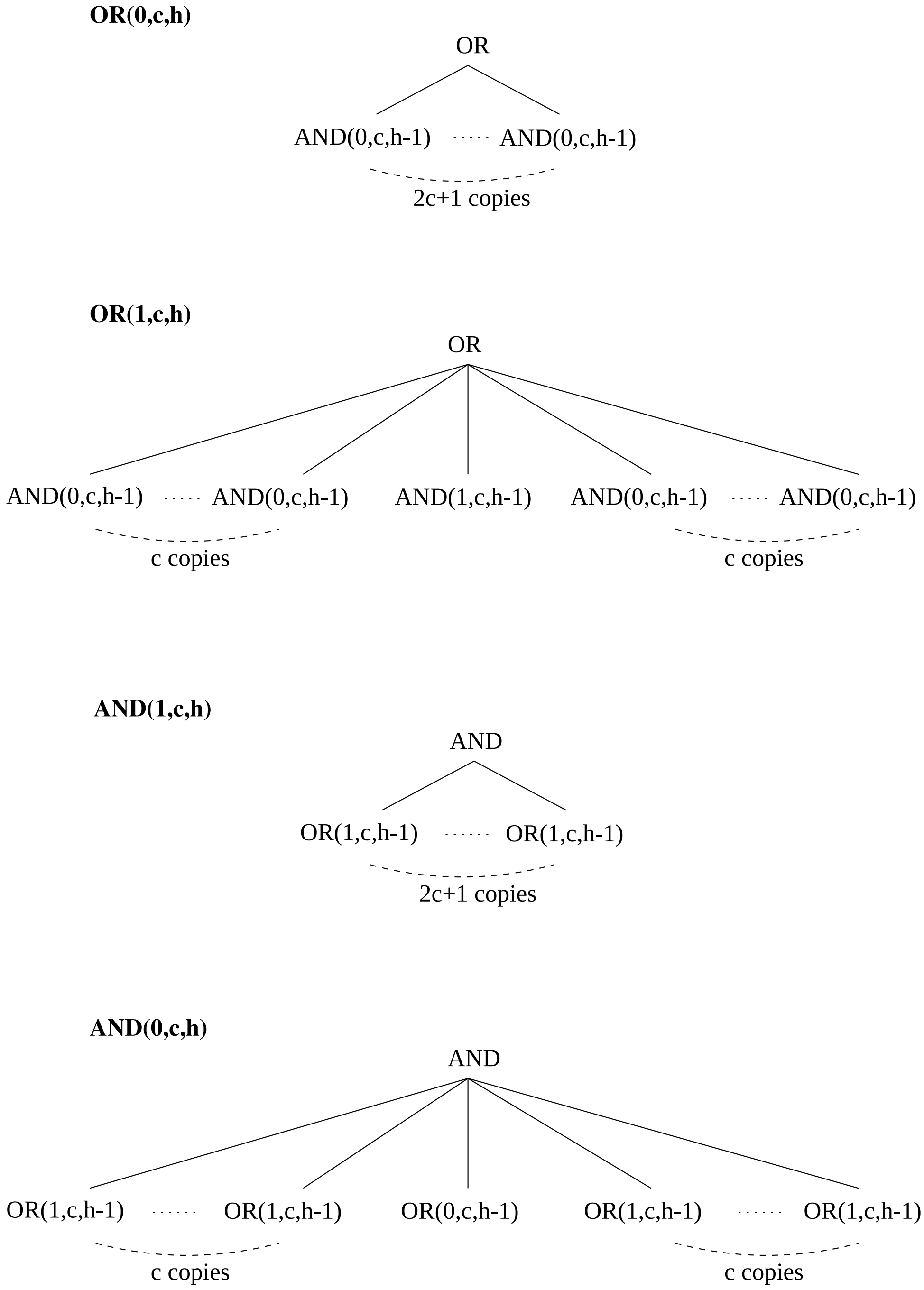

Figure 6.11: The trees $\mathrm{OR}(0, c, h)$, $\mathrm{OR}(1, c, h)$, $\mathrm{AND}(1, c, h)$, and $\mathrm{AND}(0, c, h)$.

if $\mathbf{n}_1$ and $\mathbf{n}_2$ are nodes in $\sigma(0,c,h)$ with $\mathbf{n}_1 \prec \mathbf{n}_2$ and $\mathbf{m}_1$ and $\mathbf{m}_2$ are nodes in $\sigma(1,c,h)$ with $\mathbf{m}_1 \prec \mathbf{m}_2$, such that the depth of $\mathbf{n}_1$ equals the depth of $\mathbf{m}_1$ and the depth of $\mathbf{n}_2$ equals the depth of $\mathbf{m}_2$, then $\text{path}_{\sigma(0,c,h)}(\mathbf{n}_1, \mathbf{n}_2) = \text{path}_{\sigma(1,c,h)}(\mathbf{m}_1, \mathbf{m}_2)$. Consequently, if the duplicator has a depth preserving winning strategy in the k-round FO game on $(\sigma(0,c,h), \bar{\mathbf{c}})$ and $(\sigma(1,c,h), \bar{\mathbf{d}})$ then he also wins the k-round FO^{reg} game on those structures (recall that the relation $\prec$ is added to the logical structures representing trees).

We note that we could also have adopted height preserving moves rather than depth preserving ones. The latter ones just make the statement of Lemma 6.11 easier. Before we prove this lemma, we introduce some more notation. If $\mathbf{t}$ is a tree with nodes $\bar{\mathbf{n}}$, then $\overline{\mathbf{t}_{\bar{\mathbf{n}}}}$ denotes $\mathbf{t}$ without the subtrees rooted at the nodes in $\bar{\mathbf{n}}$ (but keeping the nodes in $\bar{\mathbf{n}}$). Further, if $\mathbf{c}$ is a node, then $\mathbf{c}+1$ and $\mathbf{c}-1$ denote its right and left sibling, respectively.

The following lemma allows us to reduce an FO game on two trees to several FO games on different parts of those trees.

Lemma 6.11 *Let $\mathbf{t}$ and $\mathbf{s}$ be two trees with nodes $\mathbf{c}_1, \ldots, \mathbf{c}_n$ and $\mathbf{d}_1, \ldots, \mathbf{d}_n$, such that no $\mathbf{c}_i$ is an ancestor of a $\mathbf{c}_j$, and no $\mathbf{d}_i$ is an ancestor of a $\mathbf{d}_j$ for $i \neq j$, and for each $i = 1, \ldots, n$, the depth of $\mathbf{c}_i$ equals the depth of $\mathbf{d}_i$. Further, let $\mathbf{n}$ be a node in $\mathbf{t}$ and let $\mathbf{m}$ be a node in $\mathbf{s}$ such that the depth of $\mathbf{n}$ equals the depth of $\mathbf{m}$, and for $i = 1, \ldots, n$, $\mathbf{n}$ occurs in $\mathbf{t}_{\mathbf{c}_i}$ iff $\mathbf{m}$ occurs in $\mathbf{s}_{\mathbf{d}_i}$. Let $k \geq 0$. If the duplicator has a depth preserving winning strategy in $G_k^{\text{FO}}(\overline{\mathbf{t}_{\bar{\mathbf{c}}}}, \bar{\mathbf{c}}, \mathbf{n}; \overline{\mathbf{s}_{\bar{\mathbf{d}}}}, \bar{\mathbf{d}}, \mathbf{m})$, and in $G_k^{\text{FO}}(\mathbf{t}_{\mathbf{c}_i}, \mathbf{c}_i, \mathbf{n}; \mathbf{s}_{\mathbf{d}_i}, \mathbf{d}_i, \mathbf{m})$, for $i = 1, \ldots, n$, then the duplicator has a depth preserving winning strategy in $G_k^{\text{FO}}(\mathbf{t}, \mathbf{n}; \mathbf{s}, \mathbf{m})$.*[5]

Proof. The basic idea is to combine the winning strategies of the duplicator on the respective subtrees into a winning strategy on the whole structures like, for instance, in the case of strings in Proposition 2.6. That is, whenever the spoiler chooses a node in one of the substructures the duplicator chooses a node in the corresponding substructure, according to his strategy in the game on these substructures. As the substructures have nodes in common, we have to argue that our strategy is well defined. To this end, it suffices to note that whenever the spoiler picks $\mathbf{c}_i$ ($\mathbf{d}_i$) in a game on a substructure, the duplicator is forced to pick $\mathbf{d}_i$ ($\mathbf{c}_i$) as both $\mathbf{c}_i$ and $\mathbf{d}_i$ are distinguished constants in all substructures where they appear.

At the end of a game on the whole structure, the selected nodes define partial isomorphisms for all pairs of respective substructures. To ensure that they also define a partial isomorphism between the entire structures, we only have to note that the ancestor relation is always preserved between components since we only pick nodes from corresponding substructures. ∎

In the rest of this section we will prove the following lemma. From this lemma it readily follows that φ cannot define the class of all trees representing Boolean circuits evaluating to true. We abbreviate $\tau_k^{\text{MSO}}(\mathbf{t}, \text{root}(\mathbf{t}))$ by $\tau_k^{\text{MSO}}(\mathbf{t}, \text{root})$ for each tree $\mathbf{t}$.

[5] We abuse notation here. When we write, for example, $G_k^{\text{FO}}(\mathbf{t}_{\mathbf{c}_i}, \mathbf{n}; \mathbf{s}_{\mathbf{d}_i}, \mathbf{m})$, then we mean the game $G_k^{\text{FO}}(\mathbf{t}_{\mathbf{c}_i}; \mathbf{s}_{\mathbf{d}_i})$ if $\mathbf{n}$ and $\mathbf{m}$ do not occur in $\mathbf{t}_{\mathbf{c}_i}$ and $\mathbf{s}_{\mathbf{d}_i}$, respectively.

Lemma 6.12 *For all $k \geq 1$, $r \geq 2^{k-1}+1$, and $\ell > 2k$,*

$$(\mathrm{OR}(0,r,\ell),\mathrm{root}) \equiv_k^{\mathrm{FO}} (\mathrm{OR}(1,r,\ell),\mathrm{root})$$

and

$$(\mathrm{AND}(0,r,\ell),\mathrm{root}) \equiv_k^{\mathrm{FO}} (\mathrm{AND}(1,r,\ell),\mathrm{root}).$$

Proof. Clearly, the trees $\mathrm{AND}(0,c,h)$ and $\mathrm{AND}(1,c,h)$, and the trees $\mathrm{OR}(0,c,h)$ and $\mathrm{OR}(1,c,h)$ are isomorphic if the labels of nodes are not taken into account. Let π denote this isomorphism (it will always be clear from the context whether we consider AND or OR trees and what the values of c and h are). We can say even more: the trees $\mathrm{OR}(0,c,h)$ and $\mathrm{OR}(1,c,h)$, and $\mathrm{AND}(0,c,h)$ and $\mathrm{AND}(1,c,h)$ are identical apart from the label of one leaf. That is, there is only one node that distinguishes these trees. We introduce a special name for the nodes on the path from the root to this leaf: we call them d-nodes. The reader is invited to check that subtrees rooted at d-nodes in $\mathrm{OR}(0,c,h)$ and $\mathrm{AND}(0,c,h)$ are 0-trees while they are 1-trees in $\mathrm{OR}(1,c,h)$ and $\mathrm{AND}(1,c,h)$.

