Static Analysis of XML Transformation and Schema Languages

Wim MARTENS

DOCTORAATSPROEFSCHRIFT

2006 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Static Analysis of XML Transformation and Schema Languages

Proefschrift voorgelegd tot het behalen van de graad van Doctor in de Wetenschappen: informatica, te verdedigen door:

Wim MARTENS

Promotor: Prof. dr. Frank NEVEN

Universiteit Maastricht

universiteit hasselt

Acknowledgments

I would like to take this opportunity to thank the many people who have contributed either directly or indirectly to the development of this thesis.

First and foremost, I am deeply indebted to my advisor, Frank Neven, for his continuing guidance, support, and valuable advice during the entire evolution of this thesis. Needless to say, this dissertation would not have been possible without him. He was always prepared for technical discussions and he introduced me to several very interesting and quite challenging problems, especially in the beginning of my doctoral research. As a result, most of this thesis is joint work with him. I consider myself lucky to have been his student. Further, I am very grateful to Thomas Schwentick. His contributions and insights were crucial in the very enjoyable collaboration which lead to the work presented in Chapters 8 and 9. Finally, Chapter 10 has benefitted from a collaboration with Joachim Niehren. Several results in Sections 10.2 and 10.4 were obtained with his collaboration.

I want to thank the members of our research group for creating a stimulating environment. Among them, I especially want to mention Marc Gyssens for providing me with many tips to improve my writing, and Jan Van den Bussche and Stijn Vansummeren for many discussions and comments on my work. Also, I want to thank Stijn for always being prepared to help out with computer related problems. Moreover, as I have a continuing urge to talk about recent results and to find out what other people are working on, I thank my colleages for many pleasant discussions.

On a more personal note, I would like to thank my parents and my brother for their unconditional and continuing support. Also, I would like to thank the regular musicians I played with during the last years (Jan, Jan, Jelle, Kurt, Luk, Natalia, Tom, and Wim), for being great friends and for providing me with one of the best means possible to wind down after a week of work. Finally, I certainly would like to thank my wife, Beate, for her constant encouragement and support during my doctoral research and the writing of this thesis, and for being the best partner in life I can imagine. You're the best!

Thank you all.

Diepenbeek, March 2006

Contents

1

Introduction

XML (eXtensible Markup Language) has currently evolved to the standard data exchange format for the World Wide Web [ABS99]. Its main advantages are that it offers an intuitive and standard way of structuring a very wide range of data and that it admits the use of user-defined tags. The latter allows user communities to develop their own format of XML documents, which is defined by an *XML schema*. The presence of such a schema improves the efficiency of many tasks like, for instance, query processing, query optimization, and automatic data integration.

The XML Typechecking Problem

In the context of the World Wide Web, schemas can be used to validate data exchange. In a typical scenario, a user community agrees on a common schema and on producing only XML data conforming to that schema. This raises the issue of typechecking: verifying at compile time that every XML document which is the result of a specified query or transformation applied to a valid input, satisfies the output schema [Suc01, Suc02].

The first part of this dissertation studies the typechecking problem for *XML to XML transformations*. Typechecking consists of statically verifying whether the output of an XML transformation is always conform to an output type for documents satisfying a given input type. As types we adopt formal models for XML schemas: *Document Type Definitions* (DTDs) and their robust extension, *regular tree languages* [BKMW01, LMM00, MSV03] or, equivalently, *extended DTDs* [PV00, BPV04].[1] The latter serve as a formal model for Relax NG [CM01].

Obviously, typechecking depends on the transformation language at hand. As shown by Alon et al. [AMN$^+$03a, AMN$^+$03b], when transformation languages have the

[1] Papakonstantinou and Vianu used the term *specialized DTD* as types *specialize* tags. We prefer the term *extended DTD* as it expresses more clearly that the power of the schemas is amplified.

ability to compare data values, the typechecking problem quickly turns undecidable. However, Milo, Suciu, and Vianu argued that XML documents can be abstracted by labeled ordered trees and that the capability of most XML transformation languages can be encompassed by k-pebble transducers when data values are ignored [MSV03]. Further, the authors showed that the typechecking problem in this context is decidable. More precisely, given two types τ_1 and τ_2, represented by tree automata, and a k-pebble transducer T, it is decidable whether $T(t) \in \tau_2$ for all $t \in \tau_1$. Here, $T(t)$ is the tree obtained by running T on input t. The complexity, however, is non-elementary and cannot be improved [MSV03].

In an attempt to lower the complexity, we consider much simpler tree transformations, which correspond to structural recursion on trees [BFS00] and to simple top-down XSLT transformations [BMN02, Cla99]. Such transformations are merely used for restructuring and filtering, not for advanced querying. In brief, a transformation consists of a single top-down traversal of the input tree where every node is replaced by a new tree (possibly the empty tree).

Our work studies sound and complete typechecking algorithms, an approach that should be contrasted with the work on general-purpose XML programming languages like XDuce [HP03] and CDuce [BFC03], for instance, where the main objective is fast and sound typechecking. The latter kind of typechecking is always incomplete due to the Turing-completeness of the considered XML-transformations. That is, it can happen that type safe transformations are rejected by the typechecker. As we only consider very simple transformations which are by no means Turing-complete, it makes sense to ask for complete algorithms.

In **Chapter 2**, we formally define the basic notions that are used throughout the first part of the dissertation, such as tree languages, tree transformations, and schema languages.

Chapter 3 provides the theoretical tools and basic results that we will use in the subsequent chapters. We extend Lenstra's polynomial time algorithm for solving a conjunction of linear inequalities with a fixed number of variables to the case where we allow arbitrary Boolean combinations of linear inequalities. Further, we give an overview of the complexity of emptiness, universality, inclusion, and intersection emptiness problems of finite automata over strings and trees.

The actual study of the complexity of XML typechecking starts in **Chapter 4**. We parameterize the problem by several restrictions on the transformations (deleting, non-deleting, bounded or unbounded copying) and consider both tree automata and DTDs as input and output schemas. The overall goal of the chapter is to identify a non-trivial scenario in which the typechecking problem is in polynomial time. The complexity of the typechecking problems in the considered scenarios ranges from exponential time in the most general case to polynomial time in the case where deletion in the transformations is completely disallowed and the number of copies that the transformation makes is bounded in advance.

In **Chapter 5**, we note that the scenario that is studied in Chapter 4 is very general: both the schemas and the transformation are determined to be part of the input. However, for some exchange scenarios, it makes sense to consider the input and/or output schema to be fixed when transformations are always from within and/or to a specific community.

Therefore, we revisit the various instances of the typechecking problem considered in Chapter 4 and determine the complexity in the presence of fixed input and/or output schemas. The main goal of this chapter is to investigate to which extent the complexity of the typechecking problem is lowered in scenarios where the input and/or output schema is fixed.

Although Chapters 4 and 5 give a fairly detailed overview of the complexity of typechecking, the settings in which we have found a polynomial time typechecking algorithm are very restrictive, especially since they exclude any form of deletion in the transformation. Indeed, many simple filtering transformations use deletion as they select specific parts of the input while ignoring the non-interesting ones.

Therefore, the purpose of **Chapter 6** is to investigate larger and more flexible classes for which the complexity of the typechecking problem remains in polynomial time. By restricting schema languages and transformations, we identify several practical settings for which typechecking can be done in polynomial time. Moreover, the resulting framework provides a rather complete picture as we show that most scenarios can not be enlarged without rendering the typechecking problem intractable. Hence, the work sheds light on when to use fast complete algorithms and when to reside to sound but incomplete ones.

The tractable fragments that we identify can be divided into two classes. In the first class, tractability of the typechecking problem is obtained by bounding the *deletion path width* of the tree transformations. The deletion path width is a notion that measures the number of times that a tree transformation copies part of its input. The set of tree transformations with a bounded deletion path width strictly includes the tractable class that we identified in Chapter 4, but most importantly, it also contains tree transducers which delete in a limited manner (even recursively). In the second class, we allow all transformations (that is, the most general transformations that we allowed in Chapter 4), but we restrict the expressive power of the schema languages. We obtain that typechecking is tractable when the schemas use a restricted form of regular expressions, which we call RE^{+} expressions. Again, we show that allowing obvious extensions of RE^{+} expressions make typechecking at least coNP-hard.

We present conclusions and additional remarks on our study of the XML typechecking problem in **Chapter 7**. For instance, when a transformation does not typecheck with respect to an input and output schema, it is important to give the user some feedback *why* the transformation is not type safe. We therefore investigate the problem of producing a witness tree t in the input schema, such that the transformation of t is not in the output schema. We obtain that, for each of the tractable fragments we identified in Chapter 6, we can generate a description of such a witness tree t in polynomial time.

Foundations of XML Schema Languages

While the first part of the dissertation focuses on the interaction between XML transformation and schema languages, the second part is entirely devoted to the study of XML schema languages.

The common abstraction of XML Schema by unranked regular tree languages is not entirely accurate. In **Chapter 8** we therefore shed some light on the actual expressive power of XML Schema, by providing intuitive semantical characterizations of the Element Declarations Consistent (EDC) rule. In particular, it is obtained that schemas satisfying EDC can only reason about regular properties of ancestors of nodes. It is argued that cleaner, more robust, larger but equally feasible classes can be obtained by replacing EDC with the notion of *1-pass preorder typeability (1PPT)* or *top-down typeability (TDT).* The former are schemas that allow to determine the type of an element of a streaming document when its opening tag is met and the latter are schemas that allow to determine the type of an element without investigating its descendants or the descendants of its siblings. It is shown that the one-pass preorder typeable schemas are exactly the restrained competition tree grammars introduced by Murata et al. [MLMK05] and that the top-down typeable schemas are a natural generalization of top-down deterministic tree automata. As a result, the expressive power of schemas strictly grows when going from EDC to 1PPT, and from 1PPT to TDT.

We characterize the expressive power of the EDC rule, 1PPT and TDT in terms of the context of nodes, closure properties, allowed patterns and guarded schemas. It is further shown that deciding whether a schema allows 1PPT or TDT is tractable. Deciding whether a schema is equivalent to a 1PPT grammar, or one of its subclasses, is much more difficult: it is complete for EXPTIME.

The focus of **Chapter 9** is to investigate the complexity of several basic decision problems for these schemas satisfying EDC, 1PPT, or TDT. We aim to study the inclusion, equivalence, and intersection emptiness problem for XML schemas occurring in practice. Such practical schemas make use of regular expressions with a very simple structure: they basically consist of the concatenation of factors, where each factor is a disjunction of strings, possibly extended with "$*$", "$+$", or "?". We refer to these as CHAin Regular Expressions (CHAREs). We obtain lower and upper bounds for various fragments of CHAREs and also consider additional determinism and occurrence constraints. For the equivalence problem, we only prove an initial tractability result, leaving the complexity of more general cases open. We relate the above to optimization of XML schema languages by showing that all our lower and upper bounds for the inclusion and equivalence problem carry over to the corresponding decision problems for DTDs and to schemas with the EDC. A similar but slightly weaker result holds for schemas with the 1PPT and TDT restriction. For the intersection problem, we show that obtained complexities only carry over to DTDs. We show that a similar result does not hold for single-type, restrained competition, or top-down typeable EDTDs, unless PSPACE = EXPTIME.

Finally, in **Chapter 10**, we turn to the problem of *optimizing* formal models for XML schemas. The aim of the chapter is to investigate models which allow to efficiently minimize the number of states (or types) needed to define the schema.

First, we study the minimization problem for *unranked tree automata*, which are a formal basis for Relax NG [MLMK05, CM01]. We show that the minimization problem for the bottom-up deterministic unranked tree automata defined by Brüggeman-Klein, Murata, and Wood is NP-complete when their transition function is represented by deterministic finite automata [BKMW01]. Moreover, we show that minimal automata in the later class are not unique up to isomorphism.

Second, we investigate more recent automata classes that do allow for polynomial time minimization. We present a comparative study between *stepwise tree automata* [CNT04], *parallel tree automata* [CLT05, RB04], and bottom-up deterministic tree automata over the *first-child next-sibling* encoding of unranked trees. Among those, we show that bottom-up deterministic stepwise tree automata yield the most succinct representations.

Finally, we investigate minimization for schemas with the 1PPT constraint, in which we use deterministic finite automata to represent content models. We show that the latter notion allows for polynomial time minimization and unique minimal models. Interestingly, we show that these results also carry over to schemas with the EDC constraint, since our minimization algorithm preserves the EDC restriction on the input.

We present concluding remarks in **Chapter 11**.

Extended abstracts and articles containing most results in this dissertation are published as [MN05a, MN04, MN06, MNG05, MNS05, BMNS05, MNS04, MN05b].

Part I

The XML Typechecking Problem

2

Preliminaries

We provide the necessary background on trees, automata, and tree transducers. In the following, Σ always denotes a finite set, which we call the *alphabet*.

By $\mathbb{N}$ we denote the set of natural numbers and by $\mathbb{N}_0$ we denote $\mathbb{N} - \{0\}$. A *string* $w = a_1 \cdots a_n$ over Σ is a finite sequence of Σ-symbols $a_1, \ldots, a_n$. When $n = 0$, we say that the string w is *empty*, and we denote w by ε. We say that a Σ-symbol a *occurs* in w if there exists an a_i, $1 \leq i \leq n$, such that $a = a_i$. We say that w has k *occurrences of* a if the set $\{a_i \mid 1 \leq i \leq n, a = a_i\}$ contains exactly k elements. The set of *positions*, or the *nodes of* w is $\text{Nodes}(w) = \{1, \ldots, n\}$. The *length of* w, denoted by $|w|$, is the number n of Σ-symbols occurring in w. The *label* a_i of position i in w is denoted by $\text{lab}^w(i)$. By $w_1 \cdot w_2$ we denote the *concatenation* of two strings w_1 and w_2. For readability, we usually denote the concatenation of w_1 and w_2 by $w_1 w_2$.

For a set S, we denote by S^* the Kleene closure of S, that is, the set of all strings over alphabet S. By S^+ we abbreviate SS^*. We define the *size* $|S|$ of a finite set S to be the number of elements in S. A *string language* is a subset of Σ^*. For two string languages $L, L' \subseteq \Sigma^*$, we define their concatenation $L \cdot L'$ to be the set $\{w \cdot w' \mid w \in L, w' \in L'\}$.

Definition 2.1. The set of *regular expressions* over Σ, denoted by RE, is defined as follows:

- $\emptyset$, ε, and every Σ-symbol is a regular expression; and
- when r and s are regular expressions, then rs, $(r)+(s)$, and $(r)^*$ are also regular expressions. ◇

For readability, we often do not write all parenthesis symbols "(" and ")" in regular expressions. The *language* defined by a regular expression r, denoted by $L(r)$, is inductively defined as follows:

- $L(\emptyset) = \emptyset$;
- $L(\varepsilon) = \{\varepsilon\}$;
- $L(a) = \{a\}$;
- $L(rs) = L(r) \cdot L(s)$;
- $L(r + s) = L(r) \cup L(s)$; and
- $L(r^*) = \{\varepsilon\} \cup \bigcup_{i=1}^{\infty} L(r)^i$.

By $r?$ and r^+, we abbreviate the expressions $r + \varepsilon$ and rr^*, respectively. The *size* of a regular expression r over Σ, denoted by $|r|$, is the total number of occurrences of symbols from $\Sigma \cup \{+, *, ?, \varepsilon\}$ in r.

Definition 2.2. A *nondeterministic finite automaton (NFA)* over alphabet Σ is a 5-tuple $N = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of *states*, $\delta : Q \times \Sigma \to 2^Q$ is the *transition function*, $I \subseteq Q$ is the set of *initial states*, and $F \subseteq Q$ is the set of *final states*. ◇

A *run* ρ of N on a string $w \in \Sigma^*$ is a mapping from Nodes(w) to Q such that $\rho(1) \in \delta(q, \text{lab}^w(1))$ for a $q \in I$, and for every $i = 1, \ldots, |w|-1$, $\rho(i+1) \in \delta(\rho(i), \text{lab}^w(i+1))$. A run is *accepting* if $\rho(|w|) \in F$. A string w is *accepted* by N if there exists an accepting run of N on w. The *language accepted by* N is defined to be the set of strings that are accepted by N, and is denoted by $L(N)$. The *size* $|N|$ of N is defined as $|Q| + |\Sigma| + \sum_{q \in Q, a \in \Sigma} |\delta(q, a)|$. We call a string language L *regular* if there exists an NFA N such that $L = L(N)$.

We extend the definition of δ to strings by defining a function $\delta^* : Q \times \Sigma^* \to 2^Q$ as follows:

- for every $q \in Q$, $\delta^*(q, \varepsilon) := \{q\}$; and,
- for every $q \in Q, w \in \Sigma^*$, and $a \in \Sigma$, $\delta^*(q, w \cdot a) := \{p \mid \exists p' \in \delta^*(q, w) : p \in \delta(p', a)\}$.

Definition 2.3. A *deterministic finite automaton (DFA)* is an NFA $(Q, \Sigma, \delta, I, F)$ where *(i)* I is a singleton and *(ii)* $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. ◇

Definition 2.4. An *unambiguous finite automaton (UFA)* is an NFA N such that, for every string $w \in L(N)$ there exists a unique accepting run of N on w. ◇

To define unordered string languages, we make use of the *Specification Language SL* inspired by Neven and Schwentick [NS] and also used by Alon et al. [AMN$^+$03a, AMN$^+$03b]. The syntax of this language is as follows:

Definition 2.5. For every $a \in \Sigma$ and natural number $i \in \mathbb{N}$, $a^{=i}$ and $a^{\geq i}$ are *atomic SL-formulas*; "true" is also an atomic SL-formula. Every atomic SL-formula is an SL-formula and the negation, conjunction, and disjunction of SL-formulas are also SL-formulas. ◇

A string w over Σ *satisfies* an atomic SL-formula $a^{=i}$ if w has i occurrences of a; w satisfies $a^{\geq i}$ if it has at least i occurrences of a. Furthermore, "true" is satisfied by every string. Satisfaction of Boolean combinations of atomic formulas is defined in the obvious way.[1] By $w \models \phi$, we denote that w satisfies the SL-formula ϕ. By $L(\phi)$ we denote the set of strings that satisfy ϕ. We sometimes also refer to $L(\phi)$ as the *language defined by* ϕ.

As an example, consider the SL-formula $\neg(a^{\geq 1} \wedge \neg b^{\geq 1})$. This expresses the constraint that the symbol a can only occur when symbol b occurs. The *size* of an SL-formula is the number of symbols that occur in it, that is, Σ-symbols, logical symbols, and numbers (every i in $a^{=i}$ or $a^{\geq i}$ is written in binary notation).

2.1 Trees and Hedges

It is common to view XML documents as finite trees with labels from a finite alphabet Σ. Figures 2.1(a) and 2.1(b) give an example of an XML document together with its tree representation. Of course, elements in XML documents can also contain references to nodes. But, as XML schema languages often do not constrain these nor the data values at leaves, it is safe to view schemas as simply defining tree languages over a finite alphabet. In the rest of this section, we introduce the necessary background concerning labeled unranked trees and hedges, which are finite sequences of such trees.

The set of *unranked* Σ*-trees*, denoted by $\mathcal{T}_\Sigma$, is the smallest set of strings over Σ and the parenthesis symbols "(" and ")" such that, for $a \in \Sigma$ and $w \in \mathcal{T}_\Sigma{}^*$, $a(w)$ is in $\mathcal{T}_\Sigma$.[2] So, a tree is either ε (empty) or is of the form $a(t_1 \cdots t_n)$ where each t_i is a tree. In the tree $a(t_1 \cdots t_n)$, the subtrees $t_1, \ldots, t_n$ are attached to a root labeled a. For ease of notation, we write a rather than $a()$. A *tree language* is a set of trees. For every $t \in \mathcal{T}_\Sigma$, the *set of tree-nodes* of t, denoted by $\text{Nodes}_T(t)$, is the set defined as follows:

(i) if $t = \varepsilon$, then $\text{Nodes}_T(t) = \emptyset$; and,

(ii) if $t = a(t_1 \cdots t_n)$, where each $t_i \in \mathcal{T}_\Sigma$, then $\text{Nodes}_T(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{iu \mid u \in \text{Nodes}_T(t_i)\}$.

For two nodes u and v in $\text{Nodes}_T(t)$, we say that *(i)* u is a *child* of v, or v is the *parent* of u if $u = vi$ with $i \in \mathbb{N}_0$, *(ii)* u is a *left sibling* of v, or v is a *right sibling* of u if $u = wi$ and $v = wj$ with $i, j \in \mathbb{N}_0$ and $i < j$, and that *(iii)* u is a *descendant* of v, or v is an *ancestor* of u if $u = vi_1 \cdots i_n$ for $i_1, \ldots, i_n \in \mathbb{N}_0^+$. We say that u and v are *siblings* if u and v have the same parent. An *internal node* is a node with children and a *leaf node* is a node without any children. A *path* in a tree is a sequence of nodes $u_1, \ldots, u_n$ such that, for each $i = 1, \ldots, n-1$, u_{i+1} is a child of u_i.

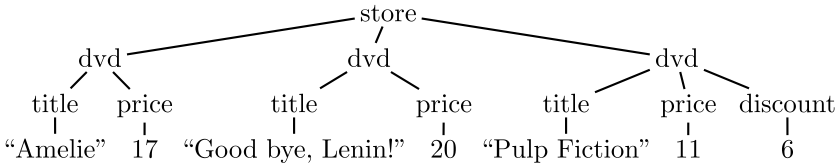
Note that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked*. Figure 2.2(a) contains a tree in which we annotated the nodes between brackets. Observe that the n child nodes of a node u are

[1] The empty string is obtained as $\bigwedge_{a \in \Sigma} a^{=0}$ and the empty set as $\neg$true.

[2] We assume that the paranthesis symbols are not in Σ.

```
<store>
  <dvd>
    <title> "Amelie" </title>
    <price> 17 </price>
  </dvd>
  <dvd>
    <title> "Good bye, Lenin!" </title>
    <price> 20 </price>
  </dvd>
  <dvd>
    <title> "Pulp Fiction" </title>
    <price> 11 </price>
    <discount> 6 </discount>
  </dvd>
</store>
```

(a) An example XML document.



(b) Its tree representation with data values.

Figure 2.1: An example of an XML document and its tree representation.

always $u1, \dots, un$, from left to right. The *label* of a node u in the tree $t = a(t_1 \cdots t_n)$, denoted by $\mathrm{lab}^t_T(u)$, is defined as follows:

(i) if $u = \varepsilon$, then $\mathrm{lab}^t_T(u) = a$; and,

(ii) if $u = iu'$, then $\mathrm{lab}^t_T(u) = \mathrm{lab}^{t_i}_T(u')$.

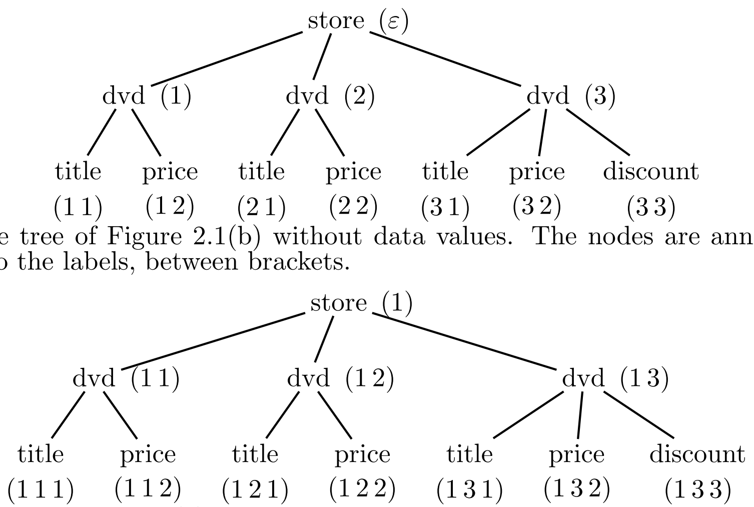
We define the *depth* of a tree t, denoted by $\mathrm{depth}(t)$, as follows: if $t = \varepsilon$, then $\mathrm{depth}(t) = 0$; and if $t = a(t_1 \cdots t_n)$, then $\mathrm{depth}(t) = \max\{\mathrm{depth}(t_i) \mid 1 \leq i \leq n\} + 1$. The *depth* of a node $i_1 \cdots i_n \in \mathbb{N}_0^*$ in a tree is $n + 1$. In the sequel, whenever we say tree, we always mean Σ-tree. A *tree language* is a set of trees.

An *unranked hedge* is a finite sequence of unranked trees. Hence, the set of hedges, denoted by $\mathcal{H}_\Sigma$, equals $\mathcal{T}_\Sigma^*$. For every hedge $h \in \mathcal{H}_\Sigma$, the *set of hedge-nodes of* h, denoted by $\mathrm{Nodes}_H(h)$, is the subset of $\mathbb{N}_0^*$ defined as follows:

(i) if $h = \varepsilon$, then $\mathrm{Nodes}_H(h) = \emptyset$; and,

(ii) if $h = t_1 \cdots t_n$ and each $t_i \in \mathcal{T}_\Sigma$, then $\mathrm{Nodes}_H(h) = \bigcup_{i=1}^n \{iu \mid u \in \mathrm{Nodes}_T(t_i)\}$.

The *label* of a node $u = iu'$ in the hedge $h = t_1 \cdots t_n$, denoted by $\mathrm{lab}^h_H(u)$, is defined as $\mathrm{lab}^h_H(u) = \mathrm{lab}^{t_i}_T(u')$. Note that the set of hedge-nodes of a hedge consisting of one tree is different from the set of tree-nodes of this tree. For example: if the tree in



(a) The tree of Figure 2.1(b) without data values. The nodes are annotated next to the labels, between brackets.

(b) Tree of Figure 2.2(a) viewed as a hedge. The nodes are annotated next to the labels, between brackets.

Figure 2.2: The document of Figure 2.1 without data values, viewed as a tree and as a hedge.

Figure 2.2(a) were to represent a single-tree hedge, it would have the set of hedge-nodes $\{1, 11, 12, 13, 111, 112, 121, 122, 131, 132, 133\}$, as shown in Figure 2.2(b). The *depth* of the hedge $h = t_1 \cdots t_n$, denoted by $\mathrm{depth}(h)$, is defined as $\max\{\mathrm{depth}(t_i) \mid i = 1, \ldots, n\}$. For a hedge $h = t_1 \cdots t_n$, we denote by $\mathrm{top}(h)$ the string obtained by concatenating the root symbols of $t_1, \ldots, t_n$, that is, $\mathrm{lab}_H^{t_1}(1) \cdots \mathrm{lab}_H^{t_n}(n)$.

In this thesis, we adopt the following conventions: we use $t, t_1, t_2, \ldots$ to denote trees and $h, h_1, h_2, \ldots$ to denote hedges. Hence, when we write $h = t_1 \cdots t_n$ we tacitly assume that every t_i is a tree. We denote Nodes_T and Nodes_H simply by Nodes, and we denote lab_T and lab_H by lab when it is understood from the context whether we are working with trees or hedges.

2.2 DTDs and Tree Automata

We use extended context-free grammars and tree automata to abstract from Document Type Definitions (DTDs) and the various proposals for XML schemas. We parameterize the definition of DTDs by a class of representations $\mathcal{M}$ of regular string languages such as, for instance, the class of DFAs (Deterministic Finite Automata) or NFAs (Non-deterministic Finite Automata). For $M \in \mathcal{M}$, we denote by $L(M)$ the set of strings accepted by M. We then abstract DTDs as follows:

Definition 2.6. Let $\mathcal{M}$ be a class of representations of regular string languages over Σ. A *Document Type Definition (DTD)* is a triple (Σ, d, s_d) where d is a function that maps Σ-symbols to elements of $\mathcal{M}$ and $s_d \in \Sigma$ is the start symbol. ◇

For convenience of notation, we denote (Σ, d, s_d) by d and leave the alphabet Σ

and start symbol s_d implicit whenever this cannot give rise to confusion. A tree t *satisfies* d if *(i)* $\text{lab}^t(\varepsilon) = s_d$ and, *(ii)* for every $u \in \text{Nodes}(t)$ with n children, $\text{lab}^t(u1) \cdots \text{lab}^t(un) \in L(d(\text{lab}^t(u)))$. By $L(d)$ we denote the set of trees satisfying d, also called the tree language defined by d.

For a DTD (Σ, d, s_d) and Σ-symbol a, we denote by (d, a) the DTD (Σ, d, a), that is, the DTD d where a is the start symbol.

Given a DTD d, we say that a Σ-symbol a *occurs in* $d(b)$ when there exist Σ-strings w_1 and w_2 such that $w_1 a w_2 \in L(d(b))$. We say that a *occurs in* d if a occurs in $d(b)$ for some $b \in \Sigma$.

We denote by $\text{DTD}(\mathcal{M})$ the class of DTDs where the regular string languages are represented by elements of $\mathcal{M}$. The *size*$|d|$ of a DTD d is the sum of the sizes of the elements of $\mathcal{M}$ used to represent the function d.