The proof proceeds by induction on k and clearly holds for $k=0$. We only give the argument for the trees $\mathbf{t} = \mathrm{OR}(0,r,\ell)$ and $\mathbf{s} = \mathrm{OR}(1,r,\ell)$. The case of $\mathrm{AND}(0,r,\ell)$ and $\mathrm{AND}(1,r,\ell)$ is similar.

Suppose the spoiler picks his first node $\mathbf{n}$ in $\mathbf{t}$. It then suffices to provide a node $\mathbf{m}$ of $\mathbf{s}$ of the same depth such that the duplicator has a depth preserving winning strategy in $G_{k-1}^{\mathrm{FO}}(\mathbf{t},\mathbf{n};\mathbf{s},\mathbf{m})$.

Let $\mathbf{c}$ be the d-node of height $2k-1$. If $\mathbf{n}$ is not in $\mathbf{t_c}$ then the duplicator chooses the node $\mathbf{m} = \pi(\mathbf{n})$ in $\mathbf{s}$ and the game continues on $(\mathbf{t_c},\mathbf{c})$ and $(\mathbf{s}_{\pi(\mathbf{c})},\pi(\mathbf{c}))$, and on $(\overline{\mathbf{t_c}},\mathbf{c},\mathbf{n})$ and $(\overline{\mathbf{s}_{\pi(\mathbf{c})}},\pi(\mathbf{c}),\mathbf{m})$. In both games the duplicator has a depth preserving winning strategy for the remaining $k-1$ rounds. Indeed, in the first case this follows by induction (note that $2k-1 > 2(k-1)$) and in the second case because the two trees are isomorphic (recall that $\mathbf{c}$ and $\pi(\mathbf{c})$ are d-nodes). By Lemma 6.11, the duplicator has a depth preserving winning strategy in $G_{k-1}^{\mathrm{FO}}(\mathbf{t},\mathbf{n};\mathbf{t},\mathbf{m})$. This concludes the proof of this case.

We now turn to the harder case. Suppose the spoiler picks $\mathbf{n}$ in $\mathbf{t_c}$. We refrain from choosing $\mathbf{m}$ in $\mathbf{s}_{\pi(\mathbf{c})}$, as $\mathbf{t_c} = \mathrm{AND}(0,r,2k-1)$ and $\mathbf{s}_{\pi(\mathbf{c})} = \mathrm{AND}(1,r,2k-1)$, and the spoiler might distinguish $(\mathbf{t_c},\mathbf{c},\mathbf{n})$ and $(\mathbf{s}_{\pi(\mathbf{c})},\pi(\mathbf{c}),\mathbf{m})$ in the remaining $k-1$ moves. Observe that the tree rooted at the right sibling of $\mathbf{c}$ is $\mathrm{AND}(0,r,2k-1)$ and the tree rooted at the left sibling of $\pi(\mathbf{c})$ is also $\mathrm{AND}(0,r,2k-1)$. We are going to fool the spoiler by mapping $\mathbf{t_c}$ to $\mathbf{s}_{\pi(\mathbf{c})-1}$ and $\mathbf{t}_{\mathbf{c}+1}$ to $\mathbf{s}_{\pi(\mathbf{c})}$. The duplicator thus chooses $\mathbf{m}$ in $\mathbf{s}_{\pi(\mathbf{c})-1}$ on exact the same position as $\mathbf{n}$ occurs in $\mathbf{t_c}$. The game now continues on $(\mathbf{t_c},\mathbf{c},\mathbf{n})$ and $(\mathbf{s}_{\pi(\mathbf{c})-1},\pi(\mathbf{c})-1,\mathbf{m})$, on $(\mathbf{t}_{\mathbf{c}+1},\mathbf{c}+1)$ and $(\mathbf{s}_{\pi(\mathbf{c})},\pi(\mathbf{c}))$, and on $(\overline{\mathbf{t}_{\mathbf{c},\mathbf{c}+1}},\mathbf{c},\mathbf{c}+1)$ and $(\overline{\mathbf{t}_{\pi(\mathbf{c})-1,\pi(\mathbf{c})}},\pi(\mathbf{c})-1,\pi(\mathbf{c}))$. We will argue that in all three cases the duplicator has a depth preserving winning strategy for the remaining $k-1$ moves. Then, by Lemma 6.11, the duplicator has a depth preserving winning strategy in $G_{k-1}^{\mathrm{FO}}(\mathbf{t},\mathbf{n};\mathbf{s},\mathbf{m})$. In the first two cases it is immediate that the duplicator wins. Indeed, in the first case both trees are isomorphic and in the second case the winning strategy follows by induction. The reason for the third case is that there are so many children that the spoiler cannot possibly distinguish the pair $(\mathbf{c}, \mathbf{c}+1)$ from the pair $(\pi(\mathbf{c})-1, \pi(\mathbf{c}))$ in the remaining $k-1$ moves. We elaborate on this. Let $\mathbf{p}$ be the parent of $\mathbf{c}$. Clearly, $(\overline{\mathbf{t_p}}, \mathbf{p})$ and $(\overline{\mathbf{s}_{\pi(\mathbf{p})}}, \pi(\mathbf{p}))$ are isomorphic. Thus, whenever the spoiler picks a node in $\overline{\mathbf{t_p}}$ or $\overline{\mathbf{s}_{\pi(\mathbf{p})}}$, the duplicator responds with the same one from $\overline{\mathbf{s}_{\pi(\mathbf{p})}}$ or $\overline{\mathbf{t_p}}$, respectively. Hence, it suffices to restrict attention to moves of the spoiler in $(\overline{\mathbf{t}_{\mathbf{c},\mathbf{c}+1}})_{\mathbf{p}}$ and $(\overline{\mathbf{s}_{\pi(\mathbf{c})-1,\pi(\mathbf{c})}})_{\mathbf{p}}$.

We first recall a helpful proposition on strings (see for example [EF95, Imm98]):

Proposition 6.13 *Let w and v be strings of length at least 2^{k-1} over the unary alphabet $\{\sigma\}$. Then the duplicator wins $G^{\mathrm{FO}}_{k-1}(w; v)$.*

From this proposition one easily obtains the following: the duplicator wins $G^{\mathrm{FO}}_{k-1}(w, r+1, r+2; v, r, r+1)$, for all strings w and v of length $2r+1$ over the unary alphabet $\{\sigma\}$.

We use the winning strategy of the duplicator in the game $G^{\mathrm{FO}}_{k-1}(w, r+1, r+2; v, r, r+1)$ to answer moves of the spoiler in the trees $(\overline{\mathbf{t}_{\mathbf{c},\mathbf{c}+1}})_{\mathbf{p}}$ and $(\overline{\mathbf{s}_{\pi(\mathbf{c})-1,\pi(\mathbf{c})}})_{\pi(\mathbf{p})}$. We focus on moves of the spoiler in the former tree. Clearly, the duplicator answers with $\pi(\mathbf{c})-1$ and $\pi(\mathbf{c})$, whenever the spoiler picks $\mathbf{c}$ and $\mathbf{c}+1$, respectively. Recall that $\mathbf{p}$ and $\pi(\mathbf{p})$ have only trees $\mathrm{AND}(0, r, 2k-1)$ rooted at their children (forgetting about the children $\mathbf{c}$, $\mathbf{c}+1$, $\pi(\mathbf{c})-1$, and $\pi(\mathbf{c})$). Suppose the spoiler picks an element in one subtree $\mathbf{t}_{\mathbf{p}i}$, with $\mathbf{p}i$ different from $\mathbf{c}$ and $\mathbf{c}+1$. Then the duplicator chooses exactly the same element in some subtree $\mathbf{s}_{\pi(\mathbf{p})j}$ which the duplicator chooses according to his winning strategy in the game on the strings. That is, we examine the game on $(w, r+1, r+2)$ and $(v, r, r+1)$ where the spoiler picks the i-th element. If, in this game, the duplicator replies with the j-th element, then the duplicator uses the subtree $\mathbf{s}_{\pi(\mathbf{p})j}$. The same strategy holds when the spoiler picks elements in $(\overline{\mathbf{s}_{\pi(\mathbf{c})-1,\pi(\mathbf{c})}})_{\pi(\mathbf{p})}$. This concludes the proof of the lemma. ∎

We thus obtain:

Theorem 6.14 *MSO is strictly more expressive than FO^{reg}.*

6.4 The encoding of ranked into unranked trees

There are several ways to encode unranked trees by, for example, binary ones. To make a clear distinction between binary and unranked trees, for the logical representation of trees, we will make use of the successor relations S_1 and S_2 (defining the first child and the second child of a node) in the former case and stick to the relations E and $<$ in the latter case. Using these successor relations it is possible to define trees with nodes that have a second child but have no first child. We will exploit this in the encoding described next.

For an unranked tree $\mathbf{t}$, we define the binary tree $\mathrm{enc}(\mathbf{t})$ as follows:

- the set of nodes of $\mathrm{enc}(\mathbf{t})$ consists of the the set of nodes of $\mathbf{t}$;
- all nodes in $\mathrm{enc}(\mathbf{t})$ have the same label as in $\mathbf{t}$;

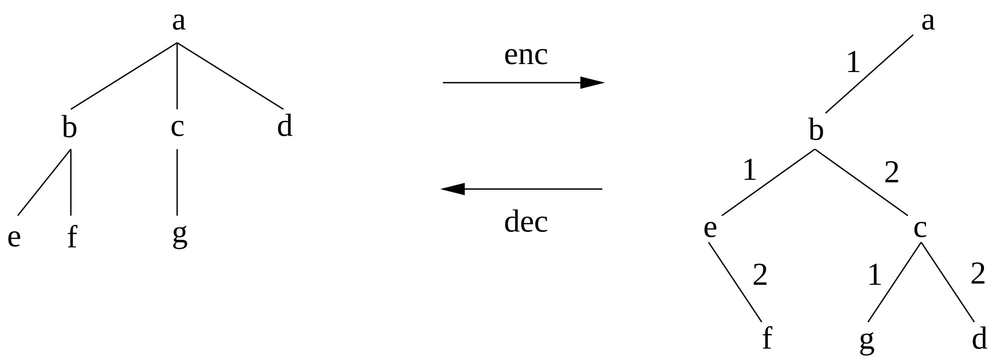

Figure 6.12: Example of enc and dec applied to a tree.

- the root of enc($\mathbf{t}$) is the root of $\mathbf{t}$;
- for all nodes $\mathbf{n}$ and $\mathbf{m}$ of $\mathbf{t}$:
  - $\mathbf{n}$ is the first child of $\mathbf{m}$ in enc($\mathbf{t}$) iff $\mathbf{n}$ is the first child of $\mathbf{m}$ in $\mathbf{t}$; and
  - $\mathbf{n}$ is the second child of $\mathbf{m}$ in enc($\mathbf{t}$) iff $\mathbf{n}$ is the first right sibling of $\mathbf{m}$ in $\mathbf{t}$.

To define the decoding, we introduce the following notion. We say that a node $\mathbf{n}$ is a *right descendant* of $\mathbf{m}$ in a binary tree if the path from $\mathbf{m}$ to $\mathbf{n}$ ($\mathbf{m}$ excluded) only contains nodes that are the second child of their parent. For a binary tree $\mathbf{t}$, we define the unranked tree dec($\mathbf{t}$) as follows:

- the set of nodes of dec($\mathbf{t}$) is the set of nodes of $\mathbf{t}$;
- a node in dec($\mathbf{t}$) has the same label as in $\mathbf{t}$;
- the root of dec($\mathbf{t}$) is the root of $\mathbf{t}$;
- for all nodes $\mathbf{n}$ and $\mathbf{m}$ of $\mathbf{t}$:
  - $\mathbf{n}$ is a child of $\mathbf{m}$ in dec($\mathbf{t}$) iff $\mathbf{n}$ is the first child of $\mathbf{m}$ in $\mathbf{t}$, that is $\mathbf{n} = \mathbf{m}1$, or $\mathbf{n}$ is a right descendant of $\mathbf{m}1$; and
  - $\mathbf{n}$ is a left sibling of $\mathbf{m}$ in dec($\mathbf{t}$) iff $\mathbf{m}$ is right descendant of $\mathbf{m}$.

We depicted in Figure 6.12 an example of the above described encoding and decoding.

It is not so difficult, but rather tedious, to prove that for every unranked tree language $\mathcal{T}$, enc($\mathcal{T}$) is recognizable iff $\mathcal{T}$ is recognizable. Of course, our automaton model for binary trees is not defined for trees with nodes that can have a second child without having a first child. However, such tree can readily be encoded by using a special symbol, say #, to denote the absence of the first child. The binary tree in Figure 6.12, would then be represented by $a(b(e(\#, f), c(g, d)))$. In one direction the proof consists of a simulation of an unranked tree automaton by a ranked one working on the encoding of unranked trees. The crux lies in the simulation of the DFAs, encoding the transition function of the unranked tree automaton, on paths consisting solemnly of nodes that are right children of their parent (as these nodes are siblings in the unranked tree). The proof of the other direction consists of the reverse simulation. Rather than going into the details of this tedious simulation we employ our results obtained in the previous chapters to prove a much more general result.

First we note that both enc and dec are MSO definable in terms of each other. Before we illustrate this, we formally define how MSO formulas can encode and decode trees. In the ranked case we focus on trees of rank m, for some m. Hence, we make use of the successor relations $S_1, \ldots, S_m$.

An *MSO definable encoding* and *decoding* are tuples $e = (\psi_1(x,y), \ldots, \psi_m(x,y))$ and $d = (\psi_E(x,y), \psi_<(x,y))$ of MSO formulas over the vocabulary $\{E, <, (O_\sigma)_{\sigma\in\Sigma}\}$ and $\{S_1, \ldots, S_m, (O_\sigma)_{\sigma\in\Sigma}\}$, respectively. For an unranked tree $\mathbf{t}$, $e(\mathbf{t})$ is the ranked tree with domain Nodes($\mathbf{t}$), where $S_i := \{(\mathbf{n},\mathbf{m}) \mid \mathbf{t} \models \psi_i[\mathbf{n},\mathbf{m}]\}$, for $i = 1, \ldots, m$. For a ranked tree $\mathbf{t}$, $d(\mathbf{t})$ is the unranked tree with domain Nodes($\mathbf{t}$), where $E := \{(\mathbf{n},\mathbf{m}) \mid \mathbf{t} \models \psi_E[\mathbf{n},\mathbf{m}]\}$ and $< := \{(\mathbf{n},\mathbf{m}) \mid \mathbf{t} \models \psi_<[\mathbf{n},\mathbf{m}]\}$. We will only consider encodings and decodings such that $e(\mathbf{t})$ and $d(\mathbf{s})$ are in fact trees for all tree $\mathbf{t}$ and $\mathbf{s}$.

Now, clearly, enc and dec are MSO definable:

$$\psi_1(x,y) := E(x,y) \wedge \neg(\exists z)(z < y),$$

$$\psi_2(x,y) := x < y \wedge \neg(\exists z)(x < z \wedge z < y),$$

$$\psi_E(x,y) := S_1(x,y) \vee (\exists z)(S_1(x,z) \wedge \mathrm{TC}[S_2(z,y)](z,y)),$$

and

$$\psi_<(x,y) := \mathrm{TC}[S_2(x,y)](x,y).$$

Here TC denotes the (monadic) transitive closure operator which can readily be expressed in MSO (cf. Example 2.2).

For a tree language $\mathcal{T}$, we write $e(\mathcal{T})$ for $\{e(\mathbf{t}) \mid \mathbf{t} \in \mathcal{T}\}$ and $d(\mathcal{T})$ for $\{d(\mathbf{t}) \mid \mathbf{t} \in \mathcal{T}\}$. Further, we say that e and d are each others inverses when for every unranked tree $\mathbf{t}$ and ranked tree $\mathbf{s}$, $d(e(\mathbf{t})) = \mathbf{t}$ and $e(d(\mathbf{s})) = \mathbf{s}$. Note that enc and dec are each others inverses.

However, by Theorem 2.13 and Theorem 4.11, we can state the following general theorem which can be proved by simply substituting $\psi_i(x,y)$ for $S_i(x,y)$, $\psi_E(x,y)$ for $E(x,y)$, and $\psi_<(x,y)$ for $x < y$:

Corollary 6.15 *For every MSO definable encoding e and decoding d:*

- *for every unranked tree language $\mathcal{T}_u$, if $e(\mathcal{T}_u)$ is recognizable then $\mathcal{T}_u$ is recognizable, and*

- *for every ranked tree language $\mathcal{T}_r$, if $d(\mathcal{T}_r)$ is recognizable then $\mathcal{T}_r$ is recognizable.*

Hence, if e and d are each others inverses then for every unranked tree language $\mathcal{T}_u$ and every ranked tree language $\mathcal{T}_r$, $e(\mathcal{T}_u)$ is recognizable iff $\mathcal{T}_u$ is recognizable, and $d(\mathcal{T}_r)$ is recognizable iff $\mathcal{T}_r$ is recognizable.

Let $\mathcal{T}_\Sigma$ be the set of all Σ-trees and let $\mathcal{T}_\Sigma^m \subset \mathcal{T}_\Sigma$ be the set of all trees of rank m. Theorem 5.18 and Theorem 5.29 again allow to state a general theorem:

Corollary 6.16 *Let e and d be an MSO definable encoding and decoding which are each others inverses. Then, a query is is expressible by an* SQA^u *iff it is expressible by a* QA^r *on* $e(\mathcal{T}_\Sigma)$*; and, a query is expressible by a* QA^r *iff it is expressible by a* QA^u *on* $d(\mathcal{T}_\Sigma^m)$*.*

In conclusion, we point out that the above correspondence between ranked and unranked trees does not always apply. For instance, Maneth and the present author defined a generalization of the top-down tree transducer model for unranked trees as a formal model for XSL [MN99]. This tree transducer model is much more powerful than the standard tree transducer model on the encoding enc of unranked trees.

We point out that several researchers have used the encoding of unranked trees into ranked once [MN99, MSV99, PQ68].

6.5 Implementing RAGs on top of a deductive database system

To conclude this chapter, we propose a design to implement the BAGs and RAGs studied in Chapter 3. Specifically, we want to show that deductive databases offer a natural platform on top of which such an implementation becomes remarkably straightforward. Since BAGs can be seen as special cases of RAGs, we focus attention on the latter ones. Note that this section concerns *ranked* trees only. The presentation of this section will be rather terse.

6.5.1 Datalog$^\neg$

In the present section, we define **datalog**$^\neg$. We refer to the book by Abiteboul, Hull, and Vianu [AHV95] for more background. We start by fixing a vocabulary τ containing only relation names. A **datalog**$^\neg$ *rule* r is an expression of the form

$$Q_0(\bar{x}_0) \leftarrow Q_1(\bar{x}_1), \ldots, Q_m(\bar{x}_m),$$

where

- Q_0 is an atomic formula of the form $R(\cdots)$ with $R \in \tau$; and
- for each $i = 1, \ldots, n$, $Q_i(\bar{x}_i)$ is an atomic formula or negation of an atomic formula. That is, each $Q_i(\bar{x}_i)$ is either $R(\bar{x}_i)$ or $\neg R(\bar{x}_i)$ where $R \in \tau$, or is of the form $x = y$ or $\neg(x = y)$.

We refer to atomic formulas or negations of atomic formulas as *literals*. In particular, we call atomic formulas *positive* literals and negations of atomic formulas *negative* literals. A **datalog**$^\neg$ *program* is a finite set of **datalog**$^\neg$ rules.

We introduce some useful notions next. The *head* of r, denoted by $head(r)$, is the expression $Q_0(\bar{x}_0)$; and the *body* of r, denoted by $body(r)$, is the expressions $Q_1(\bar{x}_1)$, ..., $Q_m(\bar{x}_m)$. Let P be a **datalog**$^\neg$ program. We make a distinction between so-called extensional and intensional relation names. An *extensional* relation name of P is a relation name occurring only in the body of rules of P. An *intensional* relation name is a relation name occurring in the head of some rule in P. The *extensional vocabulary* of P, denoted by $edb(P)$, is the set of all extensional relation names of P. The *intensional vocabulary* of P, denoted by $idb(P)$, is the set of all intensional relation names of P. The semantics of a **datalog**$^\neg$ program is a mapping from the set of all $edb(P)$-structures to the set of all $idb(P)$-structures as defined next.

A *domain instantiation* of a rule $r = Q_0(\bar{x}_0) \leftarrow Q_1(\bar{x}_1), \ldots, Q_m(\bar{x}_m)$ with respect to an $edb(P)$-structure $\mathcal{A}$, is a rule $Q_0(\rho(\bar{x}_0)) \leftarrow Q_1(\rho(\bar{x}_1)), \ldots, Q_m(\rho(\bar{x}_m))$ where ρ is a valuation that maps each variable in r to an element of $\mathcal{A}$. We denote this instantiated rule by $\rho(r)$ and call each instantiated literal $Q_i(\rho(\bar{x}_i))$ a *ground literal.*

The *reduced ground version of P with respect to A*, denoted by $redground(P, \mathcal{A})$, is the set of rules obtained from the set

$$\{r' \mid r' \text{ is a domain instantiation w.r.t. } \mathcal{A} \text{ of some } r \in P\},$$

by removing all the rules r' satisfying one of the following criteria:

- there is a ground literal $R(\bar{a})$ in $body(r)$ with $R \in edb(P)$ but $\bar{a} \notin R^{\mathcal{A}}$;[6]
- there is a ground literal $\neg R(\bar{a})$ in $body(r)$ with $R \in edb(P)$ but $\bar{a} \in R^{\mathcal{A}}$;
- there is a ground literal $a = b$ in $body(r)$ but $a \neq b$; or
- there is a ground literal $\neg(a = b)$ in $body(r)$ but $a = b$.