To improve readability in examples, we write the regular languages in DTDs as regular expressions. For clarity, we also write $a \to r$ rather than $d(a) = r$. We also do not list rules of the form $a \to \varepsilon$. We give an example of a DTD:

Example 2.7. The following DTD with start symbol *store* is satisfied by the tree in Figure 2.2(a):

$$\begin{array}{lll} \text{store} & \to & \text{dvd dvd}^* \\ \text{dvd} & \to & \text{title price (discount} + \varepsilon) \end{array}$$

This DTD defines the set of trees where the root is labeled with *store*; the children of *store* are all labeled with *dvd*; and every *dvd*-labeled node has a *title*, *price*, and an optional *discount* child. ◇

In some cases, our algorithms are easier to explain on well-behaved DTDs as considered next. A DTD d is *reduced* if, for every symbol a that occurs in d, there exists a tree $t \in L(d)$ and a node $u \in \text{Nodes}(t)$ such that $\text{lab}^t(u) = a$. Hence, for example, the DTD $(\{a\}, d, a)$ where $d(a) = a$ is *not* reduced. Reducing a DTD(DFA) is in PTIME, while reducing a DTD(SL) is in CONP (we treat this in Corollary 3.17 in Section 3.3).

We recall the definition of non-deterministic unranked tree automata from Brüggemann-Klein, Murata, and Wood [BKMW01].

Definition 2.8. A *nondeterministic unranked tree automaton (NTA)* is a quadruple $B = (Q, \Sigma, \delta, F)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^{Q^*}$ is a function such that $\delta(q, a)$ is a regular string language over Q for every $a \in \Sigma$ and $q \in Q$. ◇

A *run* of B on a tree t is a labeling $\lambda : \text{Nodes}(t) \to Q$ such that, for every $v \in \text{Nodes}(t)$ with n children, $\lambda(v1) \cdots \lambda(vn) \in \delta(\lambda(v), \text{lab}^t(v))$. Notice that, when v has no children, the criterion reduces to $\varepsilon \in \delta(\lambda(v), \text{lab}^t(v))$. A run is *accepting* if the root is labeled with an accepting state, that is, $\lambda(\varepsilon) \in F$. When $\lambda(v) = q$, we sometimes also say that B *assigns* q to v. A tree is *accepted* if there is an accepting run of B on t. The set of all accepted trees is denoted by $L(B)$ and is called the *language defined by* B. We call a tree language L *regular* if there exists an NTA B such that $L = L(B)$.

We extend the definition of δ to trees and hedges by defining a function $\delta^*(h) : \mathcal{T}_\Sigma \cup \mathcal{H}_\Sigma \to (2^Q)^*$ as follows:

- $\delta^*(a) := \{q \mid \varepsilon \in \delta(q,a)\}$;
- $\delta^*(a(t_1 \cdots t_n)) := \{q \mid \exists q_1 \in \delta^*(t_1), \ldots, \exists q_n \in \delta^*(t_n)$ such that $q_1 \cdots q_n \in \delta(q,a)\}$; and,
- $\delta^*(t_1 \cdots t_n) := \delta^*(t_1) \cdots \delta^*(t_n)$.

Notice that a tree t is accepted by B if and only if $\delta^*(t) \cap F \neq \emptyset$.

Definition 2.9. A *bottom-up deterministic unranked tree automaton (DTA)* is an NTA (Q, Σ, δ, F) such that, for all $q, q' \in Q$ with $q \neq q'$ and $a \in \Sigma$, we have that $\delta(q,a) \cap \delta(q',a) = \emptyset$. $\diamond$

Definition 2.10. An *unambiguous unranked tree automaton (UTA)* is an NTA B such that, for every tree $t \in L(B)$, there is a unique accepting run of B on t. $\diamond$

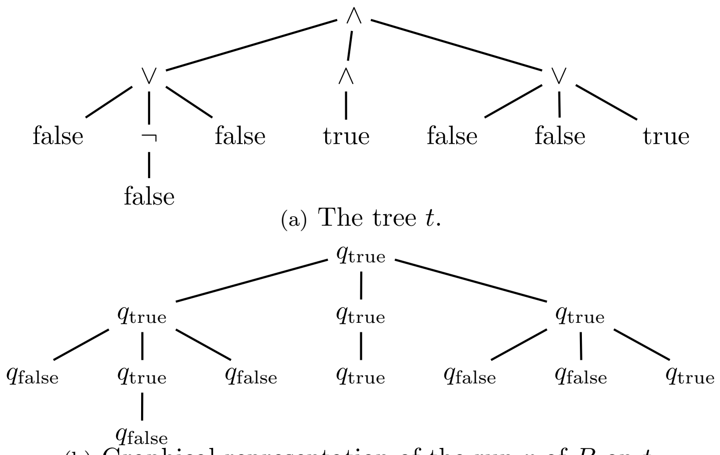
As for DTDs, we often denote the regular languages in the transition function of an NTA by regular expressions. We now give an example of an NTA.

Example 2.11. We give a bottom-up deterministic unranked tree automaton $B = (Q, \Sigma, \delta, F)$ which accepts the parse trees of well-formed Boolean expressions that are true. Here, the alphabet Σ is $\{\wedge, \vee, \neg, \text{true}, \text{false}\}$. The state set Q contains the states q_{true} and q_{false}, and the accepting state set F is the singleton $\{q_{\text{true}}\}$. The transition function of B is defined as follows:

- $\delta(q_{\text{true}}, \text{true}) = \varepsilon$. We assign the state q_{true} to leafs with label *true*.
- $\delta(q_{\text{false}}, \text{false}) = \varepsilon$. We assign the state q_{false} to leafs with label *false*.
- $\delta(q_{\text{true}}, \wedge) = q_{\text{true}} q_{\text{true}}^*$.
- $\delta(q_{\text{false}}, \wedge) = (q_{\text{true}} + q_{\text{false}})^* q_{\text{false}} (q_{\text{true}} + q_{\text{false}})^*$.
- $\delta(q_{\text{true}}, \vee) = (q_{\text{true}} + q_{\text{false}})^* q_{\text{true}} (q_{\text{true}} + q_{\text{false}})^*$.
- $\delta(q_{\text{false}}, \vee) = q_{\text{false}} q_{\text{false}}^*$.
- $\delta(q_{\text{true}}, \neg) = q_{\text{false}}$.
- $\delta(q_{\text{false}}, \neg) = q_{\text{true}}$.

Consider the tree t depicted in Figure 2.3(a). The unique accepting run r of B on t can be graphically represented as shown in Figure 2.3(b). Formally, the run of B on t is the function $\lambda : \text{Nodes}(t) \to Q : u \mapsto \text{lab}^r(u)$. Note that B is a DTA. $\diamond$

Analogously to DTDs, we parameterize NTAs by the formalism used to represent the regular languages in the transition functions $\delta(q,a)$. So, for a class of representations of regular languages $\mathcal{M}$, we denote by NTA($\mathcal{M}$) the class of NTAs where all transition functions are represented by elements of $\mathcal{M}$. The *size* $|B|$ of an automaton B then is $|Q| + |\Sigma| + \sum_{q \in Q, a \in \Sigma} |\delta(q,a)|$. Here, by $|\delta(q,a)|$ we denote the size of the automaton accepting $\delta(q,a)$. Unless explicitly specified otherwise, we assume that $\delta(q,a)$ is always represented by an NFA.



(a) The tree t.

(b) Graphical representation of the run r of B on t.

Figure 2.3: Illustrations for Example 2.11.

2.3 Transducers

We adhere to transducers as a formal model for simple transformations corresponding to structural recursion [BFS00] and a fragment of top-down XSLT. As in the work by Milo, Suciu, and Vianu [MSV03], the abstraction focuses on structure rather than on content. We next define the tree transducers used in this paper. To simplify notation, we restrict ourselves to one alphabet. That is, we consider transducers mapping Σ-trees to Σ-trees.[3]

For a set Q, denote by $\mathcal{H}_\Sigma(Q)$ (respectively $\mathcal{T}_\Sigma(Q)$) the set of Σ-hedges (respectively trees) where leaf nodes are labeled with elements from $\Sigma \cup Q$ instead of only Σ.

Definition 2.12. A *tree transducer* is a quadruple $T = (Q, \Sigma, q^0, R)$, where Q is a finite set of states, Σ is the input and output alphabet, $q^0 \in Q$ is the initial state, and R is a finite set of rules of the form $(q, a) \to h$, where $a \in \Sigma$, $q \in Q$, and $h \in \mathcal{H}_\Sigma(Q)$. When $q = q^0$, h is restricted to be either empty, or consist of only one tree with a Σ-symbol as its root label.

We say that a tree transducer is *determistic* if, for every pair (q, a), there is at most one rule $(q, a) \to h$ in R. In this dissertation, we always assume that tree transducers are deterministic. ◇

The restriction to trees on the right hand sides of rules with the initial state ensures that the output of a tree transducer is always a tree rather than a hedge.

The *translation* defined by a deterministic tree transducer $T = (Q, \Sigma, q^0, R)$ on a tree t in state q, denoted by $T^q(t)$, is inductively defined as follows:

(i) if $t = \varepsilon$ then $T^q(t) := \{\varepsilon\}$;

[3] In general, of course, one can define transducers where the input alphabet differs from the output alphabet.

(ii) if $t = a(t_1 \cdots t_n)$ and there is a rule $(q, a) \to h \in R$, then $T^q(t)$ is obtained from h by replacing every node u in h labeled with state p by the hedge $T^p(t_1) \cdots T^p(t_n)$. Note that such nodes u can only occur at leaves. So, h is only extended downwards.

(iii) if $t = a(t_1 \cdots t_n)$ and there is no rule $(q, a) \to h \in R$ then $T^q(t) := \varepsilon$.

Finally, the transformation of t by T, denoted by $T(t)$, is defined as $T^{q^0}(t)$, interpreted as a tree.

Given a tree transducer $T = (Q, \Sigma, q^0, R)$ and a tree t, we say that T *visits* the node $u \in \text{Nodes}(t)$ in state $q \in Q$ if

(i) $q = q^0$ and $u = \varepsilon$; or

(ii) u is a child of v, $\text{lab}^t(v) = a$, T visits v in state p, $(p, a) \to h$ is a rule in R, and h contains a node that is labeled with q.

For $a \in \Sigma$, $q \in Q$ and $(q, a) \to h \in R$, we denote h by $\text{rhs}(q, a)$. If q and a are not important, we say that h is an rhs. The *size* of T is $|Q| + |\Sigma| + \sum_{q \in Q, a \in \Sigma} |\text{rhs}(q, a)|$, where $|\text{rhs}(q, a)|$ denotes the number of nodes in $\text{rhs}(q, a)$. In the remainder of Part I of the thesis, we always use $p, p_1, p_2, \ldots$ and $q, q_1, q_2, \ldots$ to denote states.

Let q be a state of tree transducer T and $a \in \Sigma$. We then define $q_T[a] := \text{top}(T^q(a))$. For a string $w = a_1 \cdots a_n$, we define $q_T[w] := q_T[a_1] \cdots q_T[a_n]$. In the sequel, we leave T implicit whenever T is clear from the context.

We give an example of a tree transducer:

Example 2.13. Let $T = (Q, \Sigma, p, R)$ where $Q = \{p, q\}$, $\Sigma = \{a, b, c, d, e\}$, and R contains the rules

$$\begin{array}{ll} (p, a) \to d(e) & (p, b) \to d(q) \\ (q, a) \to c\ p & (q, b) \to c(p\ q) \end{array}$$

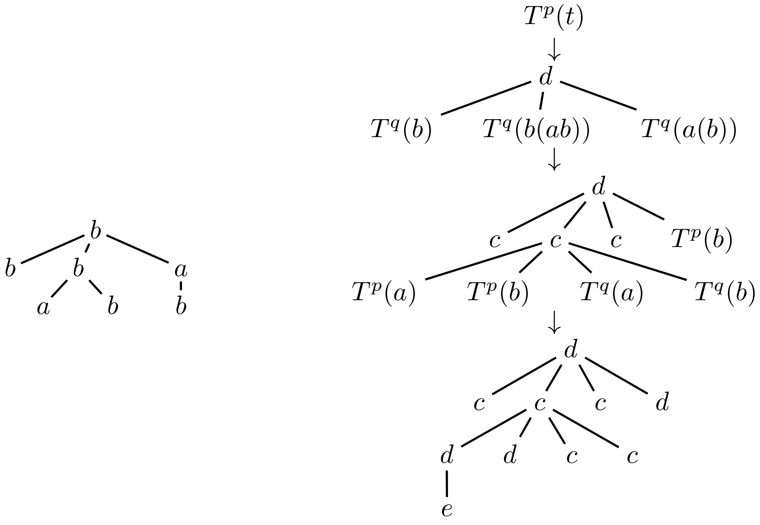
Note that the right-hand side of $(q, a) \to c\ p$ is a hedge consisting of two trees, while the other right-hand sides consist of only one tree. ◇

The following is an example of a transformation of the transducer in Example 2.13.

Example 2.14. Consider the tree t shown in Figure 2.4(a). In Figure 2.4(b) we give the translation of t by the transducer of Example 2.13. In order to keep the example simple, we did not list $T^q(\varepsilon)$ and $T^p(\varepsilon)$ explicitly in the process of translation. ◇

The purpose of our tree transducers is to serve as an abstraction of simple restructuring transformations that occur a lot in practice. Our tree transducers can be implemented as XSLT programs in a straightforward way. For instance, the XSLT program equivalent to the above transducer is given in Figure 2.5 (we assume that the program is started in mode p).

We discuss two important features of tree transducers: *copying* and *deletion*. In Example 2.13, the rule $(q, b) \to c(p\,q)$ copies the children of the current node in the input tree twice: one copy is processed in state p and the other in state q. The symbol c is the parent node of the two copies. So, one could say that the current node is translated in the new parent node labeled c. The rule $(q, a) \to c\,p$ copies the children



(a) The tree t of Example 2.14.

(b) The translation of t by the transducer T of Example 2.13.

Figure 2.4: A tree t and its translation by the tree transducer of Example 2.13.

of the current node only once. However, no parent node is given for this copy. So, there is no node in the output tree that can be interpreted as the translation of the current node in the input tree. We therefore say that it is deleted. For instance, $T^q(a(b)) = c\,d$ where d corresponds to b and not to a.

We illustrate the functionality of copying and deletion by means of a typical transformation that processes its input and filters out the parts that are of interest.

Example 2.15. The following DTD(DFA) d with start symbol *book* defines a schema for books:

$$\begin{array}{lcl}
\text{book} & \to & \text{title author}^+ \text{ chapter}^+ \\
\text{chapter} & \to & \text{title intro section}^+ \\
\text{section} & \to & \text{title paragraph}^+ \text{ section}^*
\end{array}$$

Figure 2.6 depicts a document conforming to the given schema. We now present a tree transducer T that generates a table of contents for every *book*-document in the language defined by the DTD d. That is, for every chapter of the book, the transducer produces a list of its section titles. The transducer consists of the following rules:

$$\begin{array}{lcl}
(q, \text{book}) & \to & \text{book}(q) \\
(q, \text{chapter}) & \to & \text{chapter } q \\
(q, \text{title}) & \to & \text{title} \\
(q, \text{section}) & \to & q
\end{array}$$

This transducer T transforms the document in Figure 2.6 into the tree in Figure 2.7

```
<xsl:template match="a" mode ="p">
  <d>
     <e/>
  </d>
</xsl:template>

<xsl:template match="b" mode ="p">
  <d>
     <xsl:apply-templates mode="q"/>
  </d>
</xsl:template>

<xsl:template match="a" mode ="q">
  <c/>
  <xsl:apply-templates mode="p"/>
</xsl:template>

<xsl:template match="b" mode ="q">
  <c>
      <xsl:apply-templates mode="p"/>
      <xsl:apply-templates mode="q"/>
  </c>
</xsl:template>
```

Figure 2.5: The XSLT program equivalent to the transducer of Example 2.13.

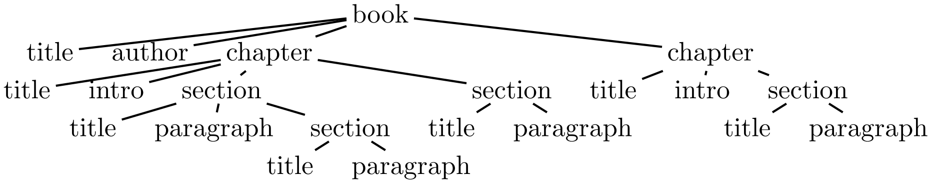


Figure 2.6: A document conforming to the schema of Example 2.15.

The example illustrates the usefulness of deleting states. Indeed, deleting states allow the transducer to skip all intermediate *section*-nodes. Furthermore, the rule

$$(q, \text{chapter}) \to \text{chapter } q$$

allows to list all section titles next to the *chapter*-element rather than below.

Next, we illustrate copying. The following transducer T' extends the previous one by adding a summary of the book to the table of contents. The summary is given by listing the title and introduction of each chapter. By using the two states p and p', we make sure that the title of the book is not printed in the summary. The transducer

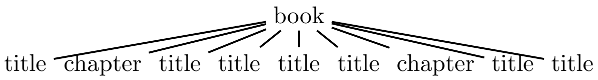


Figure 2.7: Output of the transducer T of Example 2.15, when executed on the tree in Figure 2.6.

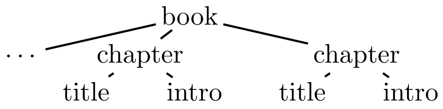


Figure 2.8: Output of the transducer T' of Example 2.15, when executed on the tree in Figure 2.6. The part of the output that is already depicted in Figure 2.7 is replaced by dots.

T' consists of the following rules:

$$\begin{array}{lcl}
(q, \text{book}) & \rightarrow & \text{book}(q\ p) \\
(q, \text{chapter}) & \rightarrow & \text{chapter}\ q \\
(q, \text{title}) & \rightarrow & \text{title} \\
(q, \text{section}) & \rightarrow & q \\
(p, \text{chapter}) & \rightarrow & \text{chapter}(p') \\
(p', \text{title}) & \rightarrow & \text{title} \\
(p', \text{intro}) & \rightarrow & \text{intro}
\end{array}$$

The output of the transformation, applied to the document in Figure 2.6 is the tree in Figure 2.8. Here, we replaced the part of the output that is also generated by the transformation T (which is depicted on Figure 2.7) with dots. ◇

It turns out that the copying and the deletion power of a tree transducer has an important effect on the complexity of the typechecking problem. We now define these classes of transducers more formally. A transducer is *deleting* if its set of rewrite rules contains a rhs with a state at its top-level. A transducer is *non-deleting* if it is not deleting. We denote by $\mathcal{T}_{\text{nd}}$ the class of non-deleting transducers and by $\mathcal{T}_{\text{d}}$ the class of transducers where we allow deletion. Hence, $\mathcal{T}_{\text{nd}} \subseteq \mathcal{T}_{\text{d}}$. Furthermore, a transducer T has *copying width* C if there are at most C occurrences of states in every sequence of siblings in an rhs. For instance, the transducer in Example 2.13 has copying width 2. Given a natural number C, we denote by $\mathcal{T}_{\text{bc}}^{C}$ the class of transducers of copying width C. Usually, we will leave C implicit. The abbreviation "bc" stands for *bounded copying*. We denote intersections of these classes by combining the indexes. For instance, $\mathcal{T}_{\text{nd,bc}}$ is the class of non-deleting transducers with bounded copying. When we want to emphasize that we also allow *unbounded copying* in a certain application, we write, for instance, $\mathcal{T}_{\text{nd,uc}}$ instead of $\mathcal{T}_{\text{nd}}$.

2.4 The Typechecking Problem

We are now ready to define the problem that is central to this first part of the thesis.

Definition 2.16. A tree transducer T *typechecks* with respect to to an input tree language S_{in} and an output tree language S_{out}, if $T(t) \in S_{\text{out}}$ for every $t \in S_{\text{in}}$. ◇

Example 2.17. The second transducer of Example 2.15 typechecks with respect to the input schema and the following DTD:

$$\begin{array}{lcl} \text{book} & \to & \text{title (chapter title}^*)^* \text{ chapter}^* \\ \text{chapter} & \to & \text{(title intro)} + \varepsilon \end{array}$$ ◇

Definition 2.18. Given S_{in}, S_{out}, and T, the *typechecking problem* consists in verifying whether T typechecks with respect to S_{in} and S_{out}. ◇

We parameterize the typechecking problem by the kind of tree transducers and tree languages we allow. Let $\mathcal{T}$ be a class of transducers and $\mathcal{S}$ be a representation of a class of tree languages. Then $\text{TC}[\mathcal{T}, \mathcal{S}]$ denotes the typechecking problem where $T \in \mathcal{T}$ and $S_{\text{in}}, S_{\text{out}} \in \mathcal{S}$. Examples of classes of tree languages are those defined by tree automata or DTDs. Classes of transducers are discussed in Section 2.3. The complexity of the typechecking problem is measured in terms of the sum of the sizes of the input and output schemas S_{in} and S_{out} and the transducer T.

2.5 Related Work

The research on typechecking XML transformations was initiated by Milo, Suciu, and Vianu [MSV03]. They obtained the decidability for typechecking of transformations realized by k-pebble transducers via a reduction to satisfiability of monadic second-order logic. Unfortunately, in this general setting, the latter non-elementary algorithm cannot be improved [MSV03]. Alon et al. [AMN+03a, AMN+03b] investigated typechecking in the presence of data values and show that the problem quickly turns undecidable. As our interest mainly lies in formalisms with a more manageable complexity for the typechecking problem, we choose to work with XML transformations that are much less expressive than k-pebble transducers and that do not change or use data values in the process of transformation.

Although the structure of XML documents can be faithfully represented by unranked trees (these are trees without a bound on the number of children of nodes), Milo, Suciu, and Vianu chose to study k-pebble transducers over binary trees as there is an immediate encoding of unranked trees into binary ones. We illustrate such an encoding in Figure 2.9 and explain it in more detail later in this section. The top-down variants of such tree transducers are well-studied on binary trees [GS97]. However, these results do not aid in the quest to precisely characterize the complexity of typechecking transformations on unranked trees. Indeed, as we show later in this section, the class of unranked tree transductions can *not* be captured by ordinary transducers working on the binary encodings. Macro tree transducers can simulate our transducers on the binary encodings [MN00, EV85], but as the complexity of their typechecking problem is rather high [MPBS05], this observation is not of much help. For these reasons, we chose to work directly with unranked tree transducers.

A problem related to typechecking is *type inference* [MS99b, PV00]. This problem consists in constructing a tight output schema, given an input schema and a transformation. Of course, solving the type inference problem implies a solution for the

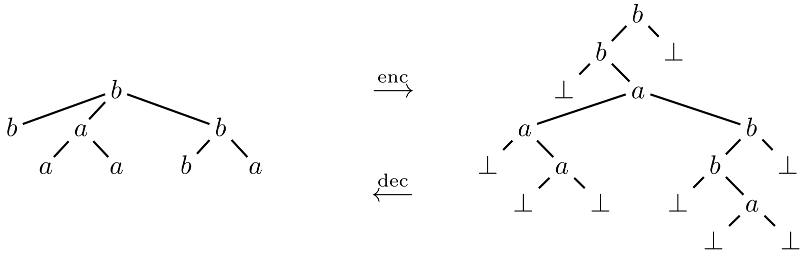


Figure 2.9: An unranked tree and its binary encoding.

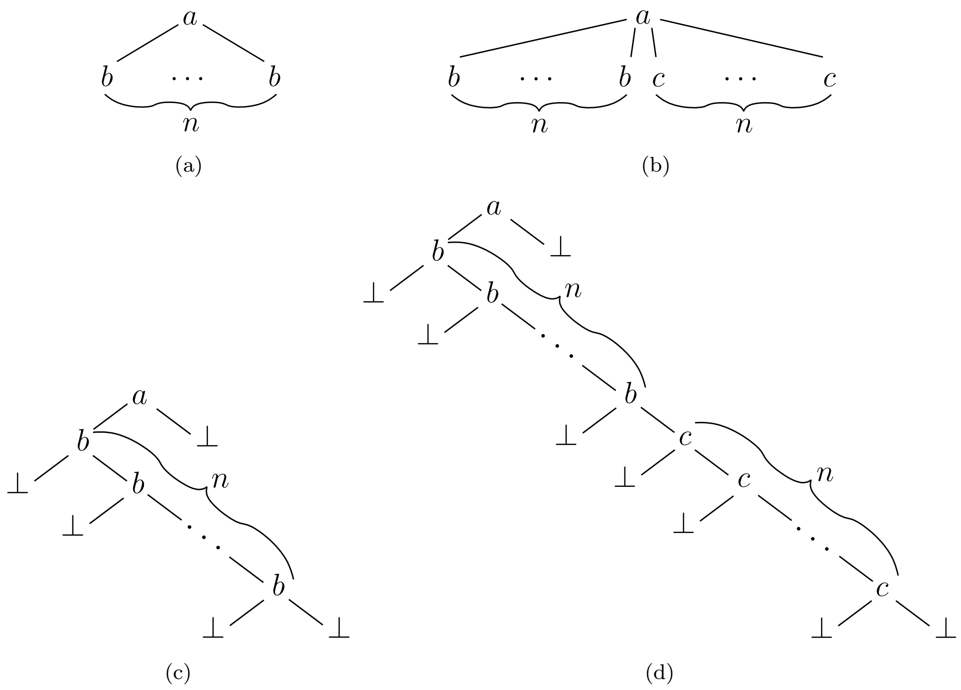
typechecking problem: check containment of the inferred schema into the given one. However, characterizing output languages of transformations is quite hard [PV00]. For this reason, we adopt different techniques for obtaining complexity upper bounds for the typechecking problem.

The tree transformations considered in the dissertation are restricted versions of the DTL-programs, studied by Maneth and Neven [MN00]. They already obtained a non-elementary upper bound on the complexity of typechecking (due to the use of monadic second-order logic in the definition of the transducers). Recently, Maneth et al. considered the typechecking problem for an extension of DTL-programs and obtained that typechecking was still decidable [MPBS05]. Their typechecking algorithm, like the one of [MSV03], is based on *inverse* type inference. That is, they compute the pre-image of all ill-formed output documents and test whether the intersection of the pre-image and the input schema is empty. Tozawa considered typechecking with respect to tree automata for a fragment of top-down XSLT [Toz01]. He uses a more general framework, but he was not able to derive a bound better than double-exponential on the complexity of his algorithm.

Ranked versus Unranked Tree Transducers. We briefly motivate why we use unranked transducers rather than their more deeply studied ranked counterparts.

As we mentioned before, it is known that unranked trees can be uniformly encoded as binary trees. The most widely known encoding is the so-called *first-child next-sibling encoding*, which we illustrate in Figure 2.9. The name of the encoding stems from the fact that it essentially views unranked trees as binary trees over their first-child and next-sibling relations. The encoding is denoted by *enc* and the decoding by *dec*. Intuitively, the first child of a node remains the first child of that node in the encoding, but it is explicitly encoded as a left child. The remaining children are right descendants of the first child. When a first child or next sibling of a node does not exist, a special dummy symbol ⊥ is inserted.

However, we argue that our unranked tree transducers cannot be simulated by deterministic top-down ranked tree transducers on binary trees using this standard encoding. A formal definition of deterministic top-down ranked tree transducers can be found in [GS97]. In Figures 2.10(a) and 2.10(b), we show two tree languages (n is arbitrary). Their binary encodings are depicted in Figures 2.10(c) and 2.10(d). Let L_1, L_2, L_3 and L_4 be the tree languages represented by the trees in Figure 2.10(a), 2.10(b), 2.10(c) and 2.10(d), respectively.



(a) (b) (c) (d)

Figure 2.10: 2.10(a) and 2.10(b) are unranked trees. 2.10(c) and 2.10(d) are their binary encodings respectively.

The language L_1 can be transformed to L_2 by the tree transducer $T = (Q, \Sigma, q^0, R)$ where $Q = \{q^0, q^b, q^c\}$, $\Sigma = \{a, b, c\}$, and R contains the following rules:

$$\begin{array}{lcl} (q^0, a) & \to & a(q^b q^c) \\ (q^b, b) & \to & b \\ (q^c, b) & \to & c \end{array}$$

Basically this tree transducer transforms the string b^n of children of the root in the input to the string $b^n c^n$ of children of the root in the output tree. However, as we argue next, L_3 cannot be transformed to L_4 by a deterministic top-down ranked tree transducer. For a tree t, let $\mathrm{path}(t)$ be the set of all strings formed by concatenating the labels of the nodes on a path in t from the root to a leaf. For a tree language L, define the string language $\mathrm{path}(L) = \{\mathrm{path}(t) \mid t \in L\}$. Given a regular tree language L and a deterministic top-down ranked tree transducer R, it is known that the language $\mathrm{path}(R(L))$, where $R(L) = \{R(t) \mid t \in L\}$, is regular (Corollary 20.13 in [GS97]). Since $\mathrm{path}(L_4) = \{ab^k c^\ell \bot \mid k, \ell \in \mathbb{N}, \ell \leq k\}$ is not regular and L_3 is a regular tree language, L_4 cannot be the result of applying a deterministic top-down ranked tree transducer to L_3.