We keep all the remaining rules but from them we remove all ground literals of the form $a = a$, $\neg(a = b)$, and $R(\bar{a})$ and $\neg R(\bar{a})$ where R is an extensional relation name. Note that $redground(P, \mathcal{A})$ only contains literals referring to $idb(P)$-predicates. The above treatment is not the standard one. However, it allows us to forget about the $edb(P)$-structure when defining the semantics of P.

In the definition of the semantics of a **datalog**$^\neg$ programs we use the following shorthand. We say that a ground literal L belongs to $\mathcal{A}$, denoted $L \in \mathcal{A}$, when L is of the form $R(\bar{a})$ and $\bar{a} \in R^{\mathcal{A}}$ or when L is of the form $\neg R(\bar{a})$ and $\bar{a} \notin R^{\mathcal{A}}$

Each program P and $edb(P)$-structure $\mathcal{A}$ determine an immediate consequence operator, denoted by $T_P^{\mathcal{A}}$, mapping each $idb(P)$-structure $\mathcal{B}$ whose domain equals the domain of $\mathcal{A}$, to an $idb(P)$-structure $T_P^{\mathcal{A}}(\mathcal{B})$ whose domain equals the domain of $\mathcal{A}$. Specifically, the $idb(P)$-structure $T_P^{\mathcal{A}}(\mathcal{B})$ is defined as follows: for each $R \in idb(P)$ and each tuple $\bar{a}$ of $\mathcal{A}$, $R(\bar{a}) \in T_P^{\mathcal{A}}(\mathcal{B})$ if there exists a rule $R(\bar{a}) \leftarrow L_1, \ldots, L_n$ in

[6]Recall from Section 2.1, that $R^{\mathcal{A}}$ denotes the interpretation in $\mathcal{A}$ of the relation name R.

redground$(P, \mathcal{A})$ such that for $i = 1, \ldots, n$, $L_i \in \mathcal{B}$. This operator forms the basis for the partial fixpoint semantics defined next.

The partial fixpoint procedure (the standard one for deductive programs without negation) is usually not considered in the presence of negation, as it is not even guaranteed to terminate; nevertheless, we will show soon that it always does on the programs that are generated by a certain translation from RAGs. Concretely, the stages induced by a **datalog**$^\neg$ program P on an instance $\mathcal{A}$ are inductively defined as follows: $T_P^{\text{pfp},(0)}(\mathcal{A})$ is the *idb*(P)-structure where all relations are interpreted by the empty set; and, for $i > 0$,

$$T_P^{\text{pfp},(i)}(\mathcal{A}) \quad := \quad T_P^{\mathcal{A}}(T_P^{\text{pfp},(i-1)}(\mathcal{A})).$$

If there exists an n such that $T_P^{\text{pfp},(n)}(\mathcal{A}) = T_P^{\text{pfp},(n+1)}(\mathcal{A})$, then we say that the *partial fixpoint* of P on A exists and define $T_P^{\text{pfp}}(\mathcal{A}) := T_P^{\text{pfp},(n)}(\mathcal{A})$. Otherwise $T_P^{\text{pfp}}(\mathcal{A})$ is undefined. If $R(\bar{a}) \in T_P^{\text{pfp},(i)}(\mathcal{A})$, then we say that $R(\bar{a})$ is *derived* by the i-th iteration of P on $\mathcal{A}$.

6.5.2 Acyclic datalog$^\neg$ programs

The notion of acyclic **datalog**$^\neg$ programs will play an important role in the following. To define this notion, we recall the definition of a precedence graph. The *precedence graph* $G_P^{\mathcal{A}}$ of a **datalog**$^\neg$ program P w.r.t. an *edb*(P)-structure $\mathcal{A}$ is a directed graph whose nodes are the positive ground literals occurring in *redground*$(P, \mathcal{A})$. There is an edge from $R(\bar{a})$ to $S(\bar{b})$ in $G_P^{\mathcal{A}}$ iff there is a rule having $S(\bar{b})$ as head and containing $R(\bar{a})$ or $\neg R(\bar{b})$ in its body. We say that P is *acyclic w.r.t.* $\mathcal{A}$ whenever $G_P^{\mathcal{A}}$ is acyclic. We refer to the nodes without incoming edged as *sources* and to the nodes without outgoing edges as *sinks*. For an acyclic program, the height of a node L in $G_P^{\mathcal{A}}$, denoted by $height_{P,\mathcal{A}}(L)$, is defined as the number of nodes on the longest path in $G_P^{\mathcal{A}}$ from a source to L (the source and L included). In particular, sources have height 1. We say that a positive ground literal L not occurring in $G_P^{\mathcal{A}}$ has height 1, as it can be seen to be defined by the rule $L \leftarrow$ *false*.

We point out that our notion of acyclic programs is equivalent to the same notion defined by Apt and Bezem [AB91] in terms of level mappings. That is, a **datalog**$^\neg$ program P is acyclic when for every instance $\mathcal{A}$ there is a function f, called a level mapping, from the ground literals in *redground*$(P, \mathcal{A})$ to the natural numbers such that

1. $f(L) = f(\neg L)$ for each positive literal L in *redground*$(P, \mathcal{A})$; and
2. for every rule $L \leftarrow L_1, \ldots, L_n$ in *redground*$(P, \mathcal{A})$, we have $f(L) > f(L_i)$ for $i = 1, \ldots, n$.

Our definition of acyclicity is just more convenient for our purpose as it much more resembles the corresponding notion of non-circularity of attribute grammars.

We stress that the partial fixpoint procedure on acyclic programs differs from the evaluation described by Apt and Bezem [AB91]. Indeed, they define the evaluation of the rules levelwise. That is, in the i-th application of the immediate consequence operator only the rules with heads of level i w.r.t. f can be applied. Consequently, a literal that is derived cannot be retracted again. The partial fixpoint procedure we consider is much more naive and can at any time derive literals of any height. Surprisingly, we will show that this naive approach leads to the desired result. Moreover, as opposed to the evaluation proposed by Apt and Bezem, this naive evaluation is trivially implementable. In particular, it does not require the computation of the actual level mapping.

6.5.3 Properties of T_P^{pfp} for acyclic programs

The following lemma says that the partial fixpoint of a program P on an instance $\mathcal{A}$ over $edb(P)$ always exists when the reduced ground version of P with respect to $\mathcal{A}$ is acyclic. Specifically, it states that the correct value of any literal of height i is reached after i iterations. In particular, this means that the partial fixpoint semantics coincides with the levelwise semantics of Apt and Bezem [AB91].

Lemma 6.17 *Let P be a* **datalog**$^\neg$ *program and let $\mathcal{A}$ be an instance. If $G_P^{\mathcal{A}}$ is acyclic, then for every $i \geq 0$ and every positive literal L with $height_{P,\mathcal{A}}(L) \leq i$,*

- *if $L \in T_P^{\text{pfp},(i)}(\mathcal{A})$ then $\forall j \geq i$: $L \in T_P^{\text{pfp},(j)}(\mathcal{A})$; and*
- *if $L \notin T_P^{\text{pfp},(i)}(\mathcal{A})$ then $\forall j \geq i$: $L \notin T_P^{\text{pfp},(j)}(\mathcal{A})$.*

Proof. The proof proceeds by induction on i. The case $i = 0$ is trivial as there are no literals of height 0. Therefore, let $i > 0$ and let L be a positive literal of height i. If $L \in T_P^{\text{pfp},(i)}$ then there exists a rule

$$L \leftarrow L_1, \ldots, L_n, \neg K_1, \ldots, \neg K_m$$

in $redground(P, \mathcal{A})$ such that $L_\ell \in T_P^{\text{pfp},(i-1)}$ and $K_k \notin T_P^{\text{pfp},(i-1)}$, for $\ell = 1, \ldots, n$ and $k = 1, \ldots, m$. As the height of L is smaller or equal to i,

$$height_{P,\mathcal{A}}(L_\ell) \leq i - 1$$

and

$$height_{P,\mathcal{A}}(K_k) \leq i - 1,$$

for all $\ell = 1, \ldots, n$ and $k = 1, \ldots, m$, simply because every path from a source to an L_ℓ or a K_k can be extended to a strictly larger path from this source to L. Hence, we have, by the inductive hypothesis, that

$$L_\ell \in T_P^{\text{pfp},(j)}$$

and

$$K_k \notin T_P^{\text{pfp},(j)},$$

for all $j \geq i-1$. In particular this means that $L \in T_P^{\mathrm{pfp},(j)}$, for all $j \geq i$.

Now suppose $L \notin T_P^{\mathrm{pfp},(i)}$. Then, for every rule

$$r = L \leftarrow L_1, \ldots, L_n, \neg K_1, \ldots, \neg K_m$$

in *redground*$(P, \mathcal{A})$ at least, one of the following holds:

1. there exists an $\ell_r \in \{1, \ldots, n\}$ such that $L_{\ell_r} \notin T_P^{\mathrm{pfp},(i-1)}$; or

2. there exists a $k_r \in \{1, \ldots, m\}$ such that $K_{k_r} \in T_P^{\mathrm{pfp},(i-1)}$.

Since, for each such ℓ_r and k_r, $\mathit{height}_{P,\mathcal{A}}(L_{\ell_r}) \leq i-1$ and $\mathit{height}_{P,\mathcal{A}}(K_{k_r}) \leq i-1$, we have, by the inductive hypothesis, that $L_{\ell_r} \notin T_P^{\mathrm{pfp},(j)}$ and $K_{k_r} \in T_P^{\mathrm{pfp},(j)}$, for all $j \geq i-1$. Hence, the rule r can never be applied and we can conclude that $L \notin T_P^{\mathrm{pfp},(j)}$, for all $j \geq i$. ■

We say that a **datalog**$^\neg$ program P is *modularly stratified w.r.t. an instance* $\mathcal{A}$, when *redground*$(P, \mathcal{A})$ is locally stratified, that is, there is an assignment of natural numbers to ground literals occurring in *redground*$(P, \mathcal{A})$ such that whenever a ground literal appears negatively in the body of a rule of *redground*(P, E), the head of that rule is of a strictly higher level, and whenever a ground literal appears positively in the body of an instantiated rule, the ground literal in the head has at least that level. Ross defined modularly stratified program more generally than we do here [Ros94]. Among many things, Ross proved that every modularly stratified program has a total well-founded model [VRS91] that coincides with the one given by the stratified semantics.

Clearly, *redground*$(P, \mathcal{A})$ is locally stratified whenever it is acyclic. Therefore, we can state the following corollary of Lemma 6.17. In particular, this will imply that the reduction of RAGs to be presented in the next section also works for deductive systems based on the well-founded semantics.

Corollary 6.18 *For any* **datalog**$^\neg$ *program P and instance $\mathcal{A}$, if redground$(P, \mathcal{A})$ is acyclic then its partial fixpoint coincides with the total well-founded model.*

Since the semantic rules of RAGs consist of FO formulas, we end this section by considering the translation of these into acyclic **datalog**$^\neg$ programs. However, we will use a more uniform notion of acyclicity. For a **datalog**$^\neg$ program P, define its precedence graph, denoted by G_P, as the graph whose nodes consist of the relation names in *idb*(P) and where there is an edge from R to R' if there is a rule having R' as head and containing R or its negation in its body. We say that P is *acyclic* when G_P is. We define the *height* of an acyclic program P, denoted by *height*(P), as the number of nodes on the longest path from a source to a sink (source and sink included). In particular, as source has height 1.

For an FO formula $\varphi(\bar{x})$, define the program P_φ inductively as follows:

$$
\begin{array}{rcllcl}
\varphi(x,y) & \equiv & x = y & & \Rightarrow & Q_\varphi(x,y) \leftarrow x = y. \\
\varphi(\bar{x}) & \equiv & R(\bar{x}), & R \in \tau & \Rightarrow & Q_\varphi(\bar{x}) \leftarrow R(\bar{x}). \\
\varphi(\bar{x}) & \equiv & \varphi_1(\bar{x}) \vee \varphi_2(\bar{x}) & & \Rightarrow & Q_\varphi(\bar{x}) \leftarrow Q_{\varphi_1}(\bar{x}); \\
 & & & & & Q_\varphi(\bar{x}) \leftarrow Q_{\varphi_2}(\bar{x}); \\
 & & & & & P_{\varphi_1}; \\
 & & & & & P_{\varphi_2}. \\
\varphi(\bar{x}) & \equiv & \neg\psi(\bar{x}) & & \Rightarrow & Q_\varphi(\bar{x}) \leftarrow \neg Q_\psi(\bar{x}); \\
 & & & & & P_\psi. \\
\varphi(\bar{x}) & \equiv & (\exists y)\psi(\bar{x},y) & & \Rightarrow & Q_\varphi(\bar{x}) \leftarrow Q_\psi(\bar{x},y); \\
 & & & & & P_\psi.
\end{array}
$$

In the case of a disjunction we can assume w.l.o.g. that $\varphi_1(\bar{x})$ and $\varphi_2(\bar{x})$ have the same free variables. Clearly, each P_φ is acyclic.

Proposition 6.19 *For every* FO *formula φ and structure $\mathcal{A}$:*

1. *$T^{\mathrm{pfp}}_{P_\varphi}(\mathcal{A})$ exists and $T^{\mathrm{pfp}}_{P_\varphi}(\mathcal{A}) = T^{\mathrm{pfp},(n)}_{P_\varphi}(\mathcal{A})$ for all $n \geq$ height(P_φ); and*

2. *$\forall \bar{a} \in \mathcal{A} : \mathcal{A} \models \varphi[\bar{a}] \Leftrightarrow Q_\varphi(\bar{a}) \in T^{\mathrm{pfp}}_{P_\varphi}(\mathcal{A})$.*

Proof. The proof proceeds by induction on the structure of FO formulas. The proposition clearly holds in case φ is an atomic formula. Let φ be of the form $\varphi_1(\bar{x}) \vee \varphi_2(\bar{x})$. Then, clearly, $T^{\mathrm{pfp}}_{P_\varphi}(\mathcal{A}) = T^{\mathrm{pfp}}_{P_{\varphi_1}}(\mathcal{A}) \cup T^{\mathrm{pfp}}_{P_{\varphi_2}}(\mathcal{A})$. Consequently, (1) and (2) hold by induction. Let φ be of the form $\neg\psi$. Then, by induction, for every $n \geq \mathit{height}(P_\psi)$ and every $\bar{a} \in \mathcal{A}$, we have $\mathcal{A} \models \varphi[\bar{a}]$ iff $Q_\psi(\bar{a}) \notin T^{\mathrm{pfp},(n)}_{P_\varphi}(\mathcal{A})$. Hence, by definition, for every $n \geq \mathit{height}(P_\psi) + 1 = \mathit{height}(P_\varphi)$ and every $\bar{a} \in \mathcal{A}$ we have, $\mathcal{A} \models \varphi[\bar{a}]$ iff $Q_\varphi(\bar{a}) \in T^{\mathrm{pfp},(n+1)}_{P_\varphi}(\mathcal{A})$. Since $T^{\mathrm{pfp},(\mathit{height}(P_\varphi))}_{P_\varphi}(\mathcal{A}) = T^{\mathrm{pfp},(\mathit{height}(P_\varphi)+1)}_{P_\varphi}(\mathcal{A})$, (1) and (2) hold. The case where φ is of the form $(\exists x)\psi$ is similar. ■

6.5.4 Translation of RAGs to datalog$^\neg$

We discuss the translation of a RAG $\mathcal{R}$ to a **datalog**$^\neg$ program (cf. Section 3.1.3 for the definition of a RAG). Let G be the context-free grammar over which the RAGs are defined and let r be the maximum length of the right-hand sides of productions in G. Consequently, all derivation trees will be of arity at most r. To facilitate our translation, we will replace in the vocabulary of trees, the ordering $<$ on the children of each node (cf. Section 2.4), by r successor relations $S_1, \ldots, S_r$. These have the following meaning: for each $i = 1, \ldots, n$, $S_i(\mathbf{n}, \mathbf{m})$ expresses that $\mathbf{m}$ is the i-th child of $\mathbf{n}$. So, the extensional vocabulary will contain the binary relations $S_1, \ldots, S_n$ and a unary relation O_X for each grammar symbol X of G.

For each RAG $\mathcal{R}$ we construct a **datalog**$^\neg$ program $P(\mathcal{R})$ containing a $(k+1)$-ary relation name a for each k-ary attribute a of $\mathcal{R}$ such that for a tree $\mathbf{t}$, a node $\mathbf{n}$, and a tuple of nodes $\bar{\mathbf{m}}$, $a(\mathbf{n}, \bar{\mathbf{m}}) \in T^{\mathrm{pfp}}_{P(\mathcal{R})}(\mathbf{t})$ if and only if $\bar{\mathbf{m}} \in \mathcal{R}(\mathbf{t})(a(\mathbf{n}))$.

More concretely, the **datalog**$^\neg$ program $P(\mathcal{R})$ is the set of **datalog**$^\neg$ rules obtained as follows. We assume, w.l.o.g., that no variable $z_0, z_1, z_2, \ldots$ occurs in a semantic rule of $\mathcal{R}$. For each rule $a(i)(\bar{x}) := \varphi(\bar{x})$ in the context (p, a, i), with $p = X_0 \to X_1 \ldots X_n$, we define the program $P_{p,a,i}$ as the rule

$$a(z_i, \bar{x}) \leftarrow S_1(z_0, z_1), \ldots, S_n(z_0, z_n), O_{X_0}(z_0), \ldots, O_{X_n}(z_n), Q_{\hat{\varphi}}(z_0, \ldots, z_n, \bar{x})$$

together with the program

$$P_{\hat{\varphi}},$$

where, $\hat{\varphi}$ is the FO formula obtained from φ by replacing each occurrence of $\mathbf{j}$ by z_j and each occurrence of $b(j)(\bar{y})$ by $b(z_j, \bar{y})$, for each $j = 0, \ldots, n$ and each attribute b. We refer to the relation names a coming from attributes as attribute names.

The program $P(\mathcal{R})$ is defined as the union of the programs $P_{p,a,i}$ for each context (p, a, i). We illustrate the above described translation with an example.

Example 6.20 Consider the following RAG $\mathcal{R}$ which computes the set of S-labeled nodes occurring on an odd position when counting upwards:

$$\begin{array}{ll} U \to S & \mathit{odd}(0)(x) := \mathit{odd}(1)(x); \\ S \to S & \mathit{odd}(0)(x) := \mathit{odd}(1)(x) \vee (x = \mathbf{0} \wedge \neg \mathit{odd}(1)(\mathbf{1})); \\ S \to s & \mathit{odd}(0)(x) := x = \mathbf{0}. \end{array}$$

Then $P(\mathcal{R})$ is defined as the program consisting of the following rules:

$$\begin{array}{lcl} \mathit{odd}(z_0, x) & \leftarrow & S_1(z_0, z_1), O_U(z_0), O_S(z_1), Q_1(z_0, z_1, x); \\ Q_1(z_0, z_1, x) & \leftarrow & \mathit{odd}(z_1, x); \\ \mathit{odd}(z_0, x) & \leftarrow & S_1(z_0, z_1), O_S(z_0), O_S(z_1), Q_2(z_0, z_1, x); \\ Q_2(z_0, z_1, x) & \leftarrow & \mathit{odd}(z_1, x); \\ Q_2(z_0, z_1, x) & \leftarrow & x = z_0, \neg \mathit{odd}(z_1, z_1); \\ \mathit{odd}(z_0, x) & \leftarrow & S_1(z_0, z_1), O_S(z_0), O_s(z_1), Q_3(z_0, z_1, x); \\ Q_3(z_0, z_1, x) & \leftarrow & x = z_0. \end{array}$$

Clearly, *redground*$(P(\mathcal{R}), \mathbf{t})$ is acyclic for every derivation tree $\mathbf{t}$. ■

Below, we will show that for each attribute-node pair $a(\mathbf{n})$ and each tuple of nodes $\bar{\mathbf{n}}'$ of a tree $\mathbf{t}$,

$$\bar{\mathbf{n}}' \in \mathcal{R}(\mathbf{t})(a(\mathbf{n})) \quad \Leftrightarrow \quad a(\mathbf{n}, \bar{\mathbf{n}}') \in T^{\mathrm{pfp}}_{P(\mathcal{R})}(\mathbf{t}).$$

First, we state some easy but helpful lemma's and introduce some more notation.

The *dependency graph* $D_{\mathcal{R}}(\mathbf{t})$ of a RAG $\mathcal{R}$ with respect to $\mathbf{t}$ consists of all attribute node pairs $a(\mathbf{n})$ such that a is an attribute of the label of $\mathbf{n}$. Further, there is an edge from $a(\mathbf{n})$ to $b(\mathbf{m})$ iff $a(\mathbf{n})$ occurs in $\Delta(\mathcal{R}, \mathbf{t}, b, \mathbf{m})$ (cf. Definition 3.16). It is well known that $\mathcal{R}$ is non-circular iff $D_{\mathcal{R}}(\mathbf{t})$ is acyclic for every derivation tree [DJL88]. The next lemma can be shown by induction on i.