3

Basic Complexity Results

In this chapter we discuss some basic results that are needed in proofs in the first part of the thesis. These results mainly consist of the complexities of basic decision problems for logics and for finite automata and should be seen as a set of *tools* that we use in the following chapters. We start with some results for SL.

3.1 Logic

For an SL-formula ϕ, we say that two strings w_1 and w_2 are *ϕ-equivalent* (denoted $w_1 \equiv_\phi w_2$) if $w_1 \models \phi$ if and only if $w_2 \models \phi$.

For $a \in \Sigma$ and $w \in \Sigma^*$, we denote by $\#_a(w)$ the number of occurrences of a in w. The following Lemma is a useful tool in proving results about SL (see Definition 2.5).

Lemma 3.1. *Let ϕ be an SL-formula and let k be the largest integer occurring in ϕ. For every $w, w' \in \Sigma^*$, if, for every $a \in \Sigma$, one of the following holds:*

- *$\#_a(w') > k$ and $\#_a(w) > k$, or*
- *$\#_a(w') = \#_a(w)$,*

then $w \equiv_\phi w'$.

Proof. We can assume that negations in ϕ only occur in front of atomic SL-formulas. We call an atomic SL-formula or a negation of an atomic SL-formula a *literal*.

To prove the lemma, simply observe that, for each $a \in \Sigma$ such that $\#_a(w) > k$, w satisfies all literals of the form $a^{\geq i}$ and $\neg a^{=j}$ and w violates all literals of the form $\neg a^{\geq i}$ and $a^{=j}$ where $i, j \in \{0, \ldots, k\}$. The same holds for w'. □

The following lemma is slightly more complicated. We make use of the lemma in the proof of Theorem 4.3(5). Intuitively, the functions f in the lemma are used to model the effect of a (nondeleting) tree transducer on a string of siblings in a tree.

Lemma 3.2. *Let ϕ_1 and ϕ_2 be SL-formulas and let k be the largest integer occurring in ϕ_1 or ϕ_2. Let $f : \Sigma^* \to \Sigma^*$ be a function such that, for every $b \in \Sigma$, there exists a fixed sequence of natural numbers $c^b, (c^b_a)_{a\in\Sigma}$ for which $\#_b(f(s)) = c^b + \sum_{a\in\Sigma}(c^b_a \times \#_a(s))$ for every $s \in \Sigma^*$. If there is a string $s \models \phi_1$ then there is a string $s' \in \Sigma^*$ such that*

- *$s' \models \phi_1$*
- *$f(s') \models \phi_2$ if and only if $f(s) \models \phi_2$, and*
- *each symbol occurs maximally $k + 1$ times in s'.*

Proof. Intuitively, the function f characterizes the effect of our tree transformations on a string of siblings in the input tree. Let s be a string such that $s \models \phi_1$. We construct s' from s by deleting x arbitrary occurrences of every symbol a that occurs $k + 1 + x$ times in s. So, $k + 1$ occurrences remain. Since by Lemma 3.1, $s' \models \phi_1$, we only need to show that $f(s') \models \phi_2$ if and only if $f(s) \models \phi_2$. Therefore, take an arbitrary symbol $b \in \Sigma$. Then $\#_b(f(s)) = c^b + \sum_{a\in\Sigma}(c^b_a \times \#_a(s))$. If, for every $a \in \Sigma$ that occurs more than k times in s, $c^b_a = 0$, then $\#_b(f(s)) = \#_b(f(s'))$. If this is not the case, take $a \in \Sigma$ that occurs more than k times in s and $c^b_a \neq 0$. Then $\#_b(f(s)) \geq \#_a(s) > k$ and $\#_b(f(s')) \geq \#_a(s') > k$, so, according to Lemma 3.1, $f(s) \models \phi_2$ if and only if $f(s') \models \phi_2$. □

The final goal of the present section is to prove that one can find in an integer solution to a Boolean combination of linear (in)equalities with a fixed number of variables in polynomial time. Of course, when the number of variables can be arbitrary, the problem is NP-complete because it subsumes the INTEGER PROGRAMMING problem (Problem MP1 in the book of Garey and Johnson [GJ79]). A restricted form of this problem (when the Boolean combination is a conjunction), is a well known theorem by Lenstra [Len83]. The final result is stated in Proposition 3.5, which we prove in a series of lemmas and propositions.

We start by revisiting a lemma due to Ferrante and Rackoff [FR75], since we need their construction in a subsequent proposition. Thereto, we need some definitions. We define logical formulas with variables $x_1, x_2, \ldots$ and linear equations with factors in $\mathbb{Q}$, the set of rationals. A *linear term* is an expression of the form a_1/b_1, $a_1/b_1x_1 + \cdots + a_n/b_nx_n$, or $a_1/b_1x_1 + \cdots + a_{n-1}/b_{n-1}x_{n-1} + a_n/b_n$ where $a_i, b_i \in \mathbb{N}$ for $i = 1, \ldots, n$. An *atomic linear formula* is either the string "true", the string "false", or a formula of the form $\vartheta_1 = \vartheta_2$, $\vartheta_1 < \vartheta_2$, or $\vartheta_1 > \vartheta_2$. A *linear formula* Φ is built up from atomic formulas using conjunction, disjunction, negation, and the quantifier symbol $\exists$ in the usual manner. Formulas are interpreted in the obvious manner over $\mathbb{Q}$. For instance, the formula $\neg\exists x_1, x_2\ (x_1 < x_2) \wedge \neg\big(\exists x_3\ (x_1 < x_3 \wedge x_3 < x_2)\big)$ states that for every two different rational numbers, there exists a third rational number that lies strictly between them. As is standard in the literature, we write $\Phi(x_1, \ldots, x_n)$ to indicate that Φ is a linear formula with free variables $x_1, \ldots, x_n$.

The *size* of a linear formula Φ is the sum of the number of brackets, Boolean connectives, the sizes of the variables, and the sizes of all rational constants occurring in Φ. Here, we assume that all rational constants are written as a/b, where a and b are integers, written in binary notation. We assume that variables are written as x_i, where i is written in binary notation.

Lemma 3.3 (Lemma 1 in [FR75])**.** *Let $\Phi(x_1, \ldots, x_n)$ be a quantifier-free linear formula. Then there exists a* PTIME *procedure for obtaining another quantifier-free linear formula, $\Phi'(x_1, \ldots, x_{n-1})$, such that*

$$\Phi'(x_1, \ldots, x_{n-1}) \textit{ is equivalent to } \exists x_n \Phi(x_1, \ldots, x_n).$$

Proof. Let $\Phi(x_1, \ldots, x_n)$ be a quantifier-free linear formula.

Step 1: *Solve for x_n* in each atomic linear formula of Φ. That is, obtain a quantifier-free linear formula, $\Psi^n(x_1, \ldots, x_n)$, such that every atomic linear formula of Ψ^n either does not involve x_n or is of the form *(i)* $x_n < \vartheta$, *(ii)* $x_n > \vartheta$, or *(iii)* $x_n = \vartheta$, where ϑ is a term not involving x_n.

Step 2: We now make the following definitions:
Given $\Psi^n(x_1, \ldots, x_n)$, to get $\Psi^n_{-\infty}(x_1, \ldots, x_{n-1})$, respectively, $\Psi^n_{\infty}(x_1, \ldots, x_{n-1})$, replace

$x_n < \vartheta^n$ in Ψ by "true" (respectively, "false");
$x_n > \vartheta^n$ in Ψ by "false" (respectively, "true"); and,
$x_n = \vartheta^n$ in Ψ by "false" (respectively, "false").

The intuition is that, for any rational numbers $r_1, \ldots, r_{n-1}$, if r is a sufficiently small rational number, then $\Psi^n(r_1, \ldots, r_{n-1}, r)$ and $\Psi^n_{-\infty}(r_1, \ldots, r_{n-1})$ are equivalent. A similar statement can be made for Ψ^n_{∞} for r sufficiently large.

Step 3: We will now eliminate the quantifier from $\exists x_n \Psi(x_1, \ldots, x_n)$. Let U be the set of all terms ϑ (not involving x_n) such that $x_n > \vartheta$, $x_n < \vartheta$, or $x_n = \vartheta$ is an atomic linear formula of Ψ. Lemma 1.1 in [FR75] then shows that $\exists x_n \Psi(x_1, \ldots, x_n)$ is equivalent to the quantifier-free linear formula $\Phi'(x_1, \ldots, x_{n-1})$ defined as

$$\Psi^n_{-\infty} \vee \Psi^n_{\infty} \vee \bigvee_{\vartheta, \vartheta' \in U} \Psi^{n,\vartheta,\vartheta'},$$

where $\Psi^{n,\vartheta,\vartheta'} = \Psi^n\big(x_1, \ldots, x_{n-1}, \frac{\vartheta+\vartheta'}{2}\big)$. □

The following proposition is implicit in the work by Ferrante and Rackoff [FR75], we prove it for completeness:

Proposition 3.4. *Let $\Phi(x_1, \ldots, x_n)$ be a quantifier-free linear formula. If n is fixed, then satisfiability of Φ over $\mathbb{Q}$ can be decided in* PTIME*. Moreover, if Φ is satisfiable, we can find $(v_1, \ldots, v_n) \in \mathbb{Q}^n$ such that $\Phi(v_1, \ldots, v_n)$ is true in polynomial time.*

Proof. We first show that satisfiability can be decided in PTIME. To this end, we simply iterate over the three steps in the proof of Lemma 3.3 until we obtain a linear formula without variables. Hence, in each iteration, one variable x_i is eliminated from Φ. For every $i = 1, \dots, n$, let Φ^i be the linear formula obtained after eliminating variable x_i.

Notice that, in each iteration of the algorithm, the number of atomic linear formulas grows quadratically when going from Φ^i to Φ^{i-1}. However, as there are only a constant number of iterations, the number of atomic linear formulas in the resulting linear formula Φ^1 is still polynomial. Moreover, Ferrante and Rackoff show that the absolute value of every integer constant occurring in any rational constant in Φ^i is at most $(s_0)^{14^n}$, where s_0 is the largest absolute value of any integer constant occurring in any rational constant in Φ (cfr. page 73 in [FR75]). As n is a fixed number, we can decide whether Φ^1 is satisfiable in polynomial time.

Suppose that Φ is satisfiable. We now show how we can construct $(v_1, \dots, v_n) \in \mathbb{Q}^n$ in polynomial time such that $\Phi(v_1, \dots, v_n)$ is true. For a term ϑ using variables $x_1, \dots, x_i$, we denote by $\vartheta(v_1, \dots, v_{i-1})$ the rational number obtained by replacing the variables $x_1, \dots, x_i$ in ϑ by $v_1, \dots, v_{i-1}$ and evaluating the resulting expression.

For every $i = 1, \dots, n$, we construct v_i from Ψ^i, $\Psi^i_{-\infty}$, Ψ^i_{∞}, and $\Psi^{i,\vartheta,\vartheta'}$ (which are defined in the proof of Lemma 3.3) as follows:

(1) If $\Psi^{i,\vartheta,\vartheta'}$ is satisfiable, then $v_i = \frac{\vartheta(v_1,\dots,v_{i-1})+\vartheta'(v_1,\dots,v_{i-1})}{2}$.

(2) Otherwise, if Ψ^i_{∞} is satisfiable, then $v_i = \max\{\vartheta(v_1, \dots, v_{i-1}) \mid x_i < \vartheta$ or $x_i > \vartheta$ or $x_i = \vartheta$ is an atomic linear formula in $\Psi^i\} + 1$.

(3) Otherwise, if $\Psi^i_{-\infty}$ is satisfiable, then $v_i = \min\{\vartheta(v_1, \dots, v_{i-1}) \mid x_i < \vartheta$ or $x_i > \vartheta$ or $x_i = \vartheta$ is an atomic linear formula in $\Psi^i\} - 1$.

It remains to show that we can represent every v_i in a polynomial manner.

In the proof of Lemma 2 in [FR75], Ferrante and Rackoff show that, if w_i is the maximum absolute value of any integer occurring in the definition of $v_1, \dots, v_i$, then we have the recurrence

$$w_{i+1} \leq (s_0)^{2^{cn}} \cdot (w_i)^i,$$

for a constant c, and for s_0 defined as before. Let $c' = 2^{cn}$. Hence, the maximum number of bits needed to represent the largest integer in v_{i+1} is $\log\big((s_0)^{c'} \cdot (w_i)^i\big) = c' \log(s_0) \cdot i \log(w_i)$, which is polynomially larger than $\log(w_i)$, the number of bits needed to represent the largest integer in v_i.

As we only have a constant number of iterations, the number of bits needed to represent the largest integer occurring in the definition of v_n is also polynomial. □

We are now ready to prove Proposition 3.5. It generalizes the theorem by Lenstra which states that there exists a polynomial time algorithm to find an integer solution for a *conjunction* of linear (in)equalities with rational factors and a fixed number of variables [Len83].

Proposition 3.5. *There exists a* PTIME *algorithm that decides whether a Boolean combination of linear (in)equalities with rational factors and a fixed number of variables has an integer solution.*

Proof. Notice that we cannot simply put the Boolean combination into disjunctive normal form, as this would lead to an exponential increase of its size.

Let $\Phi(x_1, \ldots, x_n)$ be a Boolean combination of linear formulas $\varphi_1, \ldots, \varphi_m$ with variables $x_1, \ldots, x_n$ that range over $\mathbb{Z}$, the set of integers. Here, n is a constant integer greater than zero. Without loss of generality, we can assume that every φ_i is of the form

$$k_{i,1} \times x_1 + \cdots + k_{i,n} \times x_n + k_i \geq 0,$$

where $k_i, k_{i,1}, \ldots, k_{i,n} \in \mathbb{Q}$.

We describe a PTIME procedure for finding a solution for $x_1, \ldots, x_n$, that is, for finding values $v_1, \ldots, v_n \in \mathbb{Z}$ such that $\Phi(v_1, \ldots, v_n)$ evaluates to true.

First, we introduce some notation and terminology. For every $i = 1, \ldots, m$, we denote by φ'_i the linear formula $k_{i,1} \times x_1 + \cdots + k_{i,n} \times x_n + k_i = 0$. In the following, we freely identify φ'_i with the hyperplane it defines in $\mathbb{R}^n$. For an n-tuple $\overline{y} = (y_1, \ldots, y_n) \in \mathbb{Q}^n$, we denote by $\varphi'_i(\overline{y})$ the rational number $k_{i,1} \times y_1 + \cdots + k_{i,n} \times y_n + k_i$.

Given a set of hyperplanes H in $\mathbb{R}^n$, we say that $C \subseteq \mathbb{R}^n$ is a *cell* of H when

(i) for every hyperplane φ'_i in H, and for every pair of points $\overline{y}$, $\overline{z} \in C$, we have that $\varphi'_i(\overline{y})\ \theta\ 0$ if and only if $\varphi'_i(\overline{z})\ \theta\ 0$, where, θ denotes "<", ">", or "="; and

(ii) there exists no $C' \supsetneq C$ with property (i).

Let H be the set of hyperplanes $\{\varphi'_i \mid 1 \leq i \leq m\}$.

We now describe the PTIME algorithm. The algorithm iterates over the following steps:

(1) Compute $(v'_1, \ldots, v'_n) \in \mathbb{Q}^n$ such that $\Phi(v'_1, \ldots, v'_n)$ is true.[1] If no such $(v'_1, \ldots, v'_n)$ exists, the algorithm rejects.

(2) For every $\varphi'_i \in H$, let $\theta_i \in \{<, >, =\}$ be the relation such that

$$k_{i,1} \times v'_1 + \cdots + k_{i,n} \times v'_n + k_i\ \theta_i\ 0.$$

For every $i = 1, \ldots, m$, let $\varphi''_i = k_{i,1} \times x_1 + \cdots + k_{i,n} \times x_n + k_i\ \theta_i\ 0$. So, for every $i = 1, \ldots, m$, φ''_i defines the half-space or hyperplane that contains the point $(v'_1, \ldots, v'_n)$.

Let $\Phi'(x_1, \ldots, x_n)$ be the conjunction

$$\bigwedge_{1 \leq i \leq n} \varphi''_i.$$

Notice that the points satisfying $\Phi'(x_1, \ldots, x_n)$ are precisely the points in the cell C of H that contains $(v'_1, \ldots, v'_n)$.

(3) Solve the INTEGER PROGRAMMING problem for $\Phi'(x_1, \ldots, x_n)$. That is, find a $(v_1, \ldots, v_n) \in \mathbb{Z}^n$ such that $\Phi'(v_1, \ldots, v_n)$ evaluates to true.

(4) If $(v_1, \ldots, v_n) \in \mathbb{Z}^n$ exists, then write $(v_1, \ldots, v_n)$ to the output and accept.

[1] Note that we abuse notation here, as the variables in Φ range over $\mathbb{Z}$ and not $\mathbb{Q}$.

(5) If $(v_1, \ldots, v_n) \in \mathbb{Z}^n$ does not exist, then overwrite $\Phi(x_1, \ldots, x_n)$ with

$$\Phi''(x_1, \ldots, x_n) = \Phi(x_1, \ldots, x_n) \wedge \neg\Phi'(x_1, \ldots, x_n)$$

and go back to step (1).

We show that the algorithm is correct. Clearly, if the algorithm accepts, Φ has a solution. Conversely, suppose that Φ has a solution. Hence, the algorithm computes a value $(v'_1, \ldots, v'_n) \in \mathbb{Q}^n$ in step (1) of its first iteration. It follows from the following two observations that the algorithm accepts:

(i) If the algorithm computes $(v'_1, \ldots, v'_n) \in \mathbb{Q}^n$ in step (1), and the cell C of H containing $(v'_1, \ldots, v'_n)$ also contains a point in $\mathbb{Z}^n$, then step (3) finds a solution $(v_1, \ldots, v_n) \in \mathbb{Z}^n$; and,

(ii) If the algorithm computes $(v'_1, \ldots, v'_n) \in \mathbb{Q}^n$ in step (1), and the cell C of H containing $(v'_1, \ldots, v'_n)$ does *not* contain a point in $\mathbb{Z}^n$, then step (3) does *not* find a solution. By construction of Φ'' in step (5), the solutions to the linear formula Φ'' are the solutions of Φ, minus the points in C. As C did not contain a solution, we have that Φ has a solution if and only if Φ'' has a solution. Moreover, there exists no $(v''_1, \ldots, v''_n) \in C$ such that $\Phi''(v''_1, \ldots, v''_n)$ evaluates to true.

To show that the algorithm can be implemented to run in polynomial time, we first argue that there are at most a polynomial number of iterations. This follows from the observation in step (2) that the points satisfying $\Phi'(x_1, \ldots, x_n)$ are precisely all the points in a cell C of H. Indeed, when we do not find a solution to the problem in step (3), we adapt Φ to exclude all the points in cell C in step (5). Hence, in the following iteration, step (1) cannot find a solution in cell C anymore. It follows that the number of iterations is bounded by the number of cells in H, which is $\Theta(m^n)$ (see, for example, the work by Buck [Buc43], or Theorem 1.3 in the book by Edelsbrunner [Ede87] for a more recent reference).

Finally, we argue that every step of the algorithm can be computed in PTIME.

- Step (1) can be solved by the quantifier elimination method of Ferrante and Rackoff (Lemma 3.3). Proposition 3.4 states that we can find $(v'_1, \ldots, v'_n)$ in polynomial time.
- Step (2) is easily to be seen to be in PTIME: we only have to evaluate every φ'_i once on $(v'_1, \ldots, v'_n)$.
- Step (3) can be executed in PTIME by Lenstra's algorithm for INTEGER PROGRAMMING with a fixed number of variables [Len83].
- Step (4) is in PTIME (trivial).
- Step (5) replaces $\Phi(x_1, \ldots, x_n)$ by the linear formula

$$\Phi(x_1, \ldots, x_n) \wedge \neg\Phi'(x_1, \ldots, x_n).$$

As the size of $\Phi'(x_1, \dots, x_n)$ is bounded by n plus the sum of the sizes of φ_i'' for $i = 1, \dots, n$, the linear formula Φ only grows by a linear term in each iteration. As the number of iterations is bounded by a polynomial, the maximum size of Φ is also bounded by a polynomial.

It follows that the algoritm is correct, and can be implemented to run in polynomial time. □

Corollary 3.6. *There exists a* PTIME *algorithm that decides whether a Boolean combination of linear (in)equalities with rational factors and a fixed number of variables has a solution of positive integers.*

Proof. Given a Boolean combination $\Phi(x_1, \dots, x_n)$ of linear (in)equalities with rational factors, we simply apply the algorithm of Proposition 3.5 to the linear formula

$$\Phi'(x_1, \dots, x_n) = \Phi(x_1, \dots, x_n) \wedge \bigwedge_{1 \leq i \leq n} x_i \geq 0.$$ □

3.2 Binary Tree Automata

The remainder of this chapter focuses on decision problems for finite automata. We first introduce a little background on tree automata for *binary trees.* A *binary alphabet* or *binary signature* is a pair $(\Sigma, \mathrm{rank}_\Sigma)$, where rank_Σ is a function from Σ to $\{0, 1, 2\}$. The set of *binary* Σ*-trees* is the set of Σ-trees inducively defined as follows. When $\mathrm{rank}_\Sigma(a) = 0$, then a is a binary Σ-tree. When $\mathrm{rank}_\Sigma(a) = 1$ and t_1 is a binary Σ-tree, then $a(t_1)$ is a binary Σ-tree. When $\mathrm{rank}_\Sigma(a) = 2$ and t_1, t_2 are binary Σ-trees, then $a(t_1\ t_2)$ is a binary Σ-tree. We denote the set of binary Σ-trees by $b\mathcal{T}_\Sigma$.

Definition 3.7. A *nondeterministic binary (or traditional) tree automaton (NBTA)* for binary Σ-trees is an NTA $B = (Q, \Sigma, \delta, F)$ for which Σ is a binary alphabet. Moreover, the regular languages in the transition function $\delta : Q \times \Sigma \to 2^{Q^*}$ are always represented as a finite set of strings. For each $q \in Q$ and $a \in \Sigma$ with $\mathrm{rank}_\Sigma(a) = k$, $\delta(q, a)$ is a finite subset of Q^k. ◇

A binary tree automaton B is *bottom-up deterministic* when it is a bottom-up deterministic NTA. It is *top-down deterministic* if *(i)* F is a singleton and, *(ii)* for every $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ contains at most one string. It is *unambiguous* when, for every tree $t \in L(B)$, there exists at most one accepting run of B on t. We denote the classes of bottom-up deterministic, top-down deterministic, and unambiguous binary tree automata by DBTA, TDBTA, and UBTA, respectively.

There is a well-known connection between ranked tree automata and their more general unranked counterparts. This connection is based on *encodings* between binary and unranked trees, such as, for instance, the *first-child next-sibling encoding* that we already illustrated in Section 2.5. Let *enc* denote the first-child next-sibling encoding from unranked to ranked trees, and let *dec* denote the decoding. The connection between ranked and unranked tree automata can then be summarized in the following proposition:

Proposition 3.8 ([GKPS05, Nev02, Suc01]).

- *For every NTA(NFA) B there exists a NBTA A such that $L(A) = \{enc(t) \mid t \in L(B)\}$. Moreover, A can be computed in polynomial time.*

- *For every NBTA A there exists an NTA(NFA) B such that $L(B) = \{dec(t) \mid t \in L(A)\}$. Moreover, B can be computed in polynomial time.*

3.3 Decision Problems for Finite String and Tree Automata

The present section treats the complexity of the emptiness, universality, inclusion, and intersection emptiness problems for finite automata, DTDs, ranked and unranked tree automata. Here, we say that a finite automaton A over strings, binary trees, or unranked trees is *universal* if $\Sigma^* \subseteq L(A)$, $b\mathcal{T}_\Sigma \subseteq L(A)$, or $\mathcal{T}_\Sigma \subseteq L(A)$, respectively.

The problems of interest to us are then defined as follows.

EMPTINESS: Given a finite automaton, DTD, ranked or unranked tree automaton A, is $L(A) = \emptyset$?

UNIVERSALITY: Given a finite automaton, DTD, ranked or unranked tree automaton A, is A universal?

INCLUSION: Given finite automata, DTDs, ranked or unranked tree automata A_1 and A_2, is $L(A_1) \subseteq L(A_2)$?

INTERSECTION EMPTINESS: Given the finite automata, DTDs, ranked or unranked tree automata $A_1, \ldots, A_n$, is $L(A_1) \cap \cdots \cap L(A_n) = \emptyset$?

We first give a brief overview of the well-known results about the decision problems that are of interest to us:

Proposition 3.9.

(1) EMPTINESS *for NFAs is* NLOGSPACE-*complete [Joh90].*

(2) UNIVERSALITY *for NFAs is* PSPACE-*complete [SM73].*

(3) INCLUSION *for NFAs and for REs is* PSPACE-*complete [SM73].*

(4) INCLUSION *for NBTAs is* EXPTIME-*complete [Sei90].*

(5) INTERSECTION EMPTINESS *for DFAs and for REs is* PSPACE-*complete [Koz77, GJ79].*

(6) INTERSECTION EMPTINESS *for TDBTAs is* EXPTIME-*complete [Sei94].*

We immediately obtain the following corollary to Proposition 3.9 and Proposition 3.8(4):

Corollary 3.10. INCLUSION *for NTA(NFA)s is* EXPTIME-*complete.*

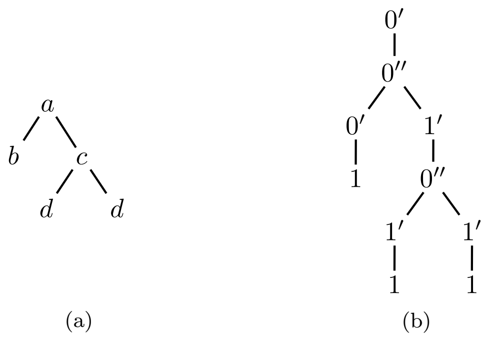


Figure 3.1: A tree t (left) and its encoding bin-enc(t) to an alphabet of size four (right).

We now focus on showing some complexity lower bounds for the automata in Proposition 3.9, even when they use a fixed size alphabet. To this end, we associate to each label $a \in \Sigma$ a unique binary string bin-enc$(a) \in \{0,1\}^*$ of length $\lceil \log |\Sigma| \rceil$. For a string $s = a_1 \cdots a_n$, define bin-enc(s) = bin-enc$(a_1) \cdots$ bin-enc(a_n). This encoding can be extended to string languages in the obvious way: for a string language $L \subseteq \Sigma^*$, we define bin-enc(L) to be the set $\{$bin-enc$(s) \mid s \in L\}$.

We show how to extend the encoding *bin-enc* to binary trees over alphabet $\Sigma_{\text{fix}} = \{0, 1, 0', 1', 0'', 1''\}$. Here, $\{0,1\}$, $\{0',1'\}$, and $\{0'',1''\}$ are the symbols with rank zero, one, and two, respectively. Let bin-enc$(a) = b_1 \cdots b_k$ for $a \in \Sigma$. Then we denote by tree-enc(a)

- the unary tree $b'_1(b'_2(\cdots(b'_{k-1}(b''_k))$ if rank$_\Sigma(a) = 2$;
- the unary tree $b'_1(b'_2(\cdots(b'_{k-1}(b'_k))$ if rank$_\Sigma(a) = 1$; or,
- the unary tree $b'_1(b'_2(\cdots(b'_{k-1}(b_k))$, otherwise.

Then, the bin-enc-fuction can be extended to binary trees as follows: for $t = a(t_1 \cdots t_n)$,

$$\text{bin-enc}(t) = \text{tree-enc}(a)(\text{bin-enc}(t_1) \cdots \text{bin-enc}(t_n)).$$

Note that we abuse notation here. The hedge bin-enc$(t_1) \cdots$ bin-enc(t_n) is intended to be the child of the leaf in tree-enc(a).

Example 3.11. We illustrate an example of this encoding in Figure 3.1. Here, we assume that bin-enc$(a) = 00$, bin-enc$(b) = 01$, bin-enc$(c) = 10$, and bin-enc$(d) = 11$. $\diamond$

The encoding can be extended to tree languages in the obvious way: for a tree language $L \subseteq b\mathcal{T}_\Sigma$, we define bin-enc$(L)$ to be the set $\{$bin-enc$(s) \mid s \in L\}$.

Proposition 3.12. *Let B be a TDBTA. Then there is a TDBTA B' over the alphabet $\{0, 1, 0', 1', 0'', 1''\}$ such that $L(B') =$ bin-enc$(L(B))$. Moreover, B' can be constructed from B using logarithmic space*

Proof. Let $B = (Q_B, \Sigma_B, \delta_B, F_B)$ be a TDBTA. Let $k := \lceil \log |\Sigma_B| \rceil$. We define $B' = (Q_{B'}, \{0, 1, 0', 1', 0'', 1''\}, \delta_{B'}, F_{B'})$. Set $Q_{B'} = \{q_x \mid q \in Q_B$ and x is a prefix of bin-enc(a), where $a \in \Sigma_B\}$ and $F_{B'} = \{q_\varepsilon \mid q \in F_B\}$. To define the transition function, we introduce some notation. For each $a \in \Sigma$ and $i, j = 1, \ldots, \lceil \log |\Sigma_B| \rceil$, denote by $a[i:j]$ the substring of bin-enc(a) from position i to position j (we abbreviate $a[i:i]$ by $a[i]$). For each transition $\delta_B(q, a) = q^1 q^2$, add the transitions $\delta_{B'}(q_\varepsilon, a[1]) = q_{a[1]}$, $\delta_{B'}(q_{a[1]}, a[2]) = q_{a[1:2]}, \ldots, \delta_{B'}(q_{a[1:k-1]}, a[k]) = q^1_\varepsilon q^2_\varepsilon$. Other transitions are defined analogously. Clearly, B' is a TDBTA, $L(B') = \text{bin-enc}(L(B))$, and B' can be constructed from B using logarithmic space. □

It is straightforward to show that Proposition 3.12 also holds for NFAs and DFAs (the proofs are completely analogous, as NFAs can also be seen as NBTAs on unary trees). It is immediate from Proposition 3.12, that the lower bounds of decision problems for automata over arbitrary alphabets in Proposition 3.9 carry over to automata working over fixed alphabets.

Hence, we obtain the following corollary to Proposition 3.12:

Corollary 3.13. *Over the alphabet* $\{0, 1\}$*, the following statements hold:*

(1) EMPTINESS *for NFAs is* NLOGSPACE*-complete.*

(2) UNIVERSALITY *for NFAs is* PSPACE*-complete.*

(3) INCLUSION *for NFAs is* PSPACE*-complete.*

(4) INTERSECTION EMPTINESS *for DFAs is* PSPACE*-complete.*

Over the alphabet $\{0, 1, 0', 1', 0'', 1''\}$*, the following statement holds:*

(5) INTERSECTION EMPTINESS *for TDBTAs is* EXPTIME*-complete.*

We say that an NFA $N = (Q_N, \Sigma, \delta_N, I_N, F_N)$ has *degree of nondeterminism 2* if *(i)* I_N has at most two elements and *(ii)* for every $q \in Q_N$ and $a \in \Sigma$, the set $\delta_N(q, a)$ has at most two elements. From the NLOGSPACE-completess of the reachability problem on graphs with out-degree 2 [Joh90], it now immediately follows that the emptiness problem for such NFAs over a fixed alphabet is also NLOGSPACE-complete:

Lemma 3.14. EMPTINESS *for NFAs with alphabet* $\{0, 1\}$ *degree of nondeterminism 2 is* NLOGSPACE*-complete.*

The following lemma treats the intersection emptiness problem for DFAs over a one-letter alphabet.

Lemma 3.15. INTERSECTION EMPTINESS *for DFAs over one-letter alphabet* $\{0\}$ *is* coNP*-hard.*

Proof. We reduce the satisfiability problem for Boolean formulas in 3-CNF to the complement of the intersection emptiness problem. The technique is an adaptation of the proof of Meyer and Stockmeyer establishing that inequivalence of regular expressions over a one-letter alphabet is NP-hard [SM73].

Let $\Phi = \phi_1 \wedge \cdots \wedge \phi_k$ be a formula in 3-CNF with variables $\{x_1, \ldots, x_n\}$. Let $p_1, \ldots, p_n$ be the first n primes. Note that, due to the Prime Number Theorem,[2] we only need to check values up to $\mathcal{O}(n \log n)$ for primality, when n grows large. Intuitively, we can represent every truth assignment of Φ with a string 0^r by assigning `true` to each x_i if $r \mod p_i = 0$ and `false` otherwise. We now construct a DFA A_i for each ϕ_i such that $\cap_{i=1}^{k} L(A_1) \neq \emptyset$ if and only if Φ is satisfiable.

We illustrate the construction of the A_i's by means of an example. Let $\phi_i = (x_1 \vee \neg x_2 \vee x_3)$ be a clause in Φ. Then $L(A_i) = (0^{p_1})^* + \overline{(0^{p_2})^*} + (0^{p_3})^*$. Hence, A_i accepts all strings that satisfy ϕ_i. Note that, since $(0^{p_j})^*$ or its complement can be easily represented by a DFA and since we only take unions of three automata, each A_i has $\mathcal{O}(n^{3 \cdot 2})$ states.

Finally, it is easy to see that a string $w \in \cap_{i=1}^{k} L(A_1)$ if and only if w encodes a truth assignment that satisfies Φ. Clearly, the reduction can be carried out in LOGSPACE. □

In the following proposition, we treat the emptiness problem for DTDs: given a DTD d, is $L(d) = \emptyset$? Note that $L(d)$ can be empty even when d is not. For instance, the trivial grammar $a \to a$ generates no finite trees.

Proposition 3.16. EMPTINESS *is*

(1) PTIME*-complete for DTD(NFA), DTD(DFA), and DTD(RE); and,*

(2) *co*NP*-complete for DTD(SL).*

Proof. (1) The upper bound follows from a (trivial) reduction to the emptiness problem for NTA(NFA)s, which is in PTIME (Proposition 3.18).

For the lower bound, we reduce in LOGSPACE from PATH SYSTEMS [Coo74], which is known to be PTIME-complete. PATH SYSTEMS is the decision problem defined as follows: given a finite set of propositions P, a set $A \subseteq P$ of axioms, a set $R \subseteq P \times P \times P$ of inference rules and some $p \in P$, is p *provable from A using* R? Here, *(i)* every proposition in A is provable from A using R and, *(ii)* if $(p_1, p_2, p_3) \in R$ and if p_1 and p_2 are provable from A using R, then p_3 is also provable from A using R.

In our reduction, we construct a DTD (Σ, d, p) such that

$$(\Sigma, d, p) \text{ is not empty if and only if } p \text{ is provable from } A \text{ using } R.$$

Concretely, we define Σ to be the set P. Furthermore, for every $(a, b, c) \in R$, we add the string ab to $d(c)$; and, for every $a \in A$, we set $d(a) = \{\varepsilon\}$. Clearly, (Σ, d, p) satisfies the requirements and that every language $d(a)$ can be represented by a sufficiently small DFA or RE.

(2) We provide an NP algorithm to check whether a DTD(SL) (Σ, d, r) defines a non-empty language. Intuitively, the algorithm computes the set $S = \{a \in \Sigma \mid L((\Sigma, d, a)) \neq \emptyset\}$ in an iterative manner and accepts when $r \in S$.

[2] The Prime Number Theorem states that $\lim_{n \to \infty} \frac{\pi(n)}{n / \ln n} = 1$, where $\pi(n)$ denotes the number of primes lesser than or equal to n. A corollary is that the n-th prime can be approximated by $n \log n$, up to a constant factor, when n grows large. A simple proof of the Prime Number Theorem has been published by Newman [New80].

Let k be the largest integer occurring in any SL-formula in d. Initially, S is empty.

The iterative step is as follows. Guess a sequence of different symbols $b_1, \ldots, b_m$ in S. Then guess a vector $(v_1, \ldots, v_m) \in \{0, \ldots, k+1\}^m$, where k is the largest integer occurring in any SL-formula in d. Intuitively, the vector $(v_1, \ldots, v_m)$ represents the string $b_1^{v_1} \cdots b_m^{v_m}$. From Lemma 3.1 it follows that any SL-formula in d is satisfiable if and only if it is satisfiable by a string of the form $a_1^{u_1} \cdots a_n^{u_n}$, where $\Sigma = \{a_1, \ldots, a_n\}$, and for all $i = 1, \ldots, n$, $u_i \in \{0, \ldots, k+1\}$. Now add to S each $a \in \Sigma$ for which $b_1^{v_1} \cdots b_m^{v_m} \models d(a)$. Note that this condition can be checked in PTIME. Repeat the iterative step at most $|\Sigma|$ times and accept when $r \in S$.

The coNP-lowerbound follows from an easy reduction of non-satisfiability. Let ϕ be a propositional formula with variables $x_1, \ldots, x_n$. Let Σ be the set $\{a_1, \ldots, a_n\}$. Let (Σ, d, r) be the DTD where $d(r) = \phi'$, where ϕ' is the formula ϕ in which every x_i is replaced by $a_i^{=1}$. Hence, (Σ, d, r) defines the empty tree language if and only if ϕ is unsatisfiable. □

Reducing a DTD is the act of finding an equivalent reduced DTD.[3]

Corollary 3.17.

(1) Reducing a DTD(NFA) is PTIME*-complete;*

(2) reducing a DTD(RE) is PTIME*-complete; and*

(3) reducing a DTD(SL) is NP*-complete.*

Proof. We first show the upper bounds. Let (Σ, d, s) be a DTD(NFA) or DTD(SL) over alphabet Σ. In both cases, the algorithm performs the following steps for each $a \in \Sigma$:

(i) Test whether a is *reachable* from s. That is, test whether there is a sequence of Σ-symbols $a_1, \ldots, a_n$ such that

- $a = s$ and $a_n = a$; and
- for every $i = 2, \ldots, n$, there exists a string $w_1 a_i w_2 \in d(a_{i-1})$, for $w_1, w_2 \in \Sigma^*$.

(ii) Test whether $L((\Sigma, d, a)) \neq \emptyset$.

Symbols that do not pass test (i) and (ii) are deleted from the alphabet Σ of the DTD. Let c be such a deleted symbol. In the case of SL, every atom $c^{\geq i}$ and $c^{=i}$ is replaced by true when $i = 0$ and false otherwise. In the case of NFAs, every transition mentioning c is removed and in the case of REs, every symbol c is replaced by $\emptyset$.

In the case of a DTD(NFA) and DTD(RE), step (i) is in NLOGSPACE and step (ii) is in PTIME. In the case of a DTD(SL), both tests (i) and (ii) are in NP.

For the lower bound, we argue that

(1) if there exists an NLOGSPACE-algorithm for reducing a DTD(NFA) or DTD(RE), then EMPTINESS for DTD(NFA)s and DTD(RE)s is in NLOGSPACE; and,

[3] Recall the definition of a reduced DTD from page 14.

```
$R_1 := \{q \in Q \mid \exists a \in \Sigma, \varepsilon \in \delta(q,a)\}$;
**for** $i := 2$ to $|Q|$ **do**
    $R_i := \{q \in Q \mid \exists a \in \Sigma, \delta(q,a) \cap R_{i-1}^* \neq \emptyset\}$;
**end for**
$R := R_{|Q|}$;
```

Figure 3.2: Computing the set R of reachable states.

(2) if there exists a PTIME-algorithm for reducing a DTD(SL), then EMPTINESS for a DTD(SL) is in PTIME.

Statements (1) and (2) are easy to show: one only has to observe that an emptiness test of a DTD can be obtained by reducing the DTD and verifying whether the alphabet of the resulting DTD still contains the start symbol. □

The following proposition is a very useful tool for obtaining upper bounds on the complexity of the typechecking problem.

Proposition 3.18.

(1) EMPTINESS *for NTA(NFA)s is in* PTIME.

(2) Deciding whether an NTA(NFA) defines a finite language is in PTIME.

(3) For a NTA(NFA) N, we can generate a description of a tree $t \in L(N)$ in PTIME.

Proof. (1) Let $B = (Q, \Sigma, \delta, F)$ be an NTA(NFA). The algorithm in Figure 3.2 computes the set of reachable states $R := \{q \mid \exists t \in \mathcal{T}_\Sigma : q \in \delta^*(t)\}$ in a bottom-up manner. Clearly, $L(B) = \emptyset$ if and only if $R \cap F = \emptyset$. Note that $R_i \subseteq R_{i+1}$ and $R_1 = \{\delta^*(a) \mid a \in \Sigma\}$. We argue that the algorithm is in PTIME. Clearly, R_1 can be computed in PTIME. Further, the for-loop makes a linear number of iterations. Every iteration is a linear number of non-emptiness tests of the intersection of an NFA with R_{i-1}^* where $R_{i-1} \subseteq Q$. As EMPTINESS for NFAs is in NLOGSPACE (Lemma 3.9), the latter is in PTIME.

(2) This immediately follows from (1) and results in [CDG$^+$01]. Indeed, an efficient way to test for finiteness is to check the existence of a loop. A language is infinite if and only if there is a loop on some useful state, that is, some state that can be used in an accepting run on some tree. The set of useful states is the subset of R (as computed in (1)) of states which are reachable from a state in F.

(3) This is an easy adaptation of the emptiness algorithm in part (1). Indeed, for every computed state $q \in R_i$ where $i > 1$, we can remember the witnesses symbol $a \in \Sigma$ and the string $w \in R_{i-1}^* \cap \delta(q,a)$. Using these witnesses, a Directed-Acyclic-Graph-representation of the counterexample tree t can easily be computed in a top-down manner, starting from an accepting state in $R_{|Q|}$. □

We say that a tree automaton is *complete* when, for every $a \in \Sigma$, we have $\bigcup_{q \in Q} \delta(q,a) = Q^*$. We denote the set of bottom-up deterministic complete tree automata by DTA^c.

Lemma 3.19. EMPTINESS *for DTA*^{c}*(DFA)s is* PTIME*-complete.*

Proof. The upper bound is immediate from Proposition 3.18(1).

The lower bound is obtained analogously as in the lower bound proof in Proposition 3.16(1). Given an instance P, A and R of PATH SYSTEMS, we construct a DTA^c(DFA) $A = (\Sigma \cup \{q_{\text{error}}\}, \Sigma, \delta, \Sigma)$ such that $L(A)$ is empty if and only if p is provable. In particular, for every $(a, b, c) \in R$, we add the string ab to $\delta(c, c)$; for every $a \in A$, $\delta(a, a) = \{\varepsilon\}$. Further, for every $a \in \Sigma$ we define $\delta(q_{\text{error}}, a)$ as $(\Sigma \cup \{q_{\text{error}}\})^* - L(\delta(a, a))$. Clearly, (d, p) satisfies the requirements. □

3.4 Alternating String Automata

We discuss two-way alternating string automata [LLS84]. We devote a separate section for this discussion, as we introduce quite a bit of technical material which is not used elsewhere in the thesis. We use the result of the present section in the proof of Theorem 4.3(3).

To prevent automata falling off the input string, we use delimiters $\triangleright$ and $\triangleleft$ not occurring in Σ. By $\Sigma_{\triangleright\triangleleft}$ we denote $\Sigma \cup \{\triangleright, \triangleleft\}$. We tacitly assume that $\triangleright$ and $\triangleleft$ only occur on the left and right end of the string, respectively.

Definition 3.20. A *two-way alternating finite automaton (2AFA)* is a tuple $A = (Q, \Sigma_{\triangleright\triangleleft}, \delta, I, F, r, U)$ where

- Q is a finite set of states;
- I, F, U are subsets of Q and are the sets of initial, final and universal states, respectively;
- $r \in Q \setminus F$ is the rejecting state;
- $\delta : Q \times \Sigma_{\triangleright\triangleleft} \to 2^{Q \times \{\leftarrow, -, \rightarrow\}}$ is the transition function. ◇

A *configuration* of A on a string $w = \triangleright w_2 \cdots w_{n-1} \triangleleft$ is a pair (j, q), where $j \in \text{Nodes}(w)$ and $q \in Q$. Intuitively, j is the current tape position and q is the current state. A configuration (j, q) is *initial* (*accepting*) if $q \in I$ ($q \in F$) and $j = 1$ ($j = |w|$). A configuration (j, q) is *universal* (*existential*) if $q \in U$ ($q \in Q - U$). Given $\gamma = (j, q)$ and $\gamma' = (j', q')$, we define the *step*-relation $\vdash$ on configurations as follows: $\gamma \vdash \gamma'$ if and only if $(q', d) \in \delta(q, a)$, $\text{lab}^w(j) = a$, and $j' = j - 1$, $j' = j$, or $j' = j + 1$ if and only if $d = \leftarrow$, $d = -$, or $d = \rightarrow$, respectively. We assume that an automaton never attempts to move to the left (right) of a delimiter $\triangleright$ ($\triangleleft$). Further, we assume that A only reaches a final state at the delimiter $\triangleleft$ and that a computation branch of A only rejects by reaching r at the delimiter $\triangleleft$. Note that because of this last convention, the transition function of a two-way alternating finite automaton is complete, that is, for all $a \in \Sigma \cup \{\triangleright\}, q \in Q$, $\delta(q, a) \neq \emptyset$ and for all $q \in Q \setminus (\{r\} \cup F), \delta(q, \triangleleft) \neq \emptyset$. For a configuration γ, a γ*-run* of A on a string w is a (possibly infinite) tree where nodes are labeled with configurations as follows:

1. the root is labeled with γ;

2. every inner node labeled with an existential configuration γ has exactly one child γ' and $\gamma \vdash \gamma'$; and,

3. let for any universal configuration γ, $\{\gamma_1, \ldots, \gamma_m\} := \{\gamma' \mid \gamma \vdash \gamma'\}$, then every inner node labeled with γ has exactly m children labeled $\gamma_1, \ldots, \gamma_m$.

An *accepting* γ-run of A on w is a γ-run which does not contain an infinite path and where every leaf node is labeled with an accepting configuration. A *run* of A on w is a γ-run where γ is an initial configuration. A Σ-string w is *accepted* by A if there exists an accepting run of A on $\triangleright w \triangleleft$. The *language accepted by* A is defined to be the set of strings that are accepted by A, and is denoted by $L(A)$. The *size* of a A is $|\Sigma| + |Q| + \sum_{q \in Q, a \in \Sigma} |\delta(q, a)|$.

We say that A *loops* on w if there is a run on w which contains an infinite path. A 2AFA is then *loop-free* when it never loops. We denote the class of loop-free two-way alternating finite automata by 2AFA^{lf} . Note that 2AFA^{lf} accept only regular string languages [LLS84]. A *two-way non-deterministic finite automaton*, denoted 2NFA, is a 2AFA where $U = \emptyset$.

The construction in the next lemma is a slight adaptation of a construction from Vardi [Var89]. In Proposition 3.22, we use an on-the-fly construction of the automaton N constructed in this proof. Although the lemma appears in the literature without a restriction to loop-free automata [CGKV88], it is not clear how to adapt it to an on-the-fly algorithm.

Lemma 3.21. *Let A be an 2AFA^{lf}, then there exists an NFA N whose size is exponential in the size of A such that $L(N) = L(A)$.*

Proof. Let $A = (Q_A, \Sigma_{\triangleright\triangleleft}, \delta_A, I_A, F_A, r_A, U_A)$ be an 2AFA^{lf}. We construct an NFA $N = (Q_N, \Sigma_{\triangleright\triangleleft}, \delta_N, I_N, F_N)$ with $Q_N = (2^{Q_A} \times 2^{Q_A})$, $I_N = \{(\emptyset, U) \mid U \cap I_A \neq \emptyset\}$, $F_N = \{(U, \emptyset) \mid U \cap F_A \neq \emptyset \text{ and } r_A \notin U\}$. For ease of exposition, N also operates over delimited strings. Intuitively, when N is in state (U, V) when processing the jth symbol of input $w = w_1 \cdots w_n$, then for every state $p \in V$, A must accept $w_1 \cdots w_n$ when started in p on position j. Note that $w_1 = \triangleright$ and $w_n = \triangleleft$. The set U is the set V of position $j - 1$. Initial and final states are of the form $(\emptyset, U)$ and $(U, \emptyset)$ as the two-way automaton cannot move past the left and right delimiter, respectively.

The transition function is defined as follows. For every $(U, V), (T, U) \in Q_N$ and $a \in \Sigma_{\triangleright\triangleleft}$, $(U, V) \in \delta_A((T, U), a)$ if and only if, for every p in $U - F_A$, the following holds:

- if p is an existential state then there exists a pair $(p', d') \in \delta_A(p, a)$ such that $p' \in T$ if $d' =\leftarrow$, $p' \in U$ if $d' = -$, and $p' \in V$ if $d' =\rightarrow$; and,

- if p is a universal state then for all pairs $(p', d') \in \delta(p, a)$, $p' \in T$ if $d' =\leftarrow$, $p' \in U$ if $d' = -$, and $p' \in V$ if $d' =\rightarrow$.

Clearly, the size of N is exponential in the size of A. It remains to show that $L(A) = L(N)$. Clearly, $\triangleright\varepsilon\triangleleft \in L(A)$ if and only if $\triangleright\varepsilon\triangleleft \in L(N)$. Therefore, let $w = \triangleright w_1 \cdots w_n \triangleleft$ for $n > 0$. Suppose that there is an accepting run r of A on input w. Define $Q_0 = \emptyset$, $Q_i = \{p \mid (i, p) \text{ is a label in } r\}$ for $i = 1, \ldots, n+1$, $Q_{n+2} = \{p \mid (n+2, p)$

is a leaf label in $r\}$, and $Q_{n+3} = \emptyset$. It is easy to check that $(Q_0, Q_1) \in I_N$ and ρ is an accepting run for N on w where $\rho(i) = (Q_i, Q_{i+1})$ for $i = 1, \ldots, n+2$.

For the other direction, suppose ρ is an accepting run of N on w. Then, let $(Q_i, Q_{i+1}) = \rho(i)$ for every $i \in \text{Nodes}(w)$. For $i \in \text{Nodes}(w)$, define $m_d(i)$ as $i-1$, i, and $i+1$, when d is $\leftarrow, -, \rightarrow$, respectively. We define the depth of a configuration (i, q) where $q \in Q_i$, denoted $\text{depth}(i, q)$, inductively as follows: if $q \in F_A$ then $\text{depth}(i, q) = 0$; otherwise, $\text{depth}(i, q)$ is

$$\max\{\text{depth}(j, q') + 1 \mid (q', d) \in \delta_A(q, \text{lab}^w(i)), q' \in Q_j \text{ and } m_d(i) = j\}.$$

As A does not loop this notion is well-defined. By induction on the depth of configurations $\gamma = (i, q)$, it is easy to construct an accepting γ-run of height $\text{depth}(i, q)$. The claim then follows for an initial configuration $(1, q)$ with $q \in Q_1 \cap I_A$.

When a 2AFA is not loop-free, then the $\text{depth}(i, q)$ is not well-defined for all strings, and the construction of a run for the 2AFA from a run of the NFA might lead to an infinite tree. □

Proposition 3.22. EMPTINESS *for NTA(2AFA$^{\text{lf}}$) is in* PSPACE.

Proof. From the proof of Proposition 3.18(1), it follows that emptiness of an NTA can be reduced to a polynomial number of tests of the following form:

(i) $\varepsilon \in \delta(q, a)$; and,

(ii) $\delta(q, a) \cap R_{i-1}^* \neq \emptyset$.

We show that when $\delta(q, a)$ is represented by a 2AFA$^{\text{lf}}$, both tests can be done in PSPACE.

Let $B = (Q, \Sigma, \delta, F)$ be an NTA(2AFA$^{\text{lf}}$) and let for every $q \in Q$ and $a \in \Sigma$, $A^{q,a} = (Q^{q,a}, Q_{\triangleright\triangleleft}, \delta^{q,a}, I^{q,a}, F^{q,a}, r^{q,a}, U^{q,a})$ be the 2AFA$^{\text{lf}}$ representing $\delta(q, a)$. Denote by $N^{q,a}$ the NFA equivalent to $A^{q,a}$ given by the construction of Lemma 3.21. Of course, we cannot construct $N^{q,a}$ in polynomial space as it is exponentially bigger than $A^{q,a}$. Therefore, we will construct $N^{q,a}$ on the fly. We denote the transition function of $N^{q,a}$ by $\delta_N^{q,a}$.

We first argue that given a $b \in R_{i-1}^*$ and two states $(T, U), (U, V)$ of $N^{q,a}$, we can check in PSPACE that $(U, V) \in \delta_N^{q,a}((T, U), b)$. Indeed, we just have to check for all elements $p \in U$ the constraints mentioned in Lemma 3.21. That is, if p is existential, we check that there is a $(p', d') \in \delta_N^{q,a}(p, b)$ such that $p' \in T$, $p' \in U$, or $p' \in V$ depending on d'. If p is a universal state, we have to verify that for all $(p', d') \in \delta_N^{q,a}(p, b)$, $p' \in T$, $p' \in U$, or $p' \in V$ depending on d'. These two tests merely involve set membership and require only constant space.

We first describe the algorithm to check (i). We need to check whether $\triangleright\triangleleft$ is accepted by $N^{q,a}$. To this end, we guess states $(T_1, U_1), (T_2, U_2), (T_3, U_3)$ such that the first state is an initial state; the last state is an accepting state; and, $(T_2, U_2) \in \delta_N^{q,a}((T_1, U_1), \triangleright)$ and $(T_3, U_3) \in \delta_N^{q,a}((T_2, U_2), \triangleleft)$. By the previous discussion, the latter can be done in PSPACE.

Next, we describe the algorithm to check (ii). Given $R_{i-1} \subseteq Q$, $q \in Q$ and $a \in \Sigma$, we need to check whether $q \in R_i$. The latter reduces to verifying whether there is some string $b_1 \cdots b_n$ in R_{i-1}^* that is accepted by $A^{q,a}$ or, equivalently, $N^{q,a}$.

1. Initialization step: We start by guessing an initial state (T, U) and a state (U, V) such that $(U, V) \in \delta_N^{q,a}((T, U), \triangleright)$. We write the state (U, V) on the tape.

2. Iteration step: Let (U, V) be the state written on the tape. We guess a state (U', V') such that $(U', V') \in \delta_N^{q,a}((U, V), \triangleleft)$. If (U', V') is final, then we know that $R_{i-1}^* \cap A^{q,a} \neq \emptyset$ and accept. Otherwise, we erase (U', V') and guess a symbol $b \in R_{i-1}$ and a state (U'', V'') such that $(U'', V'') \in \delta_N^{q,a}((U, V), b)$. We erase (U, V), write (U'', V'') on the tape and resume at at the beginning of the iteration step.

Clearly, $R_{i-1}^* \cap A^{q,a} \neq \emptyset$ if and only if there is a run of the algorithm that accepts. Further, by the discussion above, the algorithm only uses polynomial space. □

3.5 Tiling Systems

Tiling systems can be quite a useful tool for proving complexity lower bounds. We therefore briefly discuss the notions of corridor tiling and two-player corridor tiling. We make use of tiling systems in the proof of Theorem 5.6 and in Chapter 9. A *tiling system* is a tuple $D = (T, H, V, \bar{b}, \bar{t}, n)$ where n is a natural number, T is a finite set of *tiles*; $H, V \subseteq T \times T$ are *horizontal* and *vertical constraints*, respectively; and $\bar{b}, \bar{t}$ are n-tuples of tiles ($\bar{b}$ and $\bar{t}$ stand for *bottom row* and *top row*, respectively). A *corridor tiling* is a mapping $\lambda : \{1, \ldots, m\} \times \{1, \ldots, n\} \to T$, for some $m \in \mathbb{N}$, such that $\bar{b} = (\lambda(1,1), \ldots, \lambda(1,n))$ and $\bar{t} = (\lambda(m,1), \ldots, \lambda(m,n))$. Intuitively, the first and last row of the tiling are $\bar{b}$ and $\bar{t}$, respectively. A tiling is *correct* if it respects the horizontal and vertical constraints. That is, for every $i = 1, \ldots, m$ and $j = 1, \ldots, n-1$, $(\lambda(i,j), \lambda(i,j+1)) \in H$, and for every $i = 1, \ldots, m-1$ and $j = 1, \ldots, n$, $(\lambda(i,j), \lambda(i+1,j)) \in V$.

To every tiling system we can associate a *game* as follows: the game consists of two players (Constructor and Spoiler). The game is played on an $\mathbb{N} \times n$ board. Each player places tiles in turn. While player Constructor tries to construct a corridor tiling, player Spoiler tries to prevent it. Player Constructor wins if Spoiler makes an illegal move (with respect to H or V), or when a correct corridor tiling can be constructed. We say that Constructor has a *winning strategy* if she wins no matter what Spoiler does.

In the sequel, we use reductions from the following problems:

- Corridor Tiling: given a tiling system, is there a correct corridor tiling?

- Two-Player Corridor Tiling: given a tiling system, does Constructor have a winning strategy?

The following theorem is due to Chlebus [Chl86].

Theorem 3.23.

(1) Corridor Tiling *is* pspace-*complete.*

(2) Two-player Corridor Tiling *is* exptime-*complete.*

4

Towards a Tractable Case for Typechecking

We initiate the study of the typechecking problem for XML transformations. For the most general transducers, we show that even for very weak DTDs (DTDs that use DFAs or SL-formulas to represent regular languages), the typechecking problem is EXPTIME-complete. We then investigate how the complexity of typechecking can be reduced by restricting the power of the tree transducers and schema languages.