Lemma 6.21 *Let* $\ell = \max\{\mathit{height}(P_{\hat{\varphi}}) \mid \varphi$ *a semantic rule of* $\mathcal{R}\} + 1$ *and* $\mathbf{t}$ *be a tree. If* $a(\mathbf{n})$ *is defined in* $\mathcal{R}_i(\mathbf{t})$*, then* $\mathit{height}_{P(\mathcal{R}),\mathbf{t}}(a(\mathbf{n}, \bar{\mathbf{m}})) \leq i \cdot \ell$ *for all* $\bar{\mathbf{m}}$.

By the following lemma and Lemma 6.17, $T^{\text{pfp}}_{P(\mathcal{R})}(\mathbf{t})$ exists for each RAG $\mathcal{R}$ and each derivation tree $\mathbf{t}$. Recall from Chapter 3 that we only consider non-circular RAGs. In the remaining we show that this program indeed captures the semantics of $\mathcal{R}$.

Lemma 6.22 If $\mathcal{R}$ is a RAG, then *redground*$(P(\mathcal{R}),\mathbf{t})$ is acyclic for every derivation tree $\mathbf{t}$.

Proof. Note that there can be no cycle in a $G^{\mathbf{t}}_{P_{\hat\varphi}}$, since each $P_{\hat\varphi}$ is acyclic. Hence, a cycle in *redground*$(P(\mathcal{R}),\mathbf{t})$ should at least involve two attribute names. By construction, for all nodes $\mathbf{n}$ and $\mathbf{m}$, sequences of nodes $\bar{\mathbf{n}}'$ and $\bar{\mathbf{m}}'$, and attributes a and b, there is a path from $a(\mathbf{n},\bar{\mathbf{n}}')$ to $b(\mathbf{m},\bar{\mathbf{m}}')$ in $G^{\mathbf{t}}_{P(\mathcal{R})}$ that passes no other attribute name if and only if there is an edge from $a(\mathbf{n})$ to $b(\mathbf{m})$ in $D_{\mathcal{R}}(\mathbf{t})$. Hence, $G^{\mathbf{t}}_{P(\mathcal{R})}$ is acyclic, since $D_{\mathcal{R}}(\mathbf{t})$ is acyclic. ∎

We introduce some notation to state the following lemma which is the final step towards Theorem 6.24. Specifically, it says that the evaluation of the semantic rule φ in context (p,a,i) is captured by the program $P_{p,a,i}$. Let $\mathbf{t}$ be a derivation tree, let v be a valuation for the attribute-node pairs in $\Delta(\mathcal{R},\mathbf{t},a,\mathbf{n})$, and let φ be the formula defining the attribute a for the node $\mathbf{n}$ in context (p,a,i) with $p = X_0 \to X_1 \dots X_n$. Define $\mathcal{A}^v$ as the structure where for each b, $\mathbf{m}$, and $\bar{\mathbf{m}}'$, $(\mathbf{m},\bar{\mathbf{m}}') \in b^{\mathcal{A}^v}$ iff $\bar{\mathbf{m}}' \in v(b(\mathbf{m}))$. By Proposition 6.19 and the construction of P, the next lemma is immediate.

Lemma 6.23 *For any context (p,a,i), $n \geq \mathit{height}(P_{p,a,i})$, and any sequence of nodes $\bar{\mathbf{n}}'$ of a derivation tree $\mathbf{t}$, we have that $\bar{\mathbf{n}}'$ belongs to the relation defined by the formula $\Delta(\mathcal{R},\mathbf{t},a,\mathbf{n})$ where each $b(\mathbf{m})$ is interpreted by the relation $v(b(\mathbf{m}))$ iff $a(\mathbf{n},\bar{\mathbf{n}}') \in T^{\text{pfp},(n)}_{P_{p,a,i}}(\mathcal{A}^v \cup \mathbf{t})$.*[7]

We now prove that the program $P(\mathcal{R})$ captures the semantics of $\mathcal{R}$.

Theorem 6.24 *Let $\ell = \max\{\mathit{height}(P_{\hat\varphi}) \mid \varphi$ a semantic rule of $\mathcal{R}\}+1$. For all $i \geq 0$, if $a(\mathbf{n})$ is defined in $\mathcal{R}_i(\mathbf{t})$ then for all sequences of nodes $\bar{\mathbf{n}}'$ of $\mathbf{t}$,*

$$\bar{\mathbf{n}}' \in \mathcal{R}_i(\mathbf{t})(a(\mathbf{n})) \quad \Leftrightarrow \quad a(\mathbf{n},\bar{\mathbf{n}}') \in T^{\text{pfp},(i\cdot\ell)}_{P(\mathcal{R})}(\mathbf{t}).$$

Proof. The proof proceeds by induction on i. The case $i=0$ clearly holds. Suppose $i>0$ and $a(\mathbf{n})$ is defined in $\mathcal{R}_i(a(\mathbf{n}))$. By the inductive hypothesis, for every relation symbol $b(\mathbf{m})$ in $\Delta(\mathcal{R},\mathbf{t},a,\mathbf{n})$ we have that for every $\bar{\mathbf{m}}'$

$$\bar{\mathbf{m}}' \in \mathcal{R}_{i-1}(\mathbf{t})(b(\mathbf{m})) \quad \Leftrightarrow \quad b(\mathbf{m},\bar{\mathbf{m}}') \in T^{\text{pfp},((i-1)\cdot\ell)}_{P(\mathcal{R})}(\mathbf{t}).$$

By Lemma 6.21, the height of $b(\mathbf{m},\bar{\mathbf{m}}')$ is less or equal to $(i-1)\cdot\ell$. Hence, by Lemma 6.17, for all $j \geq (i-1)\cdot\ell$, $b(\mathbf{m},\bar{\mathbf{m}}') \in T^{\text{pfp},(j)}_{P(\mathcal{R})}(\mathbf{t})$ iff $\bar{\mathbf{m}}' \in \mathcal{R}_{i-1}(\mathbf{t})(b(\mathbf{m}))$.

[7] Here, $\mathcal{A}^v \cup \mathbf{t}$ denotes the structure consisting of the relations in $\mathcal{A}^v$ and in $\mathbf{t}$.

Thus, by Lemma 6.23 it follows that for every $\bar{\mathbf{n}}'$

$$\bar{\mathbf{n}}' \in \mathcal{R}_i(\mathbf{t})(a(\mathbf{n})) \quad \Leftrightarrow \quad a(\mathbf{n}', \bar{\mathbf{n}}) \in T_{P(\mathcal{R})}^{\mathrm{pfp},(i \cdot \ell)}(\mathbf{t}).$$

■

7

Discussion

7.1 Main results

In this work we studied various languages to query structured documents.

First we addressed attribute grammars as a query language for structured documents modeled as derivation trees of context-free grammars. In this respect, BAGs as a language for expressing simple retrieval queries strike a reasonable balance between expressive power and complexity; on the one hand, they are as powerful as monadic second-order logic; on the other hand, they can be evaluated in linear time. Further, RAGs as a language for expressing general relational queries on structured documents offer more expressive power than BAGs, while remaining within polynomial-time complexity. Moreover, both formalisms can be readily implemented on top of a deductive database system.

Inspired by the definition of tree automata on unranked trees by Brüggemann-Klein, Murata and Wood [BKMW98, Mur95], we introduced extended AGs as generalizations of BAGs expressing selection queries on documents modeled by extended context-free grammars. This formalism captures the selection queries definable MSO. We also established the complexity of the non-emptiness and the equivalence problem, relevant for optimization purposes, to be complete for EXPTIME. On the negative side, extended AGs can only express queries that retrieve nodes from a document. It would be interesting to see an extension of the present formalism for actual restructuring of documents. A related paper in this respect is that of Crescenzi and Mecca [CM98b]. They define an interesting formalism for the definition of wrappers that map derivation trees of regular grammars to relational databases. Their formalism, however, is only defined for regular grammars and the correspondence between actions (i.e., semantic rules) and grammar symbols occurring in regular expressions is not so flexible as for extended AGs.

Hereafter, we studied the expressiveness of query automata computing selection queries on both ranked and unranked trees. In both cases they capture MSO. However, to achieve this expressivity, we had to add special stay transitions to the computation model in the unranked case. Interestingly, these strong query automata and the ordinary query automata do accept the same class of unranked tree languages. This indicates a substantial difference between looking at automata from a formal language point of view (i.e., for defining tree languages) and looking at automata from a database point of view (i.e., for expressing queries). Further, we established the complexity of the non-emptiness and the equivalence problem to be complete for EXPTIME.

We stress that even though extended AGs and query automata are equally expressive, they are very different in nature. Indeed, query automata constitute a procedural formalism that has only local memory (in the state of the automaton), but which can visit each node more than a constant number of times. Attribute grammars, on the other hand, are a declarative formalism, whose evaluation visits each node of the input tree only a constant number of times (once for each attribute). In addition, they have a distributed memory (in the attributes at each node). It is precisely this distributed memory which makes extended AGs particularly well-suited for an efficient simulation of Region Algebra expressions. It is, therefore, not clear whether an *efficient* translation from Region Algebra expressions into query automata exists.

Further, we discussed some meaningful applications of our results and techniques. In brief,

1. we improved the upper bound on the complexity of the equivalence test of Region Algebra expressions from iterated exponential to EXPTIME;

2. we showed that already a very restricted subset of the actual XML transformation language XSLT has the ability to issue any MSO pattern at any node in the document. Hereby, on the one hand, we reveal that core XSLT has a very powerful pattern language at its disposal, and, on the other hand, provide evidence for the robustness of the language; and

3. we proved MSO, and therefore query automata and extended AGs, to be more expressive than the selective power of most current languages for semi-structured data and XML.

A further contribution of this work is that all proposed query languages can take the inherent order of the children of a node into account. As argued by Suciu [Suc98], this is a major research issue in the design of query languages for semi-structured data and XML.

The majority of the formalism studied in this work are especially tailored for expressing selection queries. These are particularly relevant since they (i) constitute the most simple and common form of document querying; and (ii) form the basis of more general transformation languages: such language have to identify (that is, select) the relevant parts of the input document that have to be combined (possible after some additional transformation) to comprise the output document.

Two related papers in the context of selection queries are those of Murata [Mur98] and Neumann and Seidl [NS98] who used pointed tree representations and a μ-calculus, respectively, to express unary queries over unranked trees. These can readily be expressed in MSO, and thus also by QAs and extended AGs.

7.2 Epilogue

As mentioned before, the result linking BAGs with MSO is shown independently by Bloem and Engelfriet [BE].[1] Especially since they did not intend to model a query language, it is striking that they considered exactly the same problem. Moreover, in my view, the research for the foundations of query languages and data models for structured documents, semi-structured data, or XML, could benefit from more interaction between the database and the formal language theory community, for the simple reason that the latter community has been studying computations on trees (and graphs) for the last three decades.