In practice, the number of copies a transformation makes is usually rather small (see, for example, Example 2.15 where the first rule makes two copies of every chapter). Therefore, it makes sense to consider the class of transducers making at most C copies where C is a number fixed in advance. In Section 4.2, we restrict the copying power of the transducers, while still allowing deletion. Unfortunately, typechecking still remains EXPTIME-complete for the weakest DTDs we consider. In Section 4.3, we investigate the orthogonal restriction: non-deleting transformations with unbounded copying. We distinguish between tree automata and DTDs as schema languages. In the case of tree automata, the complexity remains EXPTIME-hard. When considering DTDs the complexity drops to PSPACE when NFAs or DFAs are used to specify right-hand sides; when SL-formulas are used the complexity drops to coNP. The PSPACE lower bound crucially depends on the ability of a transducer to make arbitrary copies of the input tree. We show in Section 4.4 that even on this class, in the case of tree automata and DTDs with NFAs, the complexity remains EXPTIME and PSPACE-hard, respectively. Only when right-hand sides of rules are represented by DFAs, the typechecking problem becomes tractable.

In conclusion, our inquiries reveal that the complexity of the typechecking problem is determined by three features: (1) the ability of the transducer to delete interior nodes; (2) the ability to make an unbounded number of copies of subtrees; and, (3) non-determinism in the schema languages. Only when we disallow all three features, we get a PTIME complexity for the typechecking problem.

An overview of our results is given in Table 4.1. All complexities are both upper and lower bounds. NTA(X) and DTA(X) stand for non-deterministic and bottom-up deterministic tree automata, that use X to represent their regular languages, respectively. DTD(X) stands for DTDs that use X to represent their regular languages. The exact definitions were given in Chapter 2.

TT	NTA(NFA)	DTA(DFA)	DTD(NFA)	DTD(DFA)	DTD(SL)
d,uc	EXPTIME	EXPTIME	EXPTIME	EXPTIME	EXPTIME
d,bc	EXPTIME	EXPTIME	EXPTIME	EXPTIME	EXPTIME
nd,uc	EXPTIME	EXPTIME	PSPACE	PSPACE	coNP
nd,bc	EXPTIME	EXPTIME	PSPACE	PTIME	coNP

Table 4.1: Complexities of the typechecking problem in the present setting (upper and lower bounds). The top row shows the representation of the input and output schemas, the leftmost column shows the class of tree transducer: "d", "nd", "uc", and "bc" stand for "deleting", "non-deleting", "unbounded copying", and "bounded copying" respectively.

4.1 The General Case

We start by considering the complexity of the typechecking problem in its most general setting. That is, without any restrictions on transducers: both deletion and unbounded copying is allowed. We show that the problem is in EXPTIME for the most powerful schema languages, namely non-deterministic tree automata. However, the problem remains hard for EXPTIME even for the weakest DTDs: DTDs where right-hand sides are specified by DFAs or SL-formulas.

The lower bound is obtained through a reduction from the intersection emptiness problem of n top-down deterministic binary tree automata, which is known to be hard for EXPTIME (Proposition 3.9). The transducer starts by making n copies of the input tree. Thereafter, it simulates a different tree automaton on each copy. During this simulation, the transducer deletes all the processed nodes. The only output that it generates is an "error" symbol when an automaton rejects. So, the output DTD merely has to check that an "error" symbol always appears. The latter can be done by a very simple DFA or SL-formula.

The EXPTIME upper bound is obtained by a translation to the typechecking problem for non-deleting transducers. The latter problem is tackled in the next section.

Theorem 4.1.

(1) $TC[\mathcal{T}_{d,uc}, NTA(NFA)]$ *is in* EXPTIME*;*

(2) $TC[\mathcal{T}_{d,uc}, DTD(DFA)]$ *is* EXPTIME*-hard; and,*

(3) $TC[\mathcal{T}_{d,uc}, DTD(SL)]$ *is* EXPTIME*-hard.*

Proof. (1): Let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a transducer and let A_{in} and $A_{\text{out}} = (Q_A, \Sigma, \delta_A, F_A)$ be two NTAs representing the input and output schema, respectively. We next describe a *non*-deleting transducer S and an NTA B_{out} which can be constructed in LOGSPACE, such that T typechecks with respect to A_{in} and A_{out} if and only if S typechecks with respect to A_{in} and B_{out}. From Theorem 4.3(1) it then follows that TC[$\mathcal{T}_{\text{d,uc}}$,NTA] is in EXPTIME.

Intuitively, the non-deleting tree transducer S outputs the special symbol "#" whenever T would process a deleting state. For instance, the rule $(q, a) \to c\, q$ is replaced by $(q, a) \to c\, \#(q)$. We assume that $\# \notin \Sigma$. Formally, $S = (Q_S, \Sigma \cup \{\#\}, q_S^0, R_S)$ with $Q_S = Q_T$, $q_S^0 = q_T^0$, and for every rule $(q, a) \to t_1 \cdots t_n$ in R_T, R_S contains the rule $(q, a) \to t_1' \cdots t_n'$, where for every $i = 1, \ldots, n$, $t_i' = \#(t_i)$ if $t_i \in Q_T$ and $t_i' = t_i$ otherwise. Then, define the #-eliminating function γ as follows: $\gamma(a(h))$ is $\gamma(h)$ when $a = \#$ and $a(\gamma(h))$ otherwise; further, $\gamma(t_1 \cdots t_n) := \gamma(t_1) \cdots \gamma(t_n)$. So, clearly, for all $t \in \mathcal{T}_\Sigma$, $T(t) = \gamma(S(t))$.

Next, we construct B_{out} such that $\gamma(t) \in L(A_{\text{out}})$ if and only if $t \in L(B_{\text{out}})$. The underlying idea is quite simple. In a run on $\#(t_1 \cdots t_n)$, B_{out} assigns a state (q_1, q_2, q, a) to the root when the NFA for $\delta_A(q, a)$ halts in state q_2 when processing $\text{top}(\gamma(\#(t_1 \cdots t_n)))$ starting in state q_1. Here, q_1, q_2 are states of the automaton for $\delta_A(q, a)$, q is a state of A_{out} and $a \in \Sigma$. The state q and the label a are guessed. In a run on $a(t_1 \cdots t_n)$, with $a \neq \#$, B_{out} assigns a state q to the root when A_{out} assigns q to the root of $\gamma(a(t_1 \cdots t_n))$.

Let for every $a \in \Sigma$ and $q \in Q_A$, $N^{q,a} = (Q^{q,a}, Q_A, \delta^{q,a}, I^{q,a}, F^{q,a})$ be the NFA such that $\delta_A(q, a) = L(N^{q,a})$. We tacitly assume that all $Q^{q,a}$ are disjoint. Define $B_{\text{out}} = (Q_B, \Sigma \cup \{\#\}, \delta_B, F_B)$, where $Q_B = Q_A \cup \{(q_1, q_2, q, a) \mid q \in Q_A, a \in \Sigma, q_1, q_2 \in Q^{q,a}\}$, and $F_B = F_A$.

It remains to define δ_B. Thereto, fix $q \in Q_A$ and $a \in \Sigma$. Let $I, F \subseteq Q^{q,a}$. Let $M^{q,a}(I, F)$ be the automaton behaving in the same way as $N^{q,a}$ with the initial and final states replaced with I and F, respectively; further, when reading a tuple (q_1, q_2, p, b) in state q_1 the automaton jumps to state q_2 when $p = q$ and $b = a$, and rejects otherwise. Clearly, $M^{q,a}(I, F)$ is LOGSPACE constructible from $N^{q,a}$. We then simply define $\delta_B(q, a) := M^{q,a}(I^{q,a}, F^{q,a})$ and $\delta_B((q_1, q_2, p, b), \#) := M^{p,b}(\{q_1\}, \{q_2\})$ for all states $q, (q_1, q_2, p, b) \in Q_B$ and $a \in \Sigma$. It is not difficult to see that $\gamma(t) \in L(A_{\text{out}})$ if and only if $t \in L(B_{\text{out}})$.

(2),(3): It follows immediately from Theorem 4.2 that TC[$\mathcal{T}_{\text{d,uc}}$,DTD(DFA)] and TC[$\mathcal{T}_{\text{d,uc}}$,DTD(SL)] are EXPTIME-hard. □

4.2 Deleting Transformations with Bounded Copying

The lower bound of the previous section severely depends on the ability of transducers to *delete* the interior nodes of the input tree and to make an *unbounded number of copies* of subtrees. In an attempt to lower the complexity of typechecking, we restrict this copying power in the present section and the deletion power in Section 4.3. We obtain that bounding the copying power in the case of deleting tree transducers does not lower the complexity of typechecking: it remains EXPTIME-complete.

Theorem 4.2.

(1) $TC[\mathcal{T}_{d,bc}, DTD(DFA)]$ *is* EXPTIME-*complete; and*

(2) $TC[\mathcal{T}_{d,bc}, DTD(SL)]$ *is* EXPTIME-*complete.*

Proof. The EXPTIME upper bound follows from Theorem 4.1. We proceed by proving the lower bounds.

We give a LOGSPACE reduction from the INTERSECTION EMPTINESS for top-down deterministic binary tree automata (TDBTAs) over the alphabet $\Sigma = \{0, 1, 0', 1', 0'', 1''\}$. The intersection emptiness problem of TDBTAs over alphabet $\{0, 1, 0', 1', 0'', 1''\}$ is known to be EXPTIME-hard (see Corollary 3.13).

For $i = 1, \ldots, n$, let $A_i = (Q_i, \Sigma, \delta_i, \{\text{start}_i\})$ be a TDBTA, with $\Sigma = \{0, 1, 0', 1', 0'', 1''\}$. We assume that $\{0, 1\}$, $\{0', 1'\}$, and $\{0'', 1''\}$ are the symbols of rank 0, 1, and 2, respectively. Without loss of generality, we can assume that the state sets Q_i are pairwise disjoint. We call 0 and 1 *leaf labels* and $0', 1', 0''$ and $1''$ *internal labels*. In our proof, we use the markers "ℓ" and "r" to denote that a certain node is a left or a right child. Formally, define $\Sigma_\ell := \{a_\ell \mid a \in \Sigma\}$ and $\Sigma_r := \{a_r \mid a \in \Sigma\}$. We use symbols from Σ_ℓ and Σ_r for the left and right children of nodes, respectively.

We now define a transducer T and two DTDs d_{in} and d_{out} such that $\bigcap_{i=1}^{n} L(A_i) = \emptyset$ if and only if T typechecks with respect to d_{in} and d_{out}. In the construction, we exploit the copying power of transducers to make n copies of the input tree: one for each A_i. By using deleting states, we can execute each A_i on its copy of the input tree without producing output. When an A_i does not accept, we output an *error* symbol under the root of the output tree. The output DTD should then only check that an *error* symbol always appears. A bit of care needs to be taken, as a bounded copying transducer can not make an arbitrary number of copies of the input tree in the same rule. The transducer therefore goes through an initial copying phase where it repeatedly copies part of the input tree twice, until there are (at least) n copies. The transducer remains in the copying phase as long as it processes special symbols "#", which we assume not to be in Σ. The input trees are therefore of the form as depicted in Figure 4.1. In addition, the transducer should verify that the number of #-symbols in the input equals $\lceil \log n \rceil$.

The input DTD $(\Sigma_\ell \cup \Sigma_r \cup \{s, \#\}, d_{\text{in}}, s)$, which we will describe next, defines all trees of the form as described in Figure 4.1, where s and # are alphabet symbols, and t (which is depicted in Figure 4.1) t is a ranked Σ-tree, annotated with the symbols "ℓ" and "r". That is, when a node is an only child, it is labeled with an element of Σ_ℓ. Otherwise, it is labeled with an element of Σ_ℓ or an element of Σ_r if it is a left child or a right child, respectively. In this way, the transducer knows whether a node is a left or a right child by examining the label. The root symbol of t is labeled with a symbol from Σ_ℓ. Furthermore, as t is a ranked tree, all nodes of t with one or two children are labeled with labels in $\{0'_\ell, 0'_r, 1'_\ell, 1'_r\}$ and $\{0''_\ell, 0''_r, 1''_\ell, 1''_r\}$, respectively, and all leaf nodes are labeled with labels in $\{0_\ell, 0_r, 1_\ell, 1_r\}$.

The input DTD $(\Sigma_\ell \cup \Sigma_r \cup \{s, \#\}, d_{\text{in}}, s)$ is defined as follows:

- $d_{\text{in}}(s) = \# + 0_\ell + 1_\ell + 0'_\ell + 1'_\ell + 0''_\ell + 1''_\ell;$

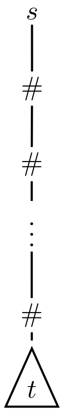


Figure 4.1: Structure of the trees defined by the input schema in the proof of Theorem 4.2.

- $d_{\text{in}}(\#) = \# + 0_\ell + 1_\ell + 0'_\ell + 1'_\ell + 0''_\ell + 1''_\ell$;
- for each $a \in \{0'_\ell, 1'_\ell, 0'_r, 1'_r\}, d_{\text{in}}(a) = (0_\ell + 1_\ell + 0'_\ell + 1'_\ell)$;
- for each $a \in \{0''_\ell, 1''_\ell, 0''_r, 1''_r\}, d_{\text{in}}(a) = (0'_\ell + 1'_\ell + 0''_\ell + 1''_\ell)(0'_r + 1'_r + 0''_r + 1''_r)$; and,
- for each $a \in \{0_\ell, 1_\ell, 0_r, 1_r\}$, $d_{\text{in}}(a) = \varepsilon$.

Obviously, d_{in} can be expressed as a DTD(DFA). It can also be expressed as a DTD(SL), as follows. For example, for every $a \in \{0''_\ell, 1''_\ell, 0''_r, 1''_r\}$, we define

$$\begin{aligned} d_{\text{in}}(a) = \Big(& ((\varphi[0_\ell^{=1}] \vee \varphi[1_\ell^{=1}] \vee \varphi[(0'_\ell)^{=1}] \vee \varphi[(1'_\ell)^{=1}]) \vee \varphi[(0''_\ell)^{=1}] \vee \varphi[(1''_\ell)^{=1}]) \\ & \wedge (\varphi[0_r^{=1}] \vee \varphi[1_r^{=1}] \vee \varphi[(0'_r)^{=1}] \vee \varphi[(1'_r)^{=1}]) \vee \varphi[(0''_r)^{=1}] \vee \varphi[(1''_r)^{=1}])\Big) \\ & \wedge s^{=0}, \end{aligned}$$

where for every $i \in \{\ell, r\}$ and $x \in \{0_i, 1_i, 0'_i, 1'_i, 0''_i, 1''_i\}$, $\varphi[x^{=1}]$ denotes the conjunction

$$(x^{=1} \wedge \bigwedge_{y \in \{0_i, 1_i, 0'_i, 1'_i, 0''_i, 1''_i\} \setminus \{x\}} y^{=0}).$$

Notice that the size of the SL-formula expressing $d_{\text{in}}(a)$ is constant.

We construct a tree transducer $T = (Q_T, \Sigma_T, q^\varepsilon_{\text{copy}}, R_T)$. The alphabet of T is $\Sigma_T = \Sigma_\ell \cup \Sigma_r \cup \{s, \#, \text{error}, \text{ok}\}$. It will use $\lceil \log n \rceil$ special copying states q^j_{copy} to make at least n copies of the input tree. To define Q_T formally, we first introduce the notation $D(k)$, for $k = 0, \dots, \lceil \log n \rceil$. Intuitively, $D(k)$ is similar to the set of nodes of a complete binary tree of depth $k+1$. For example, $D(1) = \{\varepsilon, 0, 1\}$ and $D(2) = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$. The idea is that, if $i \in D(k) \setminus D(k-1)$, for $k > 0$, then i represents the binary encoding of a number in $\{0, \dots, 2^k - 1\}$. Formally, if

$k = 0$, then $D(k) = \{\varepsilon\}$; otherwise, $D(k) = D(k-1) \cup \bigcup_{j=0,1}\{ij \mid i \in D(k-1)\}$. The state set Q_T is then the union of the sets $Q^\ell = \{q^\ell \mid q \in Q_j, 1 \leq j \leq n\}$, $Q^r = \{q^r \mid q \in Q_j, 1 \leq j \leq n\}$, the set $\{q^j_{\text{copy}} \mid j \in D(\lceil \log n \rceil)\}$ and the set $\{\text{start}^\ell_j \mid n+1 \leq j \leq 2^{\lceil \log n \rceil}\}$. Note that the last set can be empty. It only contains dummy states translating any input to the empty string.

We next describe the action of the tree transducer T. Roughly, the operation of T on the input $s(\#(\#(\cdots\#(t))))$ can be divided in two parts: *(i)* copying the tree t a sufficient number of times while reading the #-symbols; and, *(ii)* simulating one of the TDBTAs on each copy of t. The tree transducer outputs the symbol "error" when one of the TDBTAs rejects t, or when the number of #-symbols in its input is not equal to $\lceil \log n \rceil$. Apart from copying the root symbol r to the output tree, T only writes the symbol "error" to the output. Hence, the output tree always has a root labeled s which has zero or more children labeled "error". The output DTD, which we define later, should then verify whether the root has always one "error"-labeled child.

Formally, the transition rules in R_T are defined as follows:

- $(q^\varepsilon_{\text{copy}}, s) \rightarrow s(q^0_{\text{copy}} q^1_{\text{copy}})$. This rule puts s as the root symbol of the output tree.

- $(q^i_{\text{copy}}, \#) \rightarrow q^{i0}_{\text{copy}} q^{i1}_{\text{copy}}$ for $i \in D(\lceil \log n \rceil - 1) - \{\varepsilon\}$. These rules copy the tree t in the input at least n times, provided that there are enough #-symbols.

- $(q^i_{\text{copy}}, \#) \rightarrow \text{start}^\ell_k$, where $i \in D(\lceil \log n \rceil) - D(\lceil \log n \rceil - 1)$, and i is the binary representation of k. This rule starts the in-parallel simulation of the A_i's. For $i = n+1, \ldots, 2^{\lceil \log n \rceil}$, start^ℓ_i is just a dummy state transforming everything to the empty tree.

- $(q^i_{\text{copy}}, a) \rightarrow$ error for $a \in \Sigma_\ell \cup \Sigma_r$ and $i \in D(\lceil \log n \rceil)$. This rule makes sure that the output of T is accepted by the output tree automaton if there are not enough #-symbols in the input.

- $(\text{start}^\ell_k, \#) \rightarrow$ error for all $k = 1, \ldots, 2^{\lceil \log n \rceil}$. This rule makes sure that the output of T is accepted by the output tree automaton if there are too much #-symbols in the input.

- $(q^\ell, a_r) \rightarrow \varepsilon$ and $(q^r, a_\ell) \rightarrow \varepsilon$ for all $q \in Q_j$, $j = 1, \ldots, n$. This rule ensures that tree automata states intended for left (respectively right) children are not applied to right (respectively left) children.

- $(q^\ell, a_\ell) \rightarrow q_1^\ell q_2^r$ and $(q^r, a_r) \rightarrow q_1^\ell q_2^r$, for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = q_1 q_2$, and a is an internal symbol. This rule does the actual simulation of the tree automata A_i, $i = 1, \ldots, n$.

- $(q^\ell, a_\ell) \rightarrow q_1^\ell$ and $(q^r, a_r) \rightarrow q_1^\ell$, for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = q_1$ and a is an internal symbol. This rule does the actual simulation of the tree automata A_i, $i = 1, \ldots, n$.

- $(q^\ell, a_\ell) \rightarrow \varepsilon$ and $(q^r, a_r) \rightarrow \varepsilon$ for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = \varepsilon$ and a is a leaf symbol. This rule simulates accepting computations of the A_i's.

- $(q^\ell, a_\ell) \to \text{error}$ and $(q^r, a_r) \to \text{error}$ for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a)$ is undefined. This rule simulates rejecting computations of the A_i's.

It is straightforward to verify that, on input $s(\#(\#(\cdots\#(t))))$, T outputs the tree s if and only if there are $\lceil \log n \rceil$ #-symbols in the input and $t \in L(A_1) \cap \cdots \cap L(A_n)$.

Finally, $d_{\text{out}}(s) = \text{error error}^*$, which can easily be defined as a DTD(DFA) and as a DTD(SL).

It is easy to see that the reduction can be carried out in deterministic logarithmic space, that T has copying width 2, and that d_{in} and d_{out} do not depend on $A_1, \ldots, A_n$. □

4.3 Non-deleting Transformations

Unfortunately, bounding the copying power of deleting tree transducers does not seem to lower the complexity of typechecking. In this section, we investigate to which extent the complexity lowers when we restrict ourselves to *non-deleting* the tree transducers, while still allowing unbounded copying. We observe that, when schemas are represented by tree automata, the complexity remains EXPTIME-hard. When tree languages are represented by DTDs, the complexity of the typechecking problem drops to PSPACE and is hard for PSPACE even when right-hand sides of rules are represented by DFAs. When employing SL-formulas the complexity is coNP. In summary, we prove the following results:

Theorem 4.3.

(1) $TC[\mathcal{T}_{nd,uc}, NTA(NFA)]$ *is* EXPTIME-*complete;*

(2) $TC[\mathcal{T}_{nd,uc}, DTA(DFA)]$ *is* EXPTIME-*complete;*

(3) $TC[\mathcal{T}_{nd,uc}, DTD(NFA)]$ *is* PSPACE-*complete;*

(4) $TC[\mathcal{T}_{nd,uc}, DTD(DFA)]$ *is* PSPACE-*complete;*

(5) $TC[\mathcal{T}_{nd,uc}, DTD(SL)]$ *is co*NP-*complete.*

We prove the different parts of the above theorem in the following subsections.

4.3.1 Tree Automata

The proof establishing the upper bound in the case of typechecking with respect to tree automata as schema languages is similar in spirit to a proof by Neven and Schwentick [NS02], which shows that containment of Query Automata is in EXPTIME.

Theorem 4.3(1). $TC[\mathcal{T}_{nd,uc}, NTA(NFA)]$ *is* EXPTIME-*complete.*

Proof. Hardness is immediate as containment of NTAs is already hard for EXPTIME (Corollary 3.10). We therefore only prove membership in EXPTIME. Let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a non-deleting tree transducer and let $A_{\text{in}} = (Q_{\text{in}}, \Sigma, \delta_{\text{in}}, F_{\text{in}})$ and

$A_{\text{out}} = (Q_{\text{out}}, \Sigma, \delta_{\text{out}}, F_{\text{out}})$ be the NTAs representing the input and output schema, respectively.

In brief, our algorithm computes the set

$$P = \{(S,f) \mid S \subseteq Q_{\text{in}}, f : Q_T \to (2^{Q_{\text{out}}})^*, \exists t \text{ such that } \\ S = \delta^*_{\text{in}}(t) \text{ and } \forall q \in Q_T, f(q) = \delta^*_{\text{out}}(T^q(t))\}.$$

Note that since $f(q) = \delta^*_{\text{out}}(T^q(t))$ and t is a tree,[1] the length of $f(q)$ is bounded by the size of the largest rhs in T. Therefore, the number of functions f we consider is bounded by $(2^{|Q_{\text{out}}|})^{|T||Q_T|}$. Intuitively, in the definition of P, t can be seen as a witness of (S,f). Indeed, S is the set of states reachable by A_{in} at the root of t, while for each state q of the transducer, $f(q)$ is the sequence of sets of states reachable by A_{out} at the root of $T^q(t)$. So, the given instance does *not* typecheck if and only if there exists an $(S,f) \in P$ such that $F_{\text{in}} \cap S \neq \emptyset$ and $F_{\text{out}} \cap f(q^0_T) = \emptyset$. As $T^{q^0_T}(t)$ is always a tree, $f(q^0_T)$ is a subset of Q_{out}. In Figure 4.2, an algorithm for computing P is depicted. We will show that this algorithm is in EXPTIME. Hence, typechecking is in EXPTIME. We explain the notation in Figure 4.2. By $\text{rhs}(q,a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T]$, we denote the hedge obtained from $\text{rhs}(q,a)$ by replacing every occurrence of a state p by the sequence $f_1(p) \cdots f_n(p)$. By $\hat{\delta}_{\text{out}} : \mathcal{H}_\Sigma(2^{Q_c}) \to (2^{Q_c})^*$ we denote the transition function extended to hedges in $\mathcal{H}_\Sigma(2^{Q_{\text{out}}})$. To be precise, for $a \in \Sigma$, $\hat{\delta}_{\text{out}}(a) := \{q \mid \varepsilon \in \delta_{\text{out}}(q,a)\}$; for $P \subseteq Q_{\text{out}}$, $\hat{\delta}_{\text{out}}(P) := P$; for $h = a(t_1 \cdots t_n)$, $\hat{\delta}_{\text{out}}(h) := \{q \mid \forall i = 1, \ldots, n, \exists q_i \in \hat{\delta}_{\text{out}}(t_i) : q_1 \cdots q_n \in \hat{\delta}_{\text{out}}(q,a)\}$; and for $h = t_1 \cdots t_n$, $\hat{\delta}_{\text{out}}(h) = \hat{\delta}_{\text{out}}(t_1) \cdots \hat{\delta}_{\text{out}}(t_n)$. The correctness of the algorithm follows from the following lemma, which is proved by induction on the number of iterations of the while loop.

Claim 4.4. *A pair (S,f) has a witness tree of depth i if and only if $(S,f) \in P_i$.*

Proof. Immediate for $i = 1$.

For the induction step, suppose that, for some i, every pair is in P_{i-1} if and only if it has a witness of depth $i-1$. Let $(S,f) \in P_i$, then, by definition, there is an $a \in \Sigma$ and a string $(S_1,f_1) \cdots (S_n,f_n) \in P^*_{i-1}$ so that $S := \{p \mid \exists r_j \in S_j, j = 1, \ldots, n, r_1 \cdots r_n \in \delta_{\text{in}}(p,a)\}$ and for every $q \in Q_T$, $f(q) := \delta^*_{\text{out}}\big(\text{rhs}(q,a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T]\big)$. Hence, $a(t_1 \cdots t_n)$ is a witness of (S,f), where each t_j is a witness for (S_j,f_j).

Conversely, suppose that (S,f) has a witness tree $a(t_1 \cdots t_n)$ of depth i. By the induction hypothesis, there exist tuples $(S_1,f_1), \ldots, (S_n,f_n) \in P_{i-1}$ such that t_j is a witness for (S_j,f_j) for each $j = 1, \ldots, n$. Considering the definition of $\hat{\delta}_{\text{out}}$, it is then clear that the algorithm of Figure 4.2 puts (S,f) in P_i. □

It remains to show that the algorithm is in EXPTIME. The set P_1 can be computed in time polynomial in the sizes of A_{in}, A_{out}, and T. As $P_i \subseteq P_{i+1}$ for all i, and there are $2^{|Q_{\text{in}}|} \cdot (2^{|Q_{\text{out}}|})^{|T||Q_T|}$ pairs (S,f), the loop can only make an exponential number of iterations. So, it suffices to show that each iteration can be done in EXPTIME. Actually, we argue that it can be checked in PSPACE whether a tuple $(S,f) \in P_i$.

[1] Recall that $T^q(t)$ is the translation of t started in state q. See page 16 for the formal definition.

Let (S, f) be a pair. We describe separately how S and f are checked. It should be clear how the two algorithms can be merged into one PSPACE algorithm. We start with S.

(a) For every $q \in Q_{\text{in}}$ and $a \in \Sigma$, let $N^{q,a}$ be the NFA accepting those strings $R_1 \cdots R_k \in (2^{Q_{\text{in}}})^*$ for which there are $r_i \in R_i$ such that $r_1 \cdots r_k \in \delta_{\text{in}}(q, a)$. It is too expensive to actually construct the automaton $N^{q,a}$ as the alphabet is exponentially bigger than the one of $\delta_{\text{in}}(q, a)$. However, the set of states is the same. It is important to note that given a set R_i and a state q, the set of all states reachable from q by reading R_i can be computed in PSPACE.

So, we need to check the existence of an $a \in \Sigma$ and a string $Z := S_1 \cdots S_n$ that is accepted (rejected) by $N^{q,a}$ for all $q \in S$ ($q \in Q_{\text{in}} \setminus S$). The latter can be achieved in PSPACE by guessing an $a \in \Sigma$ and then guessing Z one symbol at a time while executing all $N^{q,a}$'s in parallel for every $q \in Q_{\text{in}}$. Indeed, for every automaton we remember the set of states that can be reached by reading the prefix of Z seen so far. Initially, these sets are the respective initial states. Then, whenever a new S_i is guessed, for each automaton the set of states reachable from a state from the remembered set by reading S_i, is computed. By the discussion above the latter is in PSPACE.

(b) Checking f is more technical. We use the a guessed in the previous step. Denote $\text{rhs}(q, a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T]$ by $\xi_{q,a}$. Now, we need to check for all $q \in Q_T$ whether $f(q) = \hat{\delta}_{\text{out}}(\xi_{q,a})$. For all $p \in Q_{\text{out}}$ and $b \in \Sigma$, let $M^{p,b}$ be the NFA accepting strings $R_1 \cdots R_k \in (2^{Q_{\text{out}}})^*$ for which there are $r_i \in R_i$, $i = 1, \ldots, k$, such that $r_1 \cdots r_k \in \delta_{\text{out}}(p, b)$. Again, we will not construct the latter automata. It is enough to realize that given a state and an $R \subseteq Q_{\text{out}}$, the set of states reachable from this state by reading R can be computed in PSPACE.

First, assume every $\text{rhs}(q, a)$ is of the form $b(q_1 \cdots q_\ell)$. Then, $\xi_{q,a}$ is of the form $b(w_1 \cdots w_\ell)$ with $w_j = f_1(q_j) \cdots f_n(q_j)$. So, to check that $f(q) = \hat{\delta}_{\text{out}}(\xi_{q,a})$, we need to verify that $w = w_1 \cdots w_\ell$ is accepted (rejected) by $M^{p,b}$ for all $p \in f(q)$ ($p \notin f(q)$). However, like in (1), our algorithm successively guesses new f_i's while forgetting the previous ones and should, hence, be able to run the automata on w in this way. As w consists of ℓ parts we guess ℓ sets of states $P_i^{p,b}$, $i = 0, \ldots, \ell$, where $P_0^{p,b}$ is the set of initial states of $M^{p,b}$. The meaning of these sets is the following: every automaton $M^{p,b}$ reaches precisely the states in $P_i^{p,b}$ after reading $w_1 \cdots w_{i-1}$. The algorithm can verify the latter criterion by running $M^{p,b}$ on each w_i separately started in the states $P_{i-1}^{p,b}$ and verifying whether $P_i^{p,b}$ is reached. Running $M^{p,b}$ on w_i can be done in PSPACE as described in part (a).

When right-hand sides of rules can be arbitrary trees in $\mathcal{T}(Q_T)$, we guess for every inner node u in a $\text{rhs}(q, a)$ a subset $R_u^{q,a}$ of Q_{out}. When u is the root, then $R_u^{q,a} = f(q)$. Intuitively, these sets represent precisely the sets of states that can be reached at a node u by A_{out}. For leaf nodes u, we define $R_u^{q,a}$ as $\delta^*_{\text{out}}(c)$ and as the sequence $f_1(p) \cdots f_n(p)$ when u is labeled with c and p, respectively. We then need to verify for every inner node u labeled with b with n children, that $R_{u1}^{q,a} \cdots R_{un}^{q,a}$ is accepted (rejected) by $M^{p,b}$ for all $p \in R_u^{q,a}$ ($p \notin R_u^{q,a}$). Again, the latter is checked as described above.

```
$P_0 := \emptyset$;
$i := 1$;
$P_1 := \big\{(\delta^*_{\mathrm{in}}(a), f) \mid a \in \Sigma, \forall q \in Q_T : f(q) = \delta^*_{\mathrm{out}}(T^q(a))\big\}$;
while $P_i \neq P_{i-1}$ do
  $P_i := \big\{(S, f) \mid \exists (S_1, f_1) \cdots (S_n, f_n) \in P^*_{i-1}, \exists a \in \Sigma :$
          $S = \{p \mid \exists r_k \in S_k, k = 1, \ldots, n, r_1 \cdots r_n \in \delta_{\mathrm{in}}(p, a)\}$,
          $\forall q \in Q_T : f(q) = \hat{\delta}_{\mathrm{out}}\big(\mathrm{rhs}(q, a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T]\big)\big\}$;
  $i := i + 1$;
end while
$P := P_i$;
```

Figure 4.2: The algorithm of Theorem 4.3(1) computing P.

Finally, when right-hand sides of rules can be hedges, one needs to take into account that $f(q)$ can be a sequence of sets of states. □

The EXPTIME-hardness of TC[$\mathcal{T}_{\mathrm{nd,uc}}$,DTA(DFA)] follows immediately from part (2) of Theorem 4.5.

4.3.2 DTDs

When we consider DTD(NFA)s to represent input schemas the complexity drops to PSPACE. We reduce the typechecking problem to the emptiness problem of NTAs where transition functions are represented by loop-free two-way alternating finite automata, denoted 2AFA$^{\mathrm{lf}}$.[2] The complexity of the latter problem is in PSPACE (Proposition 3.22 in Section 3.4). In particular, the constructed NTA accepts precisely those trees which satisfy the input DTD but are transformed by the transducer to trees outside the output DTD. Hence, the instance typechecks if and only if the NTA accepts the empty language. The proof makes use of two-way non-deterministic string automata, denoted 2NFA, which are also defined in Section 3.4 (page 39).

Theorem 4.3(3). *$TC[\mathcal{T}_{nd,uc}, DTD(NFA)]$ is* PSPACE*-complete.*

Proof. The hardness result can be shown by an easy reduction from the universality problem of NFAs with alphabet $\{0,1\}$. The latter problem is PSPACE-hard, as shown in Corollary 3.13.

To this end, let N be an NFA with alphabet $\{0,1\}$. The input DTD $(\{s,0,1\}, d_{\mathrm{in}}, s)$ defines a tree of depth two where $d_{\mathrm{in}}(s) = (0+1)^*$. The tree transducer is the identity transformation. The output DTD d_{out} has as start symbol s and $d_{\mathrm{out}}(s) = L(N)$. Hence, this instance typechecks if and only if $\{0,1\}^* \subseteq L(N)$. This reduction can be carried out by a deterministic logspace algorithm.

For the complexity upper bound, let T be a non-deleting tree transducer. Let d_{in} and d_{out} be the input and output DTDs, respectively. We construct an NTA(2AFA$^{\mathrm{lf}}$) B such that $L(B) = \{t \in L(d_{\mathrm{in}}) \mid T(t) \notin L(d_{\mathrm{out}})\}$. Moreover, the size of B is

[2] Alternating finite automata are discussed in Section 3.4.

polynomial in the size of T, d_{in}, and d_{out}. Thus, $L(B) = \emptyset$ if and only if T typechecks with respect to d_{in} and d_{out}. By Proposition 3.22, we can test whether $L(B) = \emptyset$ is in PSPACE, which shows that typechecking is possible in PSPACE.

To explain the operation of the automaton, we introduce the following notions. Recall that for a state q of T and $a \in \Sigma$ we denote the string $\text{top}(\text{rhs}(q, a))$ by $q[a]$. For a string $w = a_1 \cdots a_n$, we denote by $q[w]$ the string $q[a_1] \cdots q[a_n]$. For a hedge h and a DTD d, we say that h *partly satisfies* d if for every $u \in \text{Nodes}(h)$, $\text{lab}^h(u1) \cdots \text{lab}^h(un) \in L(d(\text{lab}^h(u)))$ where u has n children. Note that there is no requirement on the root nodes of the trees in h. Hence, the term partly.

Intuitively, the automaton B works as follows on $t \in \mathcal{T}_\Sigma$: *(i)* B checks that $t \in L(d_{\text{in}})$; and, *(ii)* at the same time, B non-deterministically picks a node $v \in \text{Nodes}(t)$ and a state q in which v is processed; B then accepts if h does not partly satisfy d_{out}, where h is obtained from $\text{rhs}(q, a)$ by replacing every state p by the string $p(\text{lab}^t(v1) \cdots \text{lab}^h(vn))$. Here, we assume that v is labeled a and has n children. As d_{out} is specified by NFAs and we have to check that d_{out} is *not* partly satisfied, we need to check membership in the complement of a regular expression. We therefore use alternation to specify the transition function of B. Additionally, as T can copy its input, it is convenient to use two-way automata. The latter will become clear in the actual construction.

Formally, let $T = (Q_T, \Sigma, q_T^0, R_T)$. Define $B = (Q_B, \Sigma, F_B, \delta_B)$ as follows. The set of states Q^B is the union of the following sets: Σ, $\{(a, q) \mid a \in \Sigma, q \in Q_T\}$, and $\{(a, q, \text{check}) \mid a \in \Sigma, q \in Q_T\}$. If there is an accepting run on a tree t, then a node v labeled with a state of the form a, (a, q), (a, q, check) has the following meaning:

a: v is labeled with a and the subtree rooted at v partly satisfies d_{in}.

(a, q): same as in previous case with the following two additions: (1) v is processed by T in state q; and, (2) a descendant of v will produce a tree that does not partly satisfy d_{out}.

(a, q, check): same as the previous case only now v itself will produce a tree that does not partly satisfy d_{out}.

The set of final states is $F_B := \{(a, q_T^0) \mid a \in \Sigma\}$. The transition function is defined as follows: for all $a, b \in \Sigma$, $q \in Q_T$:

1. $\delta_B(a, b) = \delta_B((a, q), b) = \delta_B((a, q, \text{check}), b) = \emptyset$ for all $a \neq b$;

2. $\delta_B(a, a) = d_{\text{in}}(a)$ and $\delta_B((a, q), a)$ consists of those strings $a_1 \cdots a_n$ such that there is precisely one index $j \in \{1, \ldots, n\}$ for which $a_j = (b, p)$ or $a_j = (b, p, \text{check})$ where p occurs in $\text{rhs}(q, a)$ and for all $i \neq j$, $a_i \in \Sigma$; further, $a_1 \cdots a_{j-1} b a_{j+1} \cdots a_n \in L(d_{\text{in}}(a))$. Note that $\delta_B((a, q), a)$ is defined in such a way that it ensures that all subtrees partly satsify d_{in} and that at least one subtree will generate a violation of d_{out}. Clearly, $\delta_B(a, a)$ and $\delta_B((a, q), a)$ can be represented by NFAs whose size is polynomial in the size of the input.

3. Finally, $\delta_B((a, q, \text{check}), a) = \{a_1 \cdots a_n \mid a_1 \cdots a_n \in d_{\text{in}}(a) \text{ and } h \text{ does not partly satisfy } L(d_{\text{out}})\}$. Here, h is obtained from $\text{rhs}(q, a)$ by replacing every state p by $p(a_1 \cdots a_n)$.

It remains to argue that $\delta_B((a, q, \text{check}), a)$ can be computed by a 2AFA$^{\text{lf}}$ A of polynomial size. We sketch the construction of this automaton. First, for every $b \in \Sigma$ and $m \in \{\text{out}, \text{in}\}$, let A^b_m be the NFA accepting $d_m(b)$.

For every v in rhs(q, a), let s_v be concatenation of the labels of the children of v. Define the 2NFA N_v as follows: suppose s_v is of the form $z_0 p_1 z_1 \cdots p_\ell z_\ell$ where $z_i \in \Sigma^*$ and $p_i \in Q_T$, then $a_1 \cdots a_n \in L(N_v)$ if and only if

$$z_0 p_1 (a_1 \cdots a_n) z_1 \cdots p_\ell (a_1 \cdots a_n) z_\ell \in L(A^{\text{lab}^h(v)}_{\text{out}}).$$

As s_v is fixed, N_v can recognize this language by reading $a_1 \cdots a_n$ ℓ times while simulating $A^{\text{lab}^h(v)}_{\text{out}}$. More precisely, the automaton simulates $A^{\text{lab}^h(v)}_{\text{out}}$ on $z_{i-1} p_i (a_1 \cdots a_n)$ on the $(i + 1)$-th pass. Note that N_v does not loop.

It remains to describe the construction of the 2AFA$^{\text{lf}}$ A. On input $a_1 \cdots a_n$, A first checks whether $a_1 \cdots a_n \in L(A^a_{\text{in}})$ by simulating A^a_{in}. Hereafter, A goes back to the beginning of the input string, guesses an internal node v in rhs(q, a) and simulates the complement of N_v. As N_v is a 2NFA that does not loop, A is a 2AFA$^{\text{lf}}$ whose size is linear in the size of the N_v's. This completes the construction of B. □

The next result shows that typechecking remains PSPACE-hard even when NFAs are replaced by DFAs. The main source of complexity is the ability of transducers to make an arbitrary number of copies.

Theorem 4.3(4). $TC[\mathcal{T}_{nd,uc}, DTD(DFA)]$ *is* PSPACE*-complete.*

Proof. We reduce the intersection emptiness problem of an arbitrary number of deterministic finite automata with alphabet $\{0, 1\}$ to the typechecking problem. This problem is known to be PSPACE-hard, as shown in Corollary 3.13 in Section 3.3. Our reduction only requires logarithmic space. We define a tree transducer $T = (Q_T,$ $\{0, 1, \#_0, \ldots, \#_n\}, q^0_T, R_T)$ and two DTDs d_{in} and d_{out} such that T typechecks with respect to d_{in} and d_{out} if and only if $\bigcap_{i=1}^n L(M_i) = \emptyset$.

The DTD $(\{s, 0, 1\}, d_{\text{in}}, s)$ defines trees of depth two, where the string formed by the children of the root is an arbitrary string in $\{0, 1\}^*$, so $d_{\text{in}}(s) = (0 + 1)^*$. The transducer makes n copies of this string, separated by the delimiters $\#_i$: $Q_T = \{q, q^0_T\}$ and R_T contains the rules $(q^0_T, s) \to s(\#_0 q \#_1 q \ldots \#_{n-1} q \#_n)$ and $(q, a) \to a$, for every $a \in \Sigma$. Finally, d_{out}, with start symbol s, defines a tree of depth two as follows:

$$d_{\text{out}}(s) = \{\#_0 w_1 \#_1 w_2 \#_2 \cdots \#_{n-1} w_n \#_n \mid \\ \exists j \in \{1, \ldots, n\} \text{ such that } M_j \text{ does not accept } w_j\}.$$

Clearly, $d_{\text{out}}(s)$ can be represented by a DFA whose size is polynomial in the sizes of the M_i's. Indeed, the DFA just simulates every M_i on the string following $\#_{i-1}$, until it encounters $\#_i$. It then verifies that at least one M_i rejects.

It is easy to see that this reduction can be carried out by a deterministic logspace algorithm. □

Next, we focus on SL-expressions as right-hand sides of DTDs. In this case, we see that the complexity of typechecking drops to coNP. In the proof of the following theorem, we make use of Lemmas 3.1 and 3.2, which are stated and proven in Section 3.1.

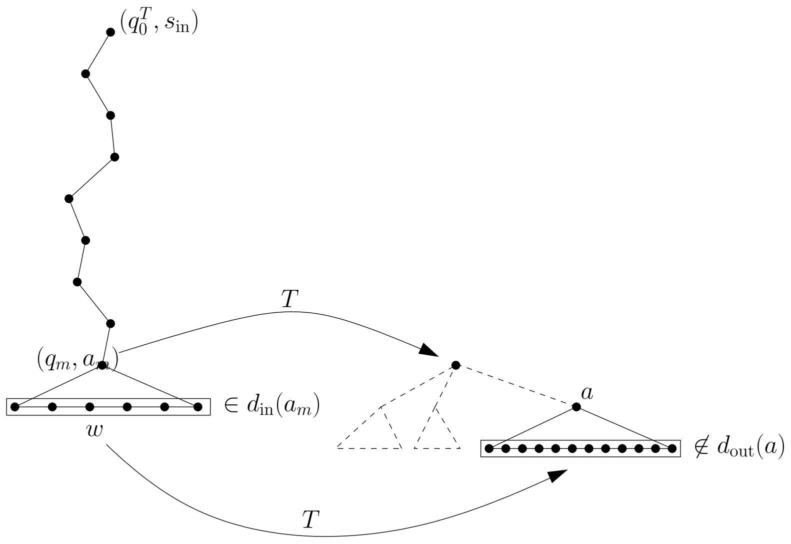


Figure 4.3: Illustration of the typechecking algorithm in the proof of Theorem 4.3(5).

Theorem 4.3(5). *$TC[\mathcal{T}_{nd,uc}, DTD(SL)]$ is co*NP*-complete.*

Proof. A coNP lower bound is obtained by a reduction from the emptiness problem of DTD(SL)s, which is coNP-complete (Proposition 3.16). Indeed, let d be a DTD(DFA). Let T be a tree transducer that relabels every symbol in the input tree to a. Finally, let d_{out} be a DTD(SL) that defines the empty language over alphabet $\{a\}$. Then $L(d) = \emptyset$ if and only if T typechecks with respect to d and d_{out}. Notice that d_{out} is in fact fixed as it uses a fixed alphabet.

Next, we prove the upper bound. Let $T = (Q_T, \Sigma, q_T^0, R_T)$ and let $(\Sigma, d_{\text{in}}, s_{\text{in}})$ and $(\Sigma, d_{\text{out}}, s_{\text{out}})$ be the input and output DTD respectively. We describe an NP algorithm that guesses a counterexample, that is, a tree $t \in L(d_{\text{in}})$ such that $T(t) \notin L(d_{\text{out}})$. In brief, we would like to guess an input tree t satisfying d_{in}, a node $v \in \text{Nodes}(t)$ labeled with a and a state $q \in Q_T$ in which v is processed such that $T^q(a(w))$ does not satisfy d_{out}. Here, w is the string obtained by concatenating the labels of the children of v. An immediate problem is that we cannot simply guess a whole tree t as the size of the latter might be exponential in the size of d_{in}. Therefore, we simply guess a path ending in v which can be extended to a tree satisfying d_{in} and a string of children w with the desired property. We explain this next.

Let k be the largest number occurring in any SL-formula in d_{in} or d_{out}. Set $r := (k + 1) \cdot |\Sigma|$.

The algorithm consists of three main parts:

1. First, we sequentially guess a subset D of the derivable symbols $\{b \in \Sigma \mid L((\Sigma, d_{\text{in}}, b)) \neq \emptyset\}$.

2. Next, we guess a path of a tree in $L(d_{\text{in}})$. In particular, we guess a sequence of pairs $(q_0, a_0), \ldots, (q_m, a_m)$ in $Q_T \times D$, with $m \leq |\Sigma| \cdot |Q_T|$, such that

   (a) $a_0 = s_{\text{in}}$ and $q_0 = q_T^0$;

   (b) there is a tree $t \in L(d_{\text{in}})$ and a node $v \in \text{Nodes}(t)$ such that $a_0 \cdots a_m$ is the concatenation of the labels of the nodes on the path from the root to v; and,

   (c) for all $i = 0, \ldots, m$: T visits a_i in state q_i.

3. Finally, we guess a string $w \in D^*$ of length at most r such that $T^{q_m}(a_m(w))$ does not partly satisfy d_{out}. As r can be exponentially large, we do not guess w itself, but a representation of w. Here, partly satisfaction is as defined in the proof of Theorem 4.3(3).

We illustrate the operation of the algorithm in Figure 4.3. In this figure, T visits the a_m-labeled node on the left in state q_m. Consequently, T outputs the hedge $\text{rhs}(q, a)$, which is illustraded by dotted lines on the right. The typechecking algorithm searches for a node u in $\text{rhs}(q, a)$ (which is labeled by a in the figure), such that the string of children of u is not in $L(d_{\text{out}}(a))$.

We describe in detail how the three parts can be implemented and show that the verification of the guesses can be done in PTIME. As all the guesses can be done at the beginning, we obtain an NP algorithm.

1. We compute D as follows.

   (a) Guessing phase: guess a sequence of different symbols $b_1, \ldots, b_{m'}$ in Σ. So, $m' \leq |\Sigma|$. Guess vectors $v_1, \ldots, v_{m'}$ where each $v_i = (\ell_1^i, \ldots, \ell_{i-1}^i) \in \{0, \ldots, k+1\}^{i-1}$. Intuitively, the vector v_i corresponds to the string $b_1^{\ell_1^i} \cdots b_{i-1}^{\ell_{i-1}^i}$. So, we interchangeably talk about the vector and the string v_i. Note that some ℓ_j^i may be zero.

   (b) Checking phase. For each $i = 1, \ldots, m'$, test that the string v_i satisfies $d_{\text{in}}(b_i)$. Note that this can be done in PTIME.

   Let $S_i = \{b_j \mid j \leq i\}$. From Lemma 3.1, it follows that if there is a string w in S_i^* such that w satisfies $d_{\text{in}}(b_i)$ then there is one such that each symbol occurs at most $k+1$ times. Hence, it suffices to guess vectors in $\{0, \ldots, k+1\}^{i-1}$. Finally, a simple induction shows that $D \subseteq \{b \in \Sigma \mid L((\Sigma, d_{\text{in}}, b)) \neq \emptyset\}$.

2. The requirement (a) can easily be checked. (c) can be checked by verifying that $q_{i+1} \in \text{rhs}(q_i, a_i)$ for all i. Let $D = \{b_1, \ldots, b_{|D|}\}$. To test (b), it suffices to guess a vector $v_i = (\ell_1, \ldots, \ell_{|D|}) \in \{0, \ldots, k+1\}^{|D|}$ for every $i \in \{0, \ldots, m-1\}$ such that $\ell_j \neq 0$ when $a_{i+1} = b_j$ and test whether $b_1^{\ell_1} \cdots b_{|D|}^{\ell_{|D|}}$ satisfies $d_{\text{in}}(a_i)$. As every symbol is in D, the path can be expanded to a tree satisfying d_{in}. By Lemma 3.1, it follows that guessing vectors of that size suffices. The upper bound $|\Sigma| \cdot |Q_T|$ on m can be obtained by a simple pumping argument.

3. Before we describe the last part of the algorithm, we make the link explicit between the transducer T, the function f and the c's described in Lemma 3.2. We start with some notation. Let q be a state of T and $a \in \Sigma$ then define $q[a] :=$ top(rhs(q, a)). For a string $w = a_1 \cdots a_n$, we define $q[w] := q[a_1] \cdots q[a_n]$. For $a \in \Sigma$ and $w \in \Sigma^*$, we also define $\#_a(w)$ to be the number of a's occurring in w. Let $q \in Q_T$, $a \in \Sigma$ and let u be a node in rhs(q, a). Let $z = z_0 p_1 z_1 \cdots p_\ell z_\ell$ be the concatenation of the labels of the children of u, such that $p_i \in Q_T$ and $z_i \in \Sigma^*$. For every $s \in \Sigma^*$, define $f_u^{q,a}(s)$ as the string obtained from z by replacing every p_i by the string $p_i(s)$. Now, we define the c's corresponding to $f_u^{q,a}(s)$. For every $b \in \Sigma$, set $c^b := \#_b(z)$ and for every $e \in \Sigma$, set $c_e^b := \sum_{j=1}^{\ell} \#_b(p_j(e))$. Clearly, for every $b \in \Sigma$ and every $s \in \Sigma^*$, $\#_b(f_u^{q,a}(s)) = c^b + \sum_{e \in \Sigma}(c_e^b \cdot \#_e(s))$.

   So, the algorithm guesses a node u in rhs(q_m, a_m). We do not guess a string w but rather a vector in $\{1, \ldots, k+1\}^{|\Sigma|}$ representing such a string (as in the previous bullets). We check whether $f_u^{q_m,a_m}(w)$ does not satisfy $d_{\text{out}}(a)$ where the label of u is a. Take f as $f_u^{q_m,a_m}$, ϕ_1 as $d_{\text{in}}(a_m)$, and ϕ_2 as $d_{\text{out}}(a)$. Then from Lemma 3.2, it follows that it suffices to guess a string represented by a vector in $\{1, \ldots, k+1\}^{|\Sigma|}$. This completes the description of the algorithm. □

4.4 Non-deleting Transformations with Bounded Copying

As can be inferred from Theorem 4.3, disallowing deletion lowers the complexity of the typechecking problem in the presence of DTDs. Unfortunately, the problem still remains intractable. In the context of DTD(DFA)s, the high complexity is a consequence of the copying power of transducers (cf. the proof of Theorem 4.3(4)). Therefore, we bound in advance the copying width of transducers by only considering transducers in the class $\mathcal{T}_{\text{nc,bc}}^{C}$ for a fixed C (see Section 2.3). In the case of DTD(DFA)s we then finally obtain a tractable scenario.

Theorem 4.5.

(1) $TC[\mathcal{T}_{nd,bc}, NTA(NFA)]$ *is* EXPTIME-*complete;*

(2) $TC[\mathcal{T}_{nd,bc}, DTA(DFA)]$ *is* EXPTIME-*complete;*

(3) $TC[\mathcal{T}_{nd,bc}, DTD(NFA)]$ *is* PSPACE-*complete;*

(4) $TC[\mathcal{T}_{nd,bc}, DTD(DFA)]$ *is* PTIME-*complete;*

(5) $TC[\mathcal{T}_{nd,bc}, DTD(SL)]$ *is* coNP-*complete.*

The lower bounds of (1), (3), and (5) follow immediately from the construction in the proofs of Theorem 4.3(1), (3), and (5). We deal with the remaining cases.

For the proof of Theorem 4.5(2), we reduce from the intersection emptiness problem of top-down deterministic ranked binary tree automata. The proof is similar to the one in Theorem 4.2. The tree transducer A_{in} for the input schema defines the same set of trees as d_{in}. The transducer in the proof of Theorem 4.2 starts the in parallel simulation of the n automata, but then, using deleting states, delays the output until it has reached the leaves of the input tree. In the present setting, we can not use deleting states. Instead, we copy the input tree and overwrite the leaves with error symbols when an automaton rejects. The output automaton then checks whether at least one error occurred.

Theorem 4.5(2). *$TC[\mathcal{T}_{nd,bc}, DTA(DFA)]$ is* EXPTIME-*complete.*

Proof. We give a LOGSPACE reduction from the intersection emptiness problem of an arbitrary number of top-down deterministic binary tree automata (TDBTAs) over the alphabet $\Sigma = \{0, 1, 0', 1', 0'', 1''\}$. The intersection emptiness problem of TDBTAs over alphabet $\{0, 1, 0', 1', 0'', 1''\}$ is known to be EXPTIME-hard (see Corollary 3.13).

For $i = 1, \ldots, n$, let $A_i = (Q_i, \Sigma, \delta_i, \{\text{start}_i\})$ be a top-down deterministic ranked binary tree automaton over alphabet $\Sigma = \{0, 1, 0', 1', 0'', 1''\}$. The transducer T is defined over the alphabet $\Sigma_T = \Sigma_\ell \cup \Sigma_r \cup \{s, \text{error}\}$. Here, for $i \in \{\ell, r\}$, $\Sigma_i = \{a_\ell \mid a \in \Sigma\}$.

First, we define $A_{\text{in}} = (Q_{\text{in}}, \Sigma_T, \delta_{\text{in}}, \{q_0\})$, where $Q_{\text{in}} = \{q_0, q_\ell, q_r, q_\#\}$. The intuition is that A_{in} accepts all trees $s(\#(\cdots\#(t)))$ as described in the proof of Theorem 4.2 and illustrated in Figure 4.1. That is, t is an arbitrary ranked Σ-tree in which every label is annotated with the symbol "ℓ" or "r" when it is a left child or right child of its parent, respectively. The transition function of A_{in} is formally defined as follows:

- $\delta_{\text{in}}(q_0, s) = q_\ell + q_\#$;
- $\delta_{\text{in}}(q_\#, \#) = q_\ell + q_\#$;
- for each $a \in \{0'', 1''\}$, $\delta(q_\ell, a_\ell) = \delta(q_r, a_r) = q_\ell q_r$;
- for each $a \in \{0', 1'\}$, $\delta(q_\ell, a_\ell) = \delta(q_r, a_r) = q_\ell$; and,
- for each $a \in \{0, 1\}$, $\delta(q_\ell, a_\ell) = \delta(q_r, a_r) = \varepsilon$;

Note that A_{in} is bottom-up deterministic.

The transducer $T = (Q_T, \Sigma_T, q_T^0, R_T)$ is defined similarly as in Theorem 4.2: $Q_T = \bigcup_{i=1}^n (Q_i^\ell \cup Q_i^r)$, where $Q_i^k = \{q^k \mid q \in Q_i\}$ for $k \in \{\ell, r\}$. The intuition is that states in Q_i^ℓ should only be used to process the left child and states in Q_i^r to process the right child. The set R_T consists of the following rules:

- $(q^\varepsilon_{\text{copy}}, s) \to s(q^0_{\text{copy}} q^1_{\text{copy}})$.
- $(q^i_{\text{copy}}, \#) \to \#(q^{i0}_{\text{copy}} q^{i1}_{\text{copy}})$ for $i \in D(\lceil \log n \rceil - 1) - \{\varepsilon\}$.
- $(q^i_{\text{copy}}, \#) \to \#(\text{start}^\ell_k)$, where $i \in D(\lceil \log n \rceil) - D(\lceil \log n \rceil - 1)$, and i is the binary representation of k.
- $(q^i_{\text{copy}}, a) \to \text{error}$ for $a \in \Sigma_\ell \cup \Sigma_r$ and $i \in D(\lceil \log n \rceil)$.
- $(\text{start}^\ell_k, \#) \to \text{error}$ for all $k = 1, \ldots, 2^{\lceil \log n \rceil}$.
- $(q^\ell, a_r) \to \varepsilon$ and $(q^r, a_\ell) \to \varepsilon$ for all $q \in Q_j$, $j = 1, \ldots, n$.

- $(q^\ell, a_\ell) \to a_\ell(q_1^\ell q_2^r)$ and $(q^r, a_r) \to a_r(q_1^\ell q_2^r)$, for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = q_1 q_2$, and a is an internal symbol.
- $(q^\ell, a_\ell) \to a_\ell(q_1^\ell)$ and $(q^r, a_r) \to a_r(q_1^\ell)$, for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = q_1$ and a is an internal symbol.
- $(q^\ell, a_\ell) \to \varepsilon$ and $(q^r, a_r) \to \varepsilon$ for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = \varepsilon$ and a is a leaf symbol.
- $(q^\ell, a_\ell) \to$ error and $(q^r, a_r) \to$ error for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a)$ is undefined.