This interest is bilateral:

(*i*) the formal language theory community has developed paradigms and formalisms that might be applied in or converted to concrete query languages as was the approach adopted in this dissertation; further, they might provide us with suitable techniques for studying already existing transformation languages;

(*ii*) this renewed interest in formal language theory motivated by applications in database theory, might generate new relevant questions not addressed by the formal language theory community.

[1] To be precise, Bloem and Engelfriet studied *finite*-valued attribute grammars, but these are essentially equivalent to BAGs.

Bibliography

[AB91] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3/4):335–364, 1991.

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[ACM98] S. Abiteboul, S. Cluet, and T. Milo. A logical view of structured files. *VLDB Journal*, 7(2):96–114, 1998.

[AHU69] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. A general theory of translation. *Mathematical Systems Theory*, 3:193–221, 1969.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AV95] S. Abiteboul and V. Vianu. Computing with first-order logic. *Journal of Computer and System Sciences*, 50(2):309–335, 1995.

[BDHD96] P. Buneman, S. Davidson, G. G. Hillebrand, and D.Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25:2 of *SIGMOD Record*, pages 505–516. ACM Press, 1996.

[BE] R. Bloem and J. Engelfriet. A comparison of tree transductions defined by monadic second order logic and by attribute grammars. Technical Report 98-02, Rijksuniversiteit Leiden, January 1998. Revised version: `http://www.wi.leidenuniv.nl/home/engelfri`.

[BE97] R. Bloem and J. Engelfriet. Monadic second order logic and node relations on graphs and trees. In Mycielski et al. [MRS97], pages 144–161.

[BEGO71] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, c-20(2):149–153, 1971.

[BH67] M. Blum and C. Hewitt. Automata on a 2-dimensional tape. In *Conference Record of 1967 Eighth Annual Symposium on Switching and Automata Theory*, pages 155–160. IEEE, 1967.

[BKD98] A. Brüggemann-Klein and Wood D. One unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[BKMW98] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets (draft 1). Unpublished manuscript, 1998.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[BYN96] R. Beaza-Yates and G. Navarro. Integrating contents and structure in text retrieval. *ACM SIGMOD Record*, 25(1):67–79, 1996.

[CD] J. Clark and S. Deach. Extensible stylesheet language (XSL). http://www.w3.org/TR/WD-xsl.

[CD88] B. Courcelle and P. Deransart. Proofs of partial correctness for attribute grammars with applications to recursive procedures and logic programming. *Information and Computation*, 78(1):1–55, 1988.

[Chl86] B. S. Chlebus. Domino-tiling games. *Journal of Computer and System Sciences*, 32(3):374–392, 1986.

[Cla99] James Clark. XSL Transformations version 1.0. http://www.w3.org/TR/WD-xslt, august 1999.

[CM94] M. Consens and T. Milo. Optimizing queries on files. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23:2 of *SIGMOD Record*, pages 301–312. ACM Press, 1994.

[CM98a] M. Consens and T. Milo. Algebras for querying text regions: Expressive power and optimization. *Journal of Computer and System Sciences*, 3:272–288, 1998.

[CM98b] V. Crescenzi and G. Mecca. Grammars have exceptions. *Information Systems – Special Issue on Semistructured Data*, 23(8):539–565, 1998.

[Con] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5. Elsevier, 1990.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: a query language for XML. In *Proceedings of the WWW8 Conference*, Toronto, 1999.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars: Definition, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer, 1988.

[Don70] J. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451, 1970.

[EF95] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer, 1995.

[EH99] J. Engelfriet and H. J. Hoogeboom. Two-way finite state transducers and monadic second-order logic. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Automata, Languages and Programming, 26th International Colloquium, ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 311–320. Springer, 1999.

[Eng79] J. Engelfriet. Two-way automata and checking automata. *Mathematical Centre Tracts*, pages 1–69, 1979.

[Fag74] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R.M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 43–73. 1974.

[FBY92] W. B. Frakes and R. Baeza-Yates, editors. *Information retrieval.* Prentice Hall, 1992.

[FFK+98] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a web-site management system. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 414–425. ACM Press, 1998.

[GH96] N. Globerman and D. Harel. Complexity results for two-way and multi-pebble automata and their logics. *Theoretical Computer Science*, 169(2):161–184, 1996.

[Gie88] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25(4):355–423, 1988.

[GS97] F. Gécseg and M. Steinby. Tree languages. In Rozenberg and Salomaa [RS97], chapter 1.

[GT87] G.H. Gonnet and F.W. Tompa. Mind your grammar: a new approach to modelling text. In *Proceedings 13th Conference on VLDB*, pages 339–346, 1987.

[HU67] J. E. Hopcroft and J. D. Ullman. An approach to a unified theory of automata. *The Bell Systems Technical Journal*, 46:1793–1829, 1967.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[IBM99] IBM. LotusXSL.
http://www.alphaworks.ibm.com/aw.nsf/xmltechnology/LotusXSL, 1999.

[Imm89] N. Immerman. Expressibility and parallel complexity. *SIAM Journal on Computing*, 18:625–638, 1989.

[Imm98] N. Immerman. *Descriptive Complexity.* Springer, 1998.

[JOR75] M. Jazayeri, W. F. Ogden, and W. C. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Communications of the ACM*, 18(12):697–706, 1975.

[KLMN90] P. Kilpeläinen, G. Lindén, H. Mannila, and E. Nikunen. A structured text database system. In R. Furuta, editor, *Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography*, The Cambridge Series on Electronic Publishing, pages 139–151. Cambridge University Press, 1990.

[KM93] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the Sixteenth International Conference on Research and Development in Information Retrieval*, pages 214–222. ACM Press, 1993.

[KM94] P. Kilpeläinen and H. Mannila. Query primitives for tree-structured data. In M. Crochemore and D. Gusfield, editors, *Proceedings of the fifth Symposium on Combinatorial Pattern Matching*, pages 213–225. Springer-Verlag, 1994.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. See also *Mathematical Systems Theory*, 5(2):95–96, 1971.

[Knu82] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1982.

[Kol90] Ph. Kolaitis. Implicit definability on finite structures and unambiguous computations. In *Proceedings 5th IEEE Symposium on Logic in Computer Science*, pages 168–180. IEEE Computer Society Press, 1990.

[KV92] P.G. Kolaitis and M.Y. Vardi. Infinitary logics and 0-1 laws. *Information and Computation*, 98(2):258–294, 1992.

[Lad77] R. E. Ladner. Application of model theoretic games to discrete linear orders and finite automata. *Information and Control*, 33(4):281–303, 1977.

[MAM+98] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. The ARANEUS web-base management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 27 of *ACM SIGMOD Record*, pages 544–546. ACM Press, 1998.

[MN99] S. Maneth and F. Neven. A formalization of tree transformations in XSL. To appear in the proceedings of the Seventh International Workshop on Database Programming Languages, *Lecture Notes in Computer Science*, 1999.

[MSV99] T. Milo, D. Suciu, and V. Vianu. Typecheking for XML transformers. Unpublished manuscript, 1999.

[Mor94] E. Moriya. On two-way tree automata. *Information Processing Letters*, 50:117–121, 1994.

[Mos74] Y.N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[MRS97] J. Mycielski, G. Rozenberg, and A. Salomaa, editors. *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Mur95] M. Murata. Forest-regular languages and tree-regular languages. Unpublished manuscript, 1995.

[Mur98] M. Murata. Data model for document transformation and assembly. In *Proceedings of the workshop on Principles of Digital Document Processing*, 1998. To appear in LNCS.

[Nev98] F. Neven. Structured document query languages based on attribute grammars: locality and non-determinism. In T. Ripke T. Polle and K.-D. Schewe, editors, *Fundamentals of Information Systems*, pages 129–142. Kluwer, 1998.

[Nev99] F. Neven. Extensions of attribute grammars for structured document queries. To appear in the proceedings of the Seventh International Workshop on Database Programming Languages, *Lecture Notes in Computer Science*, 1999.

[NOTV98] F. Neven, M. Otto, J. Tyszkiewicz, and J. Van den Bussche. Adding for-loops to first-order logic. In P. Buneman C. Beeri, editor, *Database Theory – ICDT99*, volume 1540 of *Lecture Notes in Computer Science*, pages 58–69. Springer-Verlag, 1998.

[NS99] F. Neven and T. Schwentick. Query automata. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*, pages 205–214. ACM Press, 1999.

[NV97] F. Neven and J. Van den Bussche. On implementing structured document query facilities on top of a DOOD. In F. Bry, R. Ramakrishnan, and K. Ramamohanarao, editors, *Deductive and Object-Oriented Databases*, volume 1341 of *Lecture Notes in Computer Science*, pages 351–367. Springer-Verlag, 1997.

[NV98] F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In [POD98], pages 11–17.

[NVVV98] F. Neven, J. Van den Bussche, D. Van Gucht, and G. Vossen. Typed query languages for databases containing queries. To appear in Information Systems. Extended abstract appeared in [POD98], 1999.

[NS98] A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In V. Arvind and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, pages 134–145. Springer, 1998.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the 11th International Conference on Data Engineering*, pages 251–260. IEEE Computer Society Press, 1995.

[POD98] *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems.* ACM Press, 1998.

[PQ68] C. Pair and A. Quere. Définition et etude des bilangages réguliers. *Information and Control*, 13(6):565–593, 1968.

[Ros94] K. A. Ross. Modular stratification and magic sets for datalog programs with negation. *Journal of the Association for Computing Machinery*, 41(6):1216–1266, 1994.

[RS97] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 3. Springer, 1997.

[Sha92] J. Shallit. Numeration systems, linear recurrences, and regular sets (extended abstract). In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92*, volume 623 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 1992.

[She59] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, pages 198–200, 1959.

[ST92] A. Salminen and F. Tompa. PAT expressions: an algebra for text search. *Acta Linguistica Hungarica*, 41:277–306, 1992.

[Suc98] D. Suciu. Semistructured data and XML. In *Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms*, 1998.

[Tak75] M. Takahashi. Generalizations of regular sets and their application to a study of context-free languages. *Information and Control*, 27(1):1–36, 1975.

[Tho97a] W. Thomas. Ehrenfeucht games, the composition method, and the monadic theory of ordinal words. In Mycielski et al. [MRS97], pages 118–143.

[Tho97b] W. Thomas. Languages, automata, and logic. In Rozenberg and Salomaa [RS97], chapter 7.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[Val76] Leslie G. Valiant. Relative complexity of checking and evaluating. *Information Processing Lettres*, 5(1):20–23, 1976.

[Var89] M. Y. Vardi. Automata theory for database theoreticians. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 83–92. ACM Press, 1989.

[VVV96] J. Van den Bussche, D. Van Gucht, and G. Vossen. Reflective programming in the relational algebra. *Journal of Computer and System Sciences*, 52(3):537–549, 1996.

[VRS91] A Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, 1991.