Finally, we define the output automaton $A_{\text{out}} = (Q_{\text{out}}, \Sigma_T, \delta_{\text{out}}, \{q_{\text{found}}\})$ which accepts all trees with at least one error-labeled leaf. Formally, $Q_{\text{out}} = \{q_{\text{notfound}}, q_{\text{found}}\}$ and δ_{out} is defined as follows:

- $\delta_{\text{out}}(q_{\text{notfound}}, s) = q^*_{\text{notfound}}$;
- $\delta_{\text{out}}(q_{\text{found}}, s) = Q^*_{\text{out}} q_{\text{found}} Q^*_{\text{out}}$;
- for each $a \in \{\#, 0'_\ell, 1'_\ell, 0'_r, 1'_r, 0''_\ell, 1''_\ell, 0''_r, 1''_r\}$, $\delta_{\text{out}}(q_{\text{notfound}}, a) = q^*_{\text{notfound}}$;
- for each $a \in \{\#, 0'_\ell, 1'_\ell, 0'_r, 1'_r, 0''_\ell, 1''_\ell, 0''_r, 1''_r\}$, $\delta_{\text{out}}(q_{\text{found}}, a) = Q^*_{\text{out}} q_{\text{found}} Q^*_{\text{out}}$;
- for each $a \in \{0_\ell, 1_\ell, 0_r, 1_r\}$, $\delta(q_{\text{notfound}}, a) = \varepsilon$; and,
- $\delta_{\text{out}}(q_{\text{found}}, \text{error}) = \varepsilon$.

Notice that A_{out} is bottom-up deterministic.

For future reference (page 86), we note that the above described reduction also holds when the tree transducer has copying width one, but in which the right-hand sides are allowed to contain two states. Indeed, the reduction still holds if we replace every rewrite rule of the form $(q, a) \to b(q_1\ q_2)$ by $(q, a) \to b(b(q_1)\ b(q_2))$ and use the same output schema. □

Theorem 4.5(4). *$TC[\mathcal{T}_{nd,bc}, DTD(DFA)]$ is* PTIME*-complete.*

Proof. A PTIME lower bound is obtained by a reduction from the emptiness problem of DTD(DFA)s, which is PTIME-complete (Proposition 3.16). Indeed, let d be a DTD(DFA). Let T be a tree transducer that defines the identity transformation and let d_{out} be a DTD(DFA) that defines the empty language. Then $L(d) = \emptyset$ if and only if T typechecks with respect to d and d_{out}.

We now discuss the upper bound. In the proof of Theorem 4.3(3), we reduced TC[$\mathcal{T}_{\text{nd,uc}}$,DTD(NFA)] to the emptiness problem of NTA(2AFA$^{\text{lf}}$)s. In that proof, alternation was needed to express negation of the NFAs in the output schema; two-wayness was needed because T could make arbitrary copies of the input tree. However, when transducers can make only a bounded number of copies and DFAs are used, TC[$\mathcal{T}_{\text{nd,bc}}$,DTD(DFA)] can be LOGSPACE-reduced to emptiness of NTA(NFA)s. From Proposition 3.18, it then follows that TC[$\mathcal{T}_{\text{nd,bc}}$,DTD(DFA)] is in PTIME. □

5

Fixing the Input and Output Schemas

The typechecking scenario outlined in the previous chapter is very general: both the schemas and the transducer are determined to be part of the input. However, for some exchange scenarios, it makes sense to consider the input and/or output schema to be fixed when transformations are always from within and/or to a specific community.

Therefore, we revisit the various instances of the typechecking problem considered in Chapter 4 and determine the complexity in the presence of fixed input and/or output schemas. The main goal of this chapter is to investigate to which extent the complexity of the typechecking problem is lowered in scenarios where the input and/or output schema is fixed. From a complexity theory point of view, it is important to note here that the input and/or output alphabet then also becomes fixed. An overview of our results is presented in Table 5.1. All problems are complete for the mentioned complexity classes unless specified otherwise.

We discuss the obtained results: for non-deleting transformations, we get three new tractable cases: *(i)* fixed input schema, *un*bounded copying, and DTD(SL)s; *(ii)* fixed output schema, bounded copying and DTD(NFA)s; and, *(iii)* fixed input and output schemas, *un*bounded copying and all DTDs. It is striking, however, that in the presence of deletion or tree automata (even deterministic ones) typechecking remains EXPTIME-hard for *all* scenarios.

Mostly, we only needed to strengthen the lower bound proofs of Chapter 4.

Notation. We introduce some notations that are central to the present chapter. We denote the typechecking problem where the input schema, the output schema, or both are fixed by $\mathrm{TC}^i[\mathcal{T},\mathcal{S}]$, $\mathrm{TC}^o[\mathcal{T},\mathcal{S}]$, and $\mathrm{TC}^{io}[\mathcal{T},\mathcal{S}]$, respectively. The complexity of these subproblems is measured in terms of the sum of the sizes of the input and output schemas S_{in} and S_{out}, and the transducer T, minus the size of the fixed

fixed	TT	NTA(NFA)	DTA(DFA)
in, out, in+out	d,uc	EXPTIME	EXPTIME
	d,bc	EXPTIME	EXPTIME
in, out, in+out	nd,uc	EXPTIME	EXPTIME
	nd,bc	EXPTIME	EXPTIME

fixed	TT	DTD(NFA)	DTD(DFA)	DTD(SL)
in, out, in+out	d,uc	EXPTIME	EXPTIME	EXPTIME
	d,bc	EXPTIME	EXPTIME	EXPTIME
in	nd,uc	PSPACE	PSPACE	in PTIME
	nd,bc	PSPACE	NLOGSPACE	in PTIME
out	nd,uc	PSPACE	PSPACE	coNP
	nd,bc	PTIME	PTIME	coNP
in+out	nd,uc	NLOGSPACE	NLOGSPACE	NLOGSPACE
	nd,bc	NLOGSPACE	NLOGSPACE	NLOGSPACE

Table 5.1: Complexities of the typechecking problem in the new setting (upper and lower bounds). The top rows of the tables show the representation of the input and output schemas, the leftmost columns show which schemas are fixed, and the second columns to the left show the class of tree transducer: “d”, “nd”, “uc”, and “bc” stand for “deleting”, “non-deleting”, “unbounded copying”, and “bounded copying” respectively. In the case of deleting transformations, the different possibilities are grouped as all complexities coincide.

schema(s).

5.1 Deletion: Fixed Input Schema, Fixed Output Schema, and Fixed Input and Output Schema

The EXPTIME upper bound for typechecking already follows from Theorem 4.1 in Chapter 4. Therefore, it remains to consider the lower bounds for the problems $\mathrm{TC}^{io}[\mathcal{T}_{\mathrm{d,bc}}, \mathrm{DTD(DFA)}]$ and $\mathrm{TC}^{io}[\mathcal{T}_{\mathrm{d,bc}}, \mathrm{DTD(SL)}]$. However, observe that the input and output schema in the proof of Theorem 4.2 in fact do not depend on the input of the reduction. We hence immediately obtain the following:

Theorem 5.1.

(1) $TC^{io}[\mathcal{T}_{d,bc}, DTD(DFA)]$ *is* EXPTIME*-complete; and*

(2) $TC^{io}[\mathcal{T}_{d,bc}, DTD(SL)]$ *is* EXPTIME*-complete.*

In fact, if follows from the proof that the lower bounds already hold for transducers with copying width 2.

5.2 Non-deleting: Fixed Input Schema

We turn to the typechecking problem in which we only allow non-deleting tree transducers. First, we consider the input schema as fixed. We start by showing that typechecking is in PTIME in the case where we use DTDs with SL-expressions. To this end, we recall a necessary notion that is needed for the proof of Theorem 5.2. For a hedge h and a DTD d, we say that h *partly satisfies* d if for every $u \in \text{Nodes}(h)$, $\text{lab}^h(u1) \cdots \text{lab}^h(un) \in L(d(\text{lab}^h(u)))$ where u has n children.

We are now ready to show the first PTIME result in this chapter:

Theorem 5.2. *$TC^i[\mathcal{T}_{nd,uc}$, DTD(SL)] is in* PTIME.

Proof. Denote the tree transformation by $T = (Q_T, \Sigma, q_T^0, R_T)$ and the input and output DTDs by $(\Sigma, d_{\text{in}}, s_{\text{in}})$ and $(\Sigma, d_{\text{out}}, s_{\text{out}})$, respectively. As d_{in} is fixed, we can assume that d_{in} is reduced.

Intuitively, the typechecking algorithm is successful when T does *not* typecheck with respect to d_{in} and d_{out}. The algorithm is in the same spirit as the algorithm used in the proof of Theorem 4.3(5). The outline of the algorithm is as follows:

1. Compute the set RP of "reachable pairs" (q, a) for which there exists a tree $t \in L(d_{\text{in}})$ and a node $u \in \text{Nodes}(t)$ such that $\text{lab}^t(u) = a$ and T visits u in state q. That is, we compute all pairs (q, a) such that either
   - $q = q_T^0$ and $a = s_{\text{in}}$; or
   - $(q', a') \in RP$, there is a q-labeled node in $\text{rhs}(q', a')$, and there exists a string $w_1 a w_2 \in d_{\text{in}}(a')$ for $w_1, w_2 \in \Sigma^*$.
2. For each such pair (q, a) and for each node $v \in \text{Nodes}(\text{rhs}(q, a))$, test whether there exists a string $w \in d_{\text{in}}(a)$ such that $T^q(a(w))$ does not partly satisfy d_{out}. We call w a *counterexample.*

The algorithm is successful, if and only if there exists a counterexample.

Notice that the typechecking algorithm does not assume that d_{out} is reduced (recall the definition of a reduced DTD from Section 2.2). We need to show that the algorithm is correct, that is, there exists a counterexample if and only if T does not typecheck with respect to d_{in} and d_{out}. Clearly, when the algorithm does not find a counterexample, T typechecks with respect to d_{in} and d_{out}. Conversely, suppose that the algorithm finds a pair (q, a) and a string w such that $T^q(a(w))$ does not partly satisfy d_{out}. So, since we assumed that d_{in} is reduced, there exists a tree $t \in L(d_{\text{in}})$ and a node $u \in \text{Nodes}(t)$ such that $\text{lab}^t(u) = a$ and u is visited by T in state q. Also, there exists a node v in $T^q(a(w))$, such that the label of u is c and the string of children of u is not in $d_{\text{out}}(c)$. We argue that $T(t) \notin L(d_{\text{out}})$. There are two cases:

(i) if $L(d_{\text{out}})$ contains a tree with a c-labeled node, then $T(t) \notin d_{\text{out}}$ since $T^q(a(w))$ does not partly satisfy d_{out}; and

(ii) if $L(d_{\text{out}})$ does *not* contain a tree with a c-labeled node, then $T(t) \notin d_{\text{out}}$ since $T(t)$ contains a c-labeled node.

We proceed by showing that the algorithm can be carried out in polynomial time. As the input schema is fixed, step (1) of the algorithm is in polynomial time. Indeed, we can compute the set RP of reachable pairs (q, a) in a top-down manner by a straightforward reachability algorithm.

To show that step (2) of the typechecking algorithm is in polynomial time, fix a tuple (q, a) that was computed in step (1) and a node u in $\text{rhs}(q, a)$ with label b. Let $z_0 q_1 z_1 \cdots q_n z_n$ be the concatenation of u's children, where all $z_0, \ldots, z_n \in \Sigma^*$ and $q_1, \ldots, q_n \in Q_T$. We now search for a string $w \in \Sigma^*$ for which $w \models d_{\text{in}}(a)$, but for which $z_0 q_1[w] z_1 \cdots q_n[w] z_n \not\models d_{\text{out}}(b)$. Recall from Section 2.3 that $q[w]$ is the homomorphic extension of $q[a]$ for $a \in \Sigma$, which is $\text{top}(\text{rhs}(q, a)))$ in the case of non-deleting tree transducers.

Denote $d_{\text{in}}(a)$ by ϕ. Let $\{a_1, \ldots, a_s\}$ be the different symbols occurring in ϕ and let k be the largest integer occurring in ϕ. According to Lemma 3.1, every Σ-string is ϕ-equivalent to a string of the form $w = a_1^{m_1} \cdots a_s^{m_s}$ with $0 \leq m_i \leq k+1$ for each $i = 1, \ldots, s$. Note that there are $(k+1)^s$ such strings, which is a constant number, as it only depends on the input schema. For the following, the algorithm considers each such string w.

Fix such a string w such that $w \models \phi$. For each symbol c in $d_{\text{out}}(b)$, the number $\#_c(z_0 q_1[w] z_1 \cdots q_n[w] z_n)$ is equal to the linear sum

$$k_1^c \times \#_{a_1}(w) + \cdots + k_\ell^c \times \#_{a_\ell}(w) + k_{\ell+1}^c \times \#_{a_{\ell+1}}(w) + k_s^c \times \#_{a_s}(w) + k^c,$$

where $k^c = \#_c(z_0 \cdots z_n)$ and for each $i = 1, \ldots, s$, $k_i^c = \#_c(q_1[a_i] \cdots q_n[a_i])$.

In the remainder of the proof, we test if there exists a string $w' \equiv_\phi w$ such that $z_0 q_1[w'] z_1 \cdots q_n[w'] z_n \not\models d_{\text{out}}(b)$. Let $a_1, \ldots, a_\ell$ be the symbols that occur at least $k+1$ times in w and $a_{\ell+1}, \ldots, a_s$ be the symbols that occur at most k times in w, respectively. Then, deciding whether w' exists is equivalent to finding an integer solution to the variables $x_{a_1}, \ldots, x_{a_s}$ for the boolean combination of linear (in)equalities $\Phi = \Phi_1 \wedge \neg\Phi_2$, where

- Φ_1 states that $w' \equiv_\phi w$, that is,

$$\Phi_1 = \bigwedge_{i=1}^{\ell} (x_{a_i} > k) \wedge \bigwedge_{j=\ell+1}^{s} \big(x_{a_j} = \#_{a_j}(w)\big);$$

  and

- Φ_2 states that $z_0 q_1(w') z_1 \cdots q_n(w') z_n \models d_{\text{out}}(b)$, that is, Φ_2 is defined by replacing every occurrence of $c^{=i}$ or $c^{\geq i}$ in $d_{\text{out}}(b)$ by the equation

$$\sum_{j=1}^{s} (k_j^c \times x_{a_j}) + k^c = i$$

  or by

$$\sum_{j=1}^{s} (k_j^c \times x_{a_j}) + k^c \geq i,$$

  respectively.

In the above (in)equalities, x_{a_i}, $1 \leq i \leq s$, represents the number of occurrences of a_i in w'.

Finding a solution for Φ now consists of finding integer values for $x_{a_1}, \dots, x_{a_s}$ so that Φ evaluates to `true`. Corollary 3.6 shows that we can decide in PTIME whether such a solution for Φ exists. □

Theorem 5.3. *$TC^i[\mathcal{T}_{nd,bc}$, DTD(DFA)$]$ is* NLOGSPACE*-complete.*

Proof. In Theorem 5.8, we prove that the problem is NLOGSPACE-hard, even if both the input and output schemas are fixed. Hence, it remains to show that the problem is in NLOGSPACE.

Let us denote the tree transformation by $T = (Q_T, \Sigma, q_T^0, R_T)$ and the input and output DTDs by $(\Sigma, d_{\text{in}}, r)$ and d_{out}, respectively. We can assume that d_{in} is reduced.[1]

Then, the typechecking algorithm can be summarized as follows:

1. Guess a sequence of pairs $(q_0, a_0), (q_1, a_1), \dots, (q_n, a_n)$ in $Q_T \times \Sigma_{\text{in}}$, such that

   - $(q_0, a_0) = (q_T^0, r)$; and
   - for every pair (q_i, a_i), q_{i+1} occurs in $\text{rhs}(q_i, a_i)$ and a_{i+1} occurs in some string in $L(d_{\text{in}}(a_i))$.

   We only need to remember (q_n, a_n) as a result of this step.

2. Guess a node u in $\text{rhs}(q_n, a_n)$ — say that u is labeled with b — and test whether there exists a string $w \in d_{\text{in}}(a_n)$ such that $T^q(a_n(w))$ does not partly satisfy d_{out}.

The algorithm is successful if and only if w exists and, hence, the problem does not typecheck.

The first step is a straightforward reachability algorithm, which is in NLOGSPACE. It remains to show that the second step is in NLOGSPACE.

Let (q, a) be the pair (q_n, a_n) computed in step two. Let $d_{\text{out}}(b) = (Q_{\text{out}}, \Sigma, \delta_{\text{out}}, \{p_I\}, \{p_F\})$ be a DFA and let k be the copying bound of T. Let the concatenation of u's children be $z_0 q_1 z_1 \cdots q_\ell z_\ell$, where $\ell \leq k$. So we want to check whether there exists a string w such that $z_0 q_1[w] z_1 \cdots q_\ell[w] z_\ell$ is not accepted by $d_{\text{out}}(b)$. We guess w one symbol at a time and simulate in parallel ℓ copies of $d_{\text{out}}(b)$ and one copy of $d_{\text{in}}(a)$.

By $\hat{\delta}$ we denote the canonical extension of δ to strings in Σ^*. We start by guessing states $p_1, \dots, p_\ell$ of $d_{\text{out}}(b)$, where $p_1 = \hat{\delta}_{\text{out}}(p_I, z_0)$, and keep a copy of these on tape, to which we refer as $p'_1, \dots, p'_\ell$. Next, we keep on guessing symbols c of w, whereafter we replace each p_i by $\hat{\delta}_{\text{out}}(p_i, q_i(c))$. The input automaton obviously starts in its initial state and is simulated in the straightforward way.

The machine non-deterministically stops guessing, and checks whether, for each $i = 1, \dots, \ell - 1$, $\hat{\delta}_{\text{out}}(p_i, z_i) = p'_{i+1}$ and $\hat{\delta}_{\text{out}}(p_\ell, z_\ell) = p_F$. For the input automaton, it simply checks whether the current state is the final state. If the latter tests are positive, then the algorithm accepts, otherwise, it rejects.

We only keep $2\ell + 1$ states on tape, which is a constant number, so the algorithm runs in NLOGSPACE. □

[1] Reducing d_{in} would be PTIME-complete otherwise, see Corollary 3.17.

The following result is immediate, as the lowerbound proofs of Theorem 4.3(3) and Theorem 4.3(4) in Chapter 4 use an input schema that does not depend on the input.

Theorem 5.4.

(1) $TC^i[\mathcal{T}_{nd,bc}$, DTD(NFA)] is PSPACE*-complete; and*

(2) $TC^i[\mathcal{T}_{nd,uc}$, DTD(DFA)] is PSPACE*-complete.*

5.3 Non-deleting: Fixed Output Schema

The upper bounds carry over again from Chapter 4. Also, when the output DTD is a DTD(NFA), we can convert it into an equivalent DTD(DFA) in constant time. As the PTIME typechecking algorithm for TC[$\mathcal{T}_{\text{nd,bc}}$,DTD(DFA)] in Theorem 4.5 in Chapter 4 also works when the input DTD is a DTD(NFA), we have that the problem $\text{TC}^o[\mathcal{T}_{\text{nd,bc}}$,DTD(NFA)] is in PTIME. As the PTIME-hardness proof for TC[$\mathcal{T}_{\text{nd,bc}}$, DTD(DFA)] in Theorem 4.5 uses a fixed output schema, we immediately obtain the following.

Theorem 5.5. *$TC^o[\mathcal{T}_{nc,bc}, DTD(NFA)]$ is* PTIME*-complete.*

The lower bound in the presence of tree automata will be discussed in Section 5.4. The case requiring some real work is $\text{TC}^o[\mathcal{T}_{\text{nd,uc}}$, DTD(DFA)].

Theorem 5.6. *$TC^o[\mathcal{T}_{nd,uc}$, DTD(DFA)] is* PSPACE*-complete.*

Proof. In Chapter 4, it was shown that the problem is in PSPACE. We proceed by showing PSPACE-hardness.

We use a LOGSPACE reduction from CORRIDOR TILING, which is known to be PSPACE-complete (Theorem 3.23). Let $(T, V, H, \bar{\vartheta}, \bar{\beta}, n)$ be a tiling system, where $T = \{\vartheta_1, \ldots, \vartheta_k\}$ is the set of tiles, $V \subseteq T \times T$ and $H \subseteq T \times T$ are the sets of vertical and horizontal constraints respectively, and $\bar{\vartheta}$ and $\bar{\beta}$ are the top and bottom row, respectively. The tiling system has a solution if there is an $m \in \mathbb{N}$ such that the space $m \times n$ (m rows and n columns) can be correctly tiled with the additional requirement that the bottom and top row are $\bar{\beta}$ and $\bar{\vartheta}$, respectively.

We define the input DTD d_{in} over the alphabet $\Sigma := \{(i, \vartheta_j) \mid j \in \{1, \ldots, k\}, i \in \{1, \ldots, n\}\} \cup \{r\}$; r is the start symbol. Define

$$d_{\text{in}}(r) = \#\bar{\beta}\#\Big(\Sigma_1 \cdot \Sigma_2 \cdots \Sigma_n\#\Big)^*\bar{\vartheta}\#,$$

where we denote by Σ_i the set $\{(i, \vartheta_j) \mid j \in \{1, \ldots, k\}\}$. Here, # functions as a row separator. For all other alphabet symbols $a \in \Sigma$, $d_{\text{in}}(a) = \varepsilon$. So, d_{in} encodes all possible tilings that start and end with the bottom row $\bar{\beta}$ and the top row $\bar{\vartheta}$, respectively.

We now construct a tree transducer $B = (Q_B, \Sigma, q_B^0, R_B)$ and an output DTD d_{out} such that T has no correct corridor tiling if and only if B typechecks with respect to d_{in} and d_{out}. Intuitively, the transducer and the output DTD have to work together to determine errors in input tilings. There can only be two types of error: two tiles do not match horizontally or two tiles do not match vertically. The main difficulty is that the output DTD is fixed and can, therefore, *not* depend on the tiling system. The transducer is constructed in such a way that it prepares in parallel the verification for all horizontal and vertical constraints by the output schema. In particular, the transducer outputs specific symbols from a fixed set independent of the tiling system allowing the output schema to determine whether an error occurred.

The state set Q_B is partitioned into two sets, Q_{hor} and Q_{ver}:

- Q_{hor} is for the horizontal constraints: for every $i \in \{1, \dots, n-1\}$ and $\vartheta \in T$, $q_{i,\vartheta} \in Q_{\text{hor}}$ transforms the rows in the tiling such that it is possible to check that when position i carries a ϑ, position $i+1$ carries a ϑ' such that $(\vartheta, \vartheta') \in H$; and,

- Q_{ver} is for the vertical constraints: for every $i \in \{1, \dots, n\}$ and $\vartheta \in T$, $p_{i,\vartheta} \in Q_{\text{ver}}$ transforms the rows in the tiling such that it is possible to check that when position i carries a ϑ, the next row carries a ϑ' on position i such that $(\vartheta, \vartheta') \in V$.

The tree transducer B always starts its transformation with the rule

$$(q_B^0, r) \to r(w),$$

where w is the concatenation of all of the above states, separated by the delimiter \$. The other rules are of the following form:

- Horizontal constraints: for all $(j, \vartheta) \in \Sigma$ add the rule $(q_{i,\vartheta}, (j, \vartheta')) \to \alpha$ where $q_{i,\vartheta} \in Q_{\text{hor}}$ and

$$\alpha = \begin{cases} \texttt{trigger} & \text{if } j = i \text{ and } \vartheta = \vartheta' \\ \texttt{other} & \text{if } j = i \text{ and } \vartheta \neq \vartheta' \\ \texttt{ok} & \text{if } j = i+1 \text{ and } (\vartheta, \vartheta') \in H \\ \texttt{error} & \text{if } j = i+1 \text{ and } (\vartheta, \vartheta') \notin H \\ \texttt{other} & \text{if } j \neq i \text{ and } j \neq i+1 \end{cases}$$

  Finally, $(q_{i,\vartheta}, \#) \to \texttt{hor}$.

  The intuition is as follows: if the i-th position in a row is labeled with ϑ, then this position is transformed into `trigger`. Position $i+1$ is transformed to `ok` when it carries a tile that matches ϑ horizontally. Otherwise, it is transformed to `error`. All other symbols are transformed into an `other`.

  On a row, delimited by two `hor`-symbols, the output DFA rejects if and only if there is a `trigger` immediately followed by an `error`. When there is no `trigger`, then position i was not labeled with ϑ. So, the label `trigger` acts as a trigger for the output automaton.

- Vertical constraints: for all $(j, \vartheta) \in \Sigma$, add the rule $(p_{i,\vartheta}, (j, \vartheta')) \to \alpha$ where

$p_{i,\vartheta} \in Q_{\mathrm{ver}}$ and

$$\alpha = \begin{cases} \texttt{trigger1} & \text{if } (j,\vartheta') = (i,\vartheta) \text{ and } (\vartheta,\vartheta) \in V \\ \texttt{trigger2} & \text{if } (j,\vartheta') = (i,\vartheta) \text{ and } (\vartheta,\vartheta) \notin V \\ \texttt{ok} & \text{if } j = i,\ \vartheta \neq \vartheta', \text{ and } (\vartheta,\vartheta') \in V \\ \texttt{error} & \text{if } j = i,\ \vartheta \neq \vartheta', \text{ and } (\vartheta,\vartheta') \notin V \\ \texttt{other} & \text{if } j \neq i \end{cases}$$

Finally, $(p_{i,\vartheta}, \#) \to \texttt{ver}$.

The intuition is as follows: if the i-th position in a row is labeled with ϑ, then this position is transformed into `trigger1` when $(\vartheta,\vartheta) \in V$ and to `trigger2` when $(\vartheta,\vartheta) \notin V$. Here, both `trigger1` and `trigger2` act as a trigger for the output automaton: they mean that position i was labeled with ϑ. But no `trigger1` and `trigger2` can occur in the same transformed row as either $(\theta,\theta) \in V$ or $(\theta,\theta) \notin V$. When position i is labeled with $\vartheta' \neq \vartheta$, then we transform this position into `ok` when $(\vartheta,\vartheta') \in V$, and in `error` when $(\vartheta,\vartheta') \notin V$. All other positions are transformed into `other`.

The output DFA then works as follows. If a position is labeled `trigger1` then it rejects if there is a `trigger2` or a `error` occurring after the next `ver`. If a position is labeled `trigger2`, then it rejects if there is a `trigger2` or a `error` occurring after the next `ver`. Otherwise, it accepts that row.

By making use of the delimiters `ver` and `hor`, both above described automata can be combined into one automaton, taking care of the vertical and the horizontal constraints. This automaton resets to its initial state whenever it reads the delimiter symbol \$. Note that the output automaton is defined over the fixed alphabet $\{\texttt{trigger}, \texttt{trigger1}, \texttt{trigger2}, \texttt{error}, \texttt{ok}, \texttt{other}, \texttt{hor}, \texttt{ver}, \$\}$. □

Although the results in Chapter 4 were formulated in the context of variable schemas, the proof for bounded copying, non-deleting tree transducers with DTD(SL) and with DTD(DFA) schemas actually use a fixed output schema. We can therefore sharpen these results as follows.

Theorem 5.7.

(1) $TC^o[\mathcal{T}_{nd,bc}$, DTD(SL)] is coNP*-complete;*

(2) $TC^o[\mathcal{T}_{nd,bc}$, DTD(DFA)] is PTIME*-complete.*

5.4 Non-deleting: Fixed Input and Output Schema

We turn to the case where both input and output schemas are fixed. The following two theorems give us several new tractable cases.

Theorem 5.8.

(1) $TC^{io}[\mathcal{T}_{nd,bc}$, DTD(SL)] is NLOGSPACE*-complete.*

(2) $TC^{io}[\mathcal{T}_{nd,bc}$, DTD(DFA)] is NLOGSPACE*-complete.*

Proof. For both problems, membership in NLOGSPACE follows from Theorem 5.9. Indeed, every DTD(SL) can be rewritten into an equivalent DTD(NFA) in constant time as the input and output schemas are fixed.

We proceed by showing NLOGSPACE-hardness. We say that an NFA $N = (Q_N, \Sigma, \delta_N, I_N, F_N)$ has *degree of nondeterminism 2* if *(i)* I_N has at most two elements and *(ii)* for every $q \in Q_N$ and $a \in \Sigma$, the set $\delta_N(q, a)$ has at most two elements. We give a LOGSPACE reduction from the emptiness problem of an NFA with alphabet $\{0, 1\}$ and a degree of nondeterminism 2 to the typechecking problem. According to Lemma 3.14, this problem is NLOGSPACE-hard. Intuitively, the input DTD will define all possible strings over alphabet $\{0, 1\}$. The tree transducer simulates the NFA and outputs "accept" if a computation branch accepts, and "error" if a computation branch rejects. The output DTD defines trees where all leaves are labeled with "error".