[Woo95] D. Wood. Standard generalized markup language: Mathematical and philosophical issues. In J. van Leeuwen, editor, *Computer Science Today. Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 344–365. Springer-Verlag, 1995.

Samenvatting

De steeds toenemende populariteit van het Internet samen met de opkomst van markup talen zoals HTML en XML hebben bijgedragen tot de wijde verspreiding van elektronische gestructureerde documenten. De huidige databasesystemen zijn echter niet geschikt om dergelijke nieuwe vormen van data te behandelen. Om deze reden is er nood aan nieuwe databasesystemen en bijhorende query- of ondervragingstalen die op een adequate manier elektronische gestructureerde documenten kunnen opslaan en manipuleren. In dit werk concentreren we ons op het ontwikkelen en analyseren van dergelijke querytalen.

XML is de de nieuwe standaard ontwikkeld door het World Wide Web Consortium (W3C) voor het specifiëren van gestructureerde documenten. Dit formaat is in korte tijd immens populair geworden voor het uitwisselen van allerhande gegevens via het Internet. Meer nog, veel softwaregiganten gokken erop dat XML het universele data-uitwisselingsformaat zal worden en bouwen nu reeds tools voor het importeren en exporteren van XML documenten. Merkwaardig genoeg ligt de kracht en elegantie van XML in zijn eenvoud. De bouwstenen van een XML document zijn de *elementen*. Een element is tekst omsloten door begin- en eind-tags zoals bijvoorbeeld `<author>` en `</author>`. Binnenin een element kunnen zich andere elementen bevinden, pure tekst, of een combinatie van de twee. Figuur 1.1 toont een voorbeeld van een XML document dat bibliografische informatie weergeeft. In het algemeen is het echter niet zinvol om elke willekeurige combinatie van elementen toe te laten in een XML document, en willen we alleen deze die voldoen aan een bepaalde beschrijving. In ons voorbeeld zou zo een beschrijving onder ander kunnen inhouden dat elke publicatie minstens een auteur moet hebben. In XML worden dergelijke restricties aangegeven door een Document Type Definitie (DTD). Figuur 1.2 toont een DTD voor het document in Figuur 1.1. We stellen vast dat een DTD eigenlijk een soort grammatica is. Op een meer abstract niveau kunnen we XML documenten modelleren met behulp van bomen door essentieel enkel de eind-tags weg te laten. De boomvoorstelling van het document in Figuur 1.1, bijvoorbeeld, is gegeven in Figuur 1.3.

De bovenstaande vaststellingen laten ons toe gestructureerde documenten op een natuurlijke wijze te modelleren als gelabelde geordende bomen gedefinieerd door middel van een grammatica. Een database is dus één zo een boom. Informationretrievalsystemen ondervragen echter gewoonlijk een hele verzameling documenten. Maar met betrekking tot het ontwerpen van querytalen kan een verzameling van documenten worden beschouwd als één lang gestructureerd document.

Het voorgaande boommodel is essentieel hetzelfde datamodel als voor semi-gestructureerde data met dit verschil dat daar gebruik gemaakt wordt van gelabelde grafen (in plaats van bomen). Langs de andere kant wordt in dit laast genoemde model de ordening van de knopen totaal genegeerd. In ons model is deze ordening echter wel van belang, bijvoorbeeld in het document in Figuur 1.1 waar de volgorde van de auteurs wel degelijk relevant is. Daarenboven is het helemaal niet duidelijk hoe de huidige ondervragingstalen voor semi-gestructureerde data kunnen worden aangepast om met deze ordening om te gaan. Meer nog, zoals aangestipt door Suciu, is het omgaan met deze ordening een van de belangrijkste onderzoeksonderwerpen voor het semi-gestructureerd datamodel. Eén van de bijdragen van dit werk is dan ook dat *alle* onderzochte querytalen zonder problemen de ordening van knopen in rekening kunnen brengen.

Berekeningen op bomen zijn de laatste twintig jaar nauwgezet onderzocht in het gebied van formele talen. Aangezien documenten kunnen worden gemodelleerd als bomen en de queries die wij beschouwen berekeningen op bomen zijn, ligt het voor de hand dat onderzoek terug te bekijken, maar nu vanuit het oogpunt van databases. Het doel van dit werk is dan ook na te gaan hoe dergelijke formalismen kunnen worden aangewend als querytalen. In het bijzonder zullen we de expressieve kracht en optimalisatiemogelijkheden van verscheidene attribuutgrammatica's en boomautomaten onderzoeken. We passen de ontwikkelde technieken als volgt toe: (*i*) we verbeteren drastisch de complexiteit van verscheidene optimalisatieproblemen voor de Region Algebra [CM98a], en (*ii*) we tonen de robuustheid aan van de XML transformatietaal XSLT [Cla99].

In deze thesis zijn we hoofdzakelijk geïnteresseerd in querytalen voor het uitdrukken van selectievragen. Hiermee bedoelen we het opvragen van knopen in de boom die corresponderen met posities of structurele elementen van het document. Zulke vragen kunnen ook bekeken worden als vragen naar die deelbomen wier wortel voldoet aan een bepaald patroon. We refereren naar zulke vragen als unaire vragen, vermits ze eigenlijk documenten afbeelden op een verzameling van hun knopen. De interesse in unaire vragen is tweeledig:

(*i*) De selectie van deelbomen (of knopen) in grote documenten is precies de queryvorm die ondersteund wordt door de meeste informationretrievalsystemen en omvat daarom de meest eenvoudige en meest voorkomende vorm van querying.

(*ii*) Selectiequeries vormen de basis van querytalen voor meer algemene transformaties van documenten. Inderdaad, het merendeel van de document-querytalen over grafen of bomen hebben een zekere patroontaal ter hunner beschikking om de gewenste delen van het inputdocument te identificeren en daarna te combineren (eventueel na verdere manipulatie) tot het outputdocument. Deze patroontalen zijn meestal gebaseerd op reguliere padexpressies. De patroontalen die wij voorstellen zijn echter heel wat krachtiger dan deze, zoals we formeel zullen aantonen in het laatste hoofdstuk.

We geven een kort overzicht van de inhoud van de thesis.

We beginnen met de expressiviteit van standaard attribuutgrammatica's over context-vrije grammatica's te onderzoeken. Meer bepaald beschouwen we Booleaanse (BAGs) en relationele attribuutgrammatica's (RAGs). BAGs drukken unaire queries uit en vormen een abstractie van de queryfaciliteit voorzien door informationretrievalsystemen. RAGs drukken relationale queries uit en kunnen worden beschouwd als abstracties van "wrappers".

In het bijzonder linken we BAGs met monadische tweede-orde logica (MSO), en RAGs met eerste-orde-inducties van lineaire diepte, of, equivalent hiermee, de queries berekenbaar in lineaire parallelle tijd op een machine met een polynomiaal aantal processoren. Verder tonen we aan dat RAGs die alleen synthesized attributen gebruiken strict zwakker zijn dan RAGs die zowel synthesized als inherited attributen gebruiken en dat RAGs meer expressief zijn dan monadische tweede-orde-logica voor queries van willekeurige dimensie. We besluiten met een discussie van relationele attribuutgrammatica's in de context van BAGs en RAGs. In het bijzonder tonen we aan dat in het geval van BAGs deze extensie de expressive kracht niet verhoogt, terwijl de verschillende semantieken voor relationele RAGs in staat zijn om de complexiteitsklassen NP, coNP, en UP $\cap$ coUP uit te drukken.

De bekomen resultaten zijn grafisch voorgesteld in Figuur 1.4. Een pijl van een klasse van queries C naar een klasse van queries C' wil zeggen dat $C \subsetneq C'$. Een doorstreepte pijl van C naar C' wil zeggen dat er een Booleaanse query in C is die niet in C' is.

We leggen ons nu strikt toe op formalismen voor het uitdrukken van unaire queries. We definiëren een extensie van attribuutgrammatica's (extended AGs) geschikt voor het ondervragen van documenten gemodelleerd door uitgebreide context-vrije grammatica's. Zulke grammatica's zijn betere benaderingen van XML DTDs dan de hiervoor bestudeerde context-vrije grammatica's. Helaas zijn afleidingsbomen nu niet langer begrensd, in de zin dat knopen niet langer een vast maximaal aantal kinderen hebben. Dit op het eerste gezicht onschuldig verschil bemoeilijkt enorm de definitie van extended AGs. We geven een volledig overzicht van de expressieve kracht van dit formalisme en verkrijgen de exacte complexiteit van verscheidene optimalisatieproblemen. In het bijzonder tonen we aan dat extended AGs precies overeenkomen met MSO en dat de uitbreiding naar relationele extended AGs de expressieve kracht niet verhoogt. Verder tonen we aan dat testen of een extended AG de lege query uitdrukt en testen of twee extended AGs equivalent zijn compleet zijn voor EXPTIME.

Hierna verlaten we attribuutgrammatica's en leggen we ons toe op een ander berekeningsmodel voor bomen: de boomautomaat. We willen begrijpen hoe zulke automaten, zowel over begrensde als onbegrensde bomen, kunnen worden aangewend om selectiequeries uit te drukken over gestructureerde documenten. We definiëren een queryautomaat als een deterministische boomautomaat die in twee richtingen over de inputboom kan lopen uitgebreid met een selectiefunctie. Eerst karakteriseren we de expressiviteit van het formalisme als de unaire queries uitdrukbaar in MSO. Verrassend genoeg moeten we speciale "stay-transities" toevoegen aan de query automaat in het geval van onbegrensde bomen om volledig MSO te bereiken. Dit was niet nodig voor de queryautomaten over begrensde bomen. Verder bestuderen we de optimalisatieproblemen reeds behandeld bij extended AGs. Ook hier tonen we aan dat beide problemen compleet zijn voor EXPTIME.

We beëindigen deze thesis met het toepassen van de bekomen resultaten. Eerst verbeteren we drastisch de complexiteit van de equivalentietest voor Region Algebra expressies van hyperexponentieel naar EXPTIME. Ons algoritme benadert veel meer de reeds gekende coNP-ondergrens. Hierna tonen we aan, gebruik makende van de technieken ontwikkeld in deze thesis, dat reeds een zeer gerestricteerd deel van de XML transformatietaal XSLT in staat is om alle unaire queries definieerbaar in MSO uit te drukken. Dit misschien wel verrassende resultaat geeft een idee van de expressieve kracht van XSLT en toont de robuustheid van de taal aan. Verder bewijzen we formeel dat de talen onderzocht in deze thesis expressiever zijn dan de meeste talen ontwikkeld voor het semi-gestructureerd datamodel. We eindigen met een voorstel om BAGs en RAGs effectief te implementeren. Meer in het bijzonder tonen we aan dat BAGs en RAGs op een zeer eenvoudige manier vertaald kunnen worden naar datalogprogramma's met negatie. Dit geeft aan dat deductieve databases een natuurlijk platform zijn waarop deze talen kunnen worden geïmplementeerd.