More concretely, let $N = (Q_N, \{0, 1\}, \delta_N, \{q_N^0\}, F_N)$ be an NFA with degree of nondeterminism 2. The input DTD $(\{0, 1, \#\}, d_{\text{in}}, r)$ defines all unary trees, where the unique leaf is labeled with a special marker #. That is, $d_{\text{in}}(r) = d_{\text{in}}(0) = d_{\text{in}}(1) = (0 + 1 + \#)$ and $d_{\text{in}}(\#) = \varepsilon$. Note that these languages can be defined by SL-formulas or DFAs which are sufficiently small for our purpose.

Given a tree $t = r(a_1(\cdots(a_n(\#))\cdots))$, the tree transducer will simulate every computation of N on the string $a_1 \cdots a_n$. The tree transducer $T = (Q_T, \{r, \#, 0, 1, \text{error}, \text{accept}\}, q_T^0, R_T)$ simulates N's nondeterminism by copying the remainder of the input twice in every step. Formally, Q_T is the union of $\{q_T^0\}$ and Q_N, and R_T contains the following rules:

- $(q_T^0, r) \to r(q_N^0)$. This rule puts r as the root symbol of the output tree and starts the simulation of N.
- $(q_N, a) \to a(q_N^1, q_N^2)$, where $q_N \in Q_N$, $a \in \{0, 1\}$ and $\delta_N(q_N, a) = \{q_N^1, q_N^2\}$. This rule does the actual simulation of N. By continuing in both states q_N^1 and q_N^2, we simulate *all* possible computations of N.
- $(q_N, a) \to \text{error}$ if $\delta_N(q_N, a) = \emptyset$. If N rejects, we output the symbol "error".
- $(q_N, \#) \to \text{error}$ for $q_N \notin F_N$; and
- $(q_N, \#) \to \text{accept}$ for $q_N \in F_N$. These last two rules verify whether N is in an accepting state after reading the entire input string.

Notice that T outputs the symbol "error" (respectively "accept") if and only if a computation branch of N rejects (respectively accepts).

The output of T is always a tree in which only the symbols "error" and "accept" occur at the leaves. The output DTD then needs to verify that only the symbol "error" occurs at the leaves. Formally, $d_{\text{out}}(r) = d_{\text{out}}(0) = d_{\text{out}}(1) = \{0, 1, \text{error}\}^+$ and $d_{\text{out}}(\text{error}) = \varepsilon$. Again, these languages can be defined by sufficiently small SL-formulas or DFAs.

It is easy to see that the reduction only requires logarithmic space. □

Theorem 5.9. *$TC^{io}[\mathcal{T}_{nd,uc}$, DTD(NFA)] is* NLOGSPACE-*complete.*

Proof. The NLOGSPACE-hardness of the problem follows from Theorem 5.8, where it shown that the problem is already NLOGSPACE-hard when DTD(DFA)s are used as input and output schema.

We show that the problem is also in NLOGSPACE. Thereto, let $T = (Q_T, \Sigma, q_T^0, R_T)$ be the tree transducer, and let $(\Sigma, d_{\text{in}}, r)$ and d_{out} be the input and output DTDs, respectively. As both d_{in} and d_{out} are fixed, we can assume without loss of generality that they are reduced.[2] For the same reason, we can also assume that the NFAs in d_{in} and d_{out} are determinized.

We guess a sequence of state-label pairs $(p_0, a_0), (p_1, a_1), \ldots, (p_n, a_n)$ where $n < |Q_T||\Sigma|$ such that

- $(p_0, a_0) = (q_T^0, r)$; and
- for every pair (p_i, a_i), p_{i+1} occurs in $\text{rhs}(p_i, a_i)$ and a_{i+1} occurs in some string in $L(d_{\text{in}}(a_i))$.

Each time we guess a new pair in this sequence, we forget the previous one, so that we only keep a state, an alphabet symbol, a counter, and the binary representation of $|Q_T||\Sigma|$ on tape.

For simplicity, we write (p_n, a_n) as (p, a) in the remainder of the proof. We guess a node $u \in \text{Nodes}(\text{rhs}(p, a))$. Let $b = \text{lab}^{\text{rhs}(p,a)}(u)$ and let $z_0 q_1 z_1 \cdots q_k z_k$ be the concatenation of u's children, where every $z_0, \ldots, z_k \in \Sigma^*$ and every $q_1, \ldots, q_k \in Q_T$, then we want to check whether there exists a string $w \in d_{\text{in}}(a)$ such that $z_0 q_1[w] z_1 \cdots q_k[w] z_k$ is not accepted by $d_{\text{out}}(b)$. Recall from Section 2.3 that, for a state $q \in Q_T$, we denote by $q[w]$ the homomorphic extension of $q[c]$ for $c \in \Sigma$, which is $\text{top}(\text{rhs}(q, c)))$ in the case of non-deleting tree transducers. We could do this by guessing w one symbol at a time and simulating k copies of $d_{\text{out}}(b)$ and one copy of $d_{\text{in}}(a)$ in parallel, like in the proof of Theorem 5.3. However, as k is not fixed, the algorithm would use superlogarithmic space.

So, we need a different approach. To this end, let $A = (Q_{\text{in}}, \Sigma, \delta_{\text{in}}, q_{\text{in}}^0, F_{\text{in}})$ and $B = (Q_{\text{out}}, \Sigma, \delta_{\text{out}}, q_{\text{out}}^0, F_{\text{out}})$ be DFAs accepting $L(d_{\text{in}}(a))$ and $L(d_{\text{out}}(b))$, respectively. To every $q \in Q_T$, we associate a function

$$f_q : Q_{\text{out}} \times \Sigma \to Q_{\text{out}} : (p', c) \mapsto \hat{\delta}_{\text{out}}(p', q[c]),$$

where $\hat{\delta}_{\text{out}}$ denotes the canonical extension of δ_{out} to strings in Σ^*. Note that there are maximally $|Q_{\text{out}}|^{|Q_{\text{out}}||\Sigma|}$ such functions. Let K be the cardinality of the set $\{f_q \mid q \in Q_T\}$. Hence, K is bounded from above by $|Q_{\text{out}}|^{|Q_{\text{out}}||\Sigma|}$, which is a constant (with respect to the input). Let $f_1, \ldots, f_K$ an arbitrary enumeration of $\{f_q \mid q \in Q_T\}$.

The typechecking algorithm continues as follows. We start by writing the $(1 + K \cdot |Q_{\text{out}}|)$-tuple $(q_{\text{in}}^0, q_1', \ldots, q_{|Q_{\text{out}}|}', \ldots, q_1', \ldots, q_{|Q_{\text{out}}|}')$ on tape, where Q_{out} is the set $\{q_1', \ldots, q_{|Q_{\text{out}}|}'\}$. We will refer to this tuple as the tuple $\bar{p} := (p_0', \ldots, p_{K \cdot |Q_{\text{out}}|}')$. We explain how we update $\bar{p}$ when guessing w symbol by symbol. Every time when we guess the next symbol c of w, we overwrite the tuple $\bar{p}$ by

$$\big(\delta_{\text{in}}(p_0', c), f_1(p_1', c), \ldots, f_1(p_{|Q_{\text{out}}|}', c), \ldots \\ \ldots, f_K(p_{(K-1)\cdot|Q_{\text{out}}|+1}', c), \ldots, f_K(p_{K \cdot |Q_{\text{out}}|}', c)\big).$$

[2] In general, reducing a DTD(NFA) is PTIME-complete (Section 2.2).

Notice that there are at most $|Q_{\text{in}}| \cdot K \cdot |Q_{\text{out}}|^2$ different $(K \cdot |Q_{\text{out}}| + 1)$-tuples of this form. We nondeterministically determine when we stop guessing symbols of w.

It now remains to verify whether w was indeed a string such that $w \in d_{\text{in}}(a)$ and $z_0 q_1[w] z_1 \cdots q_k[w] z_k \notin d_{\text{out}}(b)$. The former condition is easy to test: we simply have to test whether $p'_0 \in F_{\text{in}}$. To test the latter condition, we read the string $z_0 q_1 z_1 \cdots q_k z_k$ from left to right while performing the following tests. We keep a state of $d_{\text{out}}(b)$ in memory and refer to it as the "current state".

1. The initial current state is q^0_{out}.

2. If the current state is p' and we read z_j, then we change the current state to $\hat{\delta}_{\text{out}}(p', z_j)$.

3. If the current state is p' and we read q_j, then we change the current state to p'_i in $\bar{p}$, where for i, the following condition holds. Let $\ell, m = 1, \ldots, K \cdot |Q_{\text{out}}|$ be the smallest integers such that

   - $p' = q'_\ell$ in Q_{out}, and
   - $f_{q_j} = f_m$.

   Then $i = (m-1)K + \ell$.

   Note that deciding whether $p' = q'_\ell$ and $f_{q_j} = f_m$ can be done deterministically in logarithmic space, as the output schema is fixed. Consequently, i can also be computed in constant time and space.

4. We stop and accept if the current state is a non-accepting state after reading z_k. □

The following theorem immediately follows from the proof of Theorem 4.5(2), as we used a fixed input and output schema in the reduction.

Theorem 5.10. *$TC^{io}[\mathcal{T}_{nd,bc}$, $DTA(DFA)]$ is* EXPTIME-*complete.*

6 Frontiers of Tractability

The previous chapters have given us a fairly complete overview of the complexity of the typechecking problem for simple XSLT transformations. However, the settings in which we have found a polynomial time typechecking algorithm seem very restrictive, especially since they exclude every form of deletion in the transformation. As illustrated by Example 2.15, deletion of an arbitrary number of interior nodes is quite typical for filtering transformations. Indeed, many simple transformations select specific parts of the input while ignoring the non-interesting ones.

Therefore, we investigate in this chapter larger and more flexible classes for which the complexity of the typechecking problem remains in PTIME. Additionally, we also want to identify the frontier where these scenarios become intractable.[1] Hence, our work sheds light on when to use fast complete algorithms and when to reside to sound but incomplete ones.

We first investigate deletion in the setting where DTDs use deterministic finite automata (DFAs) to define right-hand sides of rules and transducers can only make a bounded number of copies of nodes in the input tree. By proving a general lemma which quantifies the combined effect of copying and deletion on the complexity of typechecking, we derive conditions under which typechecking becomes tractable. In particular, these conditions allow arbitrary deletion when no copying occurs (like in Example 2.15), but at the same time permit limited copying for those rules that only delete in a limited fashion. This result is relevant in practice as in common filtering transformations arbitrary deletion almost never occurs together with copying.

We then show that the present setting cannot be enlarged without increasing the complexity. In particular, we show that combining a slight relaxation of the limited deletion restriction with copying the input only twice makes typechecking intractable. Finally, we briefly examine tree automata to define schemas and show that in the case

[1] That is, NP- or CONP-hard, assuming that the latter classes are different from PTIME.

of deterministic tree automata, no copying but arbitrary deletion, we get a PTIME algorithm.

As an alternative to deletion, one can skip nodes in the input tree by adding XPath expressions to the transformation language. In the case where DTDs use DFAs, we obtain a tractable fragment by translating the transformation language to transducers without XPath expressions. As XPath containment in the presence of DTDs [NS03, Woo03] can easily be reduced to the typechecking problem, lower bounds establishing intractability readily follow for XPath fragments containing filters and disjunction.

The first PTIME results still rely on a uniform bound on the number of copies a rule of the transducer can make. Although this number will always be fairly small in practice, it would still be more elegant to have an algorithm which is tractable for any transducer in a specific class. Thereto, we have to restrict the schema languages. In fact, we show that only for very weak DTDs, those where all regular expressions are concatenations of symbols a and a^+ (which we call RE$^+$ expressions), typechecking becomes tractable, and that obvious extensions of such expressions make the problem at least coNP-hard. So, the price for arbitrary deletion and copying is very high.

6.1 Deletion, Bounded Copying, and DFAs

Although deletion has an enormous impact on the complexity of typechecking, as is exemplified by the first two rows of Tables 4.1 and 5.1, more often than not, the ability to skip nodes in the input tree is critical. Indeed, many simple transformations like the ones in Example 2.15 select specific parts of the input while deleting the non interesting ones. Moreover, such deletion can be unbounded. That is, the number of deleted nodes on a path depends only on the input tree and not on the schema.

Since the typechecking problem is immediately intractable for DTD(NFA)s and transducers with unbounded copying, we focus in this section on DTD(DFA)s and on bounded copying transducers. We prove a general lemma which quantifies the combined effect of copying and deletion on the complexity of typechecking. From this lemma we then derive conditions under which typechecking becomes tractable. Interestingly, these conditions allow arbitrary deletion when no copying occurs, but at the same time permit bounded copying for those rules that only delete in a bounded fashion. We further show that these conditions cannot be relaxed without increasing the complexity. Finally, we discuss typechecking in the context of schemas represented by deterministic tree automata.

6.1.1 A Practical Case

We start by introducing some terminology to describe the effect of deleting states. Let $T = (Q_T, \Sigma, q_0^T, R_T)$ be a transducer. A *deletion path* is a sequence of states $q_1, \ldots, q_n$ in Q_T such that q_i occurs in $\mathrm{top}(\mathrm{rhs}(q_{i-1}, a_{i-1}))$ for all $i = 2, \ldots, n$ and some $a_2, \ldots, a_n \in \Sigma$. Note that every $q_1, \ldots, q_{n-1}$ is a deleting state as defined in Section 2.3.

The *deletion width of a state* $q \in Q_T$ is the maximum number of states in $\mathrm{top}(\mathrm{rhs}(q, a))$ for all $a \in \Sigma$. We denote the deletion width of q by $\mathrm{dw}(q)$. The *width of a deletion path* $q_1, \ldots, q_n$ is the product $\prod_{i=1}^{n-1} \mathrm{dw}(q_i)$. Note that we do not take the deletion width of q_n into account as it may be zero. We say that T has *deletion path width* K if every deletion path has width smaller than or equal to K.

Example 6.1. Let T be the transducer consisting of the following rules:

$$\begin{array}{ll}
(q_0^T, a) \to a(q_1\, q_5) & \\
(q_1, a) \to q_2\, a\, q_2\, a & (q_5, a) \to q_6\, a\, a\, q_6 \\
(q_2, a) \to a\, q_3\, q_3\, a\, q_3 & (q_6, a) \to q_7\, q_7 \\
(q_3, a) \to q_4 & (q_7, a) \to a\, q_8\, a \\
(q_4, a) \to a & (q_8, a) \to a\, a\, q_7
\end{array}$$

The deletion widths of the states are given as follows:

state	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
deletion width	2	3	1	0	2	2	1	1

The sequences q_1, q_2, q_3, q_4 and q_5, q_6, q_7, q_8, q_7 are examples of deletion paths in T. The former has deletion width six and the latter has deletion width four. Note that the deletion path

$$q_5, q_6, q_7, q_8, q_7, q_8, q_7, q_8$$

also has deletion width four. The reason is that the deletion width of q_7 and q_8 is one. Would there be a rule $(q_7, b) \to q_8 q_8$ then paths of arbitrary large deletion width could be constructed.

Notice that the deletion path width of T is six. We discuss a general algorithm to compute the deletion path width of a tree transducer in the proof of Proposition 6.5. ◇

A deleting state is *recursively deleting* if it occurs twice in some deletion path; otherwise, it is said to be *non-recursively deleting.* The *deletion depth* of a state q is the maximum length of a deletion path in which it occurs. When no such maximum exists, we say that the state has *unbounded deletion depth.* In particular, all recursively deleting states have unbounded deletion depth.

By $\mathcal{T}_{\mathrm{trac}}^{C,K}$, we denote the class of transducers with copying width at most C and deletion path width at most K. When the actual values of C and K are not important, we simply write $\mathcal{T}_{\mathrm{trac}}$ instead of $\mathcal{T}_{\mathrm{trac}}^{C,K}$.

Note that a class $\mathcal{T}_{\mathrm{trac}}^{C,K}$ allows recursive deletion, but only for those states that do not copy at the same time. Should such a state occur, the width of deletion paths cannot be bounded. So, if a state of a $\mathcal{T}_{\mathrm{trac}}^{C,K}$-transducer is recursively deleting then every associated rhs is of the form *hpg* where p is a state and h and g are hedges containing no states on their top level and with at most C occurrences of states in every sequence of siblings. When a state is non-recursively deleting, then simultaneous copying and deletion is allowed but only in a bounded fashion. That is, every deletion path containing that state is of deletion width at most K.

Example 6.2. The first transducer in Example 2.15 belongs to $\mathcal{T}_{\mathrm{trac}}^{1,1}$ while the second is in $\mathcal{T}_{\mathrm{trac}}^{2,1}$. The transducer of Example 6.1 is in $\mathcal{T}_{\mathrm{trac}}^{3,6}$. ◇

The next lemma provides a detailed analysis of the complexity of typechecking in terms of copying and deletion power. Its proof is a non-trivial generalization from non-deleting to deleting transducers of the reduction from TC[$\mathcal{T}_{\text{nd,bc}}$,DTD(DFA)] to the emptiness test of unranked tree automata in Theorem 4.5(4) in Chapter 4, followed by an analysis of the size of the obtained automaton.

We use the following terminology in the proof of Lemma 6.3. For a tree t and a node $u \in \text{Nodes}(t)$, we denote by t/u the subtree of t rooted at u. For a hedge h and a DTD (d, s_d), we say that h *partly satisfies* d if for every $u \in \text{Nodes}(h)$, $\text{lab}^h(u1) \cdots \text{lab}^h(un) \in L(d(\text{lab}^h(u)))$ where u has n children. Note that there is no requirement on the root nodes of the trees in h. Hence, the term partly.

Lemma 6.3. *$TC[\mathcal{T}^{C,K}_{trac}, DTD(DFA)]$ can be decided in*

$$\mathcal{O}\big((|d_{in}||T|^{C\times K}|d_{out}|^{C\times K})^{\alpha}\big) \textit{ time},$$

where $|d_{in}|$ and $|d_{out}|$ are the sizes of the input and output schema, respectively; $|T|$ is the size of the tree transducer T; and α is a constant.

Proof. Let $T = (Q_T, \Sigma, q^0_T, R_T) \in \mathcal{T}^{C,K}_{\text{trac}}$ be a tree transducer. Let d_{in} and d_{out} be the input and output DTDs, respectively. We construct an NTA(NFA) B such that $L(B) = \{t \in L(d_{\text{in}}) \mid T(t) \notin L(d_{\text{out}})\}$. Thus, B accepts all counterexample trees. Therefore, $L(B) = \emptyset$ if and only if T typechecks with respect to d_{in} and d_{out}. We argue that B can be constructed in time

$$\mathcal{O}((|d_{\text{in}}||T|^{C\times K}|d_{\text{out}}|^{C\times K})^{\beta})$$

for a constant β. As the emptiness problem of NTA(NFA)s is in PTIME (Proposition 3.18), the complexity of the typechecking problem is

$$\mathcal{O}\big((|d_{\text{in}}||T|^{C\times K}|d_{\text{out}}|^{C\times K})^{\alpha}\big),$$

for a constant α.

Behavior of B. A tree automaton can easily verify that the input tree satisfies d_{in}. To check that the translated tree violates the output schema d_{out}, B non-deterministically locates a node v in the input tree generating a subtree

$$\sigma(a_1(s_1) \cdots a_m(s_m))$$

such that $a_1 \cdots a_m \notin d_{\text{out}}(\sigma)$. We explain how the latter can be verified. Thereto, let $a(t_1 \cdots t_n)$ be the tree rooted at v. Assume that T processes v in state q and that $\text{rhs}(q, a)$ contains the subtree $\sigma(z_0 q_1 z_1 \cdots z_{k-1} q_k z_k)$, where $z_0, \ldots, z_k \in \Sigma^*$ and $q_1, \ldots, q_k \in Q_T$. Then, B needs to simulate the complement of the DFA D for $d_{\text{out}}(\sigma)$ on the string

$$z_0 \ \text{top}\big(T^{q_1}(t_1) \cdots T^{q_1}(t_n)\big) \ z_1 \cdots z_{k-1} \ \text{top}\big(T^{q_k}(t_1) \cdots T^{q_k}(t_n)\big) \ z_k.$$

As the strings $\text{top}(T^{q_i}(t_i))$ depend on the subtrees t_i rooted at v, B cannot simply run D. Instead, for each t_i, the automaton B guesses k pairs of states $(p^1_{i,1}, p^1_{i,2}), \ldots, (p^k_{i,1}, p^k_{i,2})$ of D, and verifies later that indeed $\text{top}(T^{q_j}(t_i))$ takes D from state $p^j_{i,1}$ to state $p^j_{i,2}$. At present, B can only verify that

1. z_0 takes D from its initial state to $p^1_{1,1}$;

2. z_k takes D from $p^k_{n,2}$ to an accepting state;

3. for each $j = 1, \ldots, k-1$, z_j takes D from $p^j_{n,2}$ to $p^{j+1}_{1,1}$; and

4. for each $i = 1, \ldots, n-1$ and $j = 1, \ldots, k$, we have that $p^j_{i,2} = p^j_{i+1,1}$.

Note that for this step, B needs to remember at most $2C$ states of D for each subtree. We briefly sketch how B can verify that the string $\text{top}(T^{q_j}(t_i))$ takes D from state $p^j_{i,1}$ to state $p^j_{i,2}$. If $\text{rhs}(q_j, \sigma_i)$, where σ_i is the root of t_i, contains no deleting states, then $\text{top}(T^{q_j}(t_i))$ only depends on $\text{rhs}(q_j, \sigma_i)$ and not on t_i and B can simply run D. When $\text{rhs}(q_j, \sigma_i)$ contains ℓ deleting states, then we just need to guess ℓ new pairs of states $(p_{i,1}, p_{i,2})$ and defer verification to the children of the present node. As long as the transducer deletes, new pairs of states are guessed. As K is an upper bound for this number, $C \times K$ is the maximum number of pairs that need to be remembered at all time to check whether for every i, $\text{top}(T^{q_j}(t_i))$ takes D from state $p^j_{i,1}$ to state $p^j_{i,2}$. Note that we also allow recursively deleting states but as these cannot copy, they do not increase the number of pairs of states B has to guess.

Construction. Let $T = (Q_T, \Sigma, q^0_T, R_T)$ and let for each $a \in \Sigma$, $A_a = (Q_a, \Sigma, \delta_a, I_a, F_a)$ be the DFA for which $d_{\text{out}}(a) = L(A_a)$. Without loss of generality, we assume that the sets Q_a are pairwise disjoint. Set $M = C \times K$. Intuitively, M is an upper bound on the number of states of some A_a that B needs to remember. This will become clear later. Next, we define the tree automaton $B = (Q_B, \Sigma, F_B, \delta_B)$. The set of states Q_B is the union of the following sets:

- Σ,

- $\{(a, q) \mid a \in \Sigma, q \in Q_T\}$,

- $\{(a, q, \text{check}) \mid a \in \Sigma, q \in Q_T\}$, and

- $\bigcup_{i=1}^{M} \{(a, (q_1, \ell^b_1, r^b_1), \ldots, (q_M, \ell^b_M, r^b_M)) \mid$ $(q_1, \ell^b_1, r^b_1) \cdots (q_M, \ell^b_M, r^b_M) \in \{(Q_T \times Q_a \times Q_a)^i \cdot (\#, \#, \#)^{M-i} \mid i = 1, \ldots, M\}$, $a, b \in \Sigma\}$, where $\# \notin Q_T \cup \bigcup_{a \in \Sigma} Q_a$.

Note that the size of Q^B is $\mathcal{O}(|\Sigma||Q_T|^M |d_{\text{out}}|^{2M})$. We explain the intuition behind these states. When there is an accepting run on a tree t, then a node v labeled with a state of the form

$$a, (a, q), (a, q, \text{check}), \text{ or } (a, (q_1, \ell^b_1, r^b_1), \ldots, (q_M, \ell^b_M, r^b_M))$$

has the following meaning:

a**:** v is labeled with a and the subtree rooted at v partly satisfies d_{in}.

(a, q)**:** same as in previous case with the following two additions: (1) v is processed by T in state q; and, (2) a descendant of v will produce a tree that does not partly satisfy d_{out}.

(a, q, check)**:** same as the previous case only now v itself will produce a tree that does not partly satisfy d_{out}.

$(a, (q_1, \ell_1^b, r_1^b), \ldots, (q_M, \ell_M^b, r_M^b))$**:** v is labeled with a and the subtree rooted at v partly satisfies d_{in}. Furthermore, v is processed by T in states $q_1, \ldots, q_j$, where j is maximal such that $q^j \neq \#$, and v is a descendant of the node labeled with (a, q, check). The triple (q_i, ℓ_i^b, r_i^b), $i \leq j$, indicates that $\text{top}(T^{q_i}(t/v))$ takes A_b from state ℓ_i^b to r_i^b.

The set of final states is $F_B := \{(s_{\text{in}}, q_T^0)\}$ where s_{in} is the start symbol of d_{in}.

The transition function is defined as follows: for all $a, b \in \Sigma$ with $a \neq b$ and $q \in Q_T$

1. we have

$$\begin{aligned}
&\delta_B(a, b) = \emptyset;\\
&\delta_B((a, q), b) = \emptyset;\\
&\delta_B((a, q, \text{check}), b) = \emptyset; \text{ and}\\
&\delta_B((a, (q_1, \ell_1^c, r_1^c), \ldots, (q_M, \ell_M^c, r_M^c)), b) = \emptyset.
\end{aligned}$$

2. $\delta_B(a, a) = d_{\text{in}}(a)$ and $\delta_B((a, q), a)$ consists of those strings $a_1 \cdots a_n$ such that there is precisely one index $j \in \{1, \ldots, n\}$ for which $a_j = (b, p)$ or $a_j = (b, p, \text{check})$ where p occurs in $\text{rhs}(q, a)$ and for all $i \neq j$, $a_i \in \Sigma$; further, $a_1 \cdots a_{j-1} b a_{j+1} \cdots a_n \in L(d_{\text{in}}(a))$. Note that $\delta_B((a, q), a)$ is defined in such a way that it ensures that all subtrees partly satisfy d_{in} and that at least one subtree will generate a violation of d_{out}. Clearly, $\delta_B(a, a)$ and $\delta_B((a, q), a)$ can be represented by DFAs whose size is at most quadratic in the size of the input DTD plus the size of the transducer.

3. We define $\delta_B((a, q, \text{check}), a)$. Let u be an arbitrary node in $\text{rhs}(q, a)$ labeled with $b \in \Sigma$ and let $\overline{A_b} = (Q_b, \Sigma, \delta_b, I_b, Q_b - F_b)$. Let $s_u = z_0 q_1 z_1 \cdots z_{k-1} q_k z_k$ be the concatenation of the labels of the children of u, where every $z_i \in \Sigma^*$ and $q_i \in Q_T$. Intuitively, if v is the node in the input tree t labeled with (a, q, check), and v has n children, then we want to check here whether the string

$$\begin{aligned}
s = z_0 \text{top}(T^{q_1}(t/v1)) \cdots \text{top}(T^{q_1}(t/vn)) z_1 \cdots&\\
\cdots z_{k-1} \text{top}(T^{q_k}(t/v1)) \cdots \text{top}(T^{q_k}(t/vn)) z_k&
\end{aligned}$$

is accepted by $\overline{A_b}$ (or, equivalently, rejected by A_b). Of course, at v the automaton B does not know the strings $\text{top}(T^{q_j}(t/vi))$. Instead, B guesses $k \cdot n$ pairs of states $(\ell_{j,i}, r_{j,i})$ of $\overline{A_b}$, where $i = 1, \ldots, n$ and $j = 1, \ldots, k$, such that $\overline{A_b}$ accepts the string

$$\begin{aligned}
z_0 (\ell_{1,1}, r_{1,1})(\ell_{1,2}, r_{1,2}) \cdots (\ell_{1,n}, r_{1,n}) z_1 \cdots&\\
\cdots z_{k-1} (\ell_{k,1}, r_{k,1})(\ell_{k,2}, r_{k,2}) \cdots (\ell_{k,n}, r_{k,n}) z_k&
\end{aligned}$$

where the behavior of $\overline{A_b}$ is modified as follows: when $\overline{A_b}$ reaches $(\ell_{j,i}, r_{j,i})$ in state $\ell_{j,i}$, it moves to state $r_{j,i}$, otherwise it rejects. So, B guesses the input-output behavior $(\ell_{j,i}, r_{j,i})$ of $\overline{A_b}$ at every string $\text{top}(T^{q_j}(t/vi))$. These guesses should then be verified further down in the tree.

Formally, let for $I, F \subseteq Q_b$, $N_b(I,F) = (Q_b, \Sigma \cup (Q_b \times Q_b), \delta_{N_b}, I, F)$ be the DFA that behaves the same way as $\overline{A_b}$, but when it reads a symbol (q_1', q_2') in state q_1' it immediately jumps to state q_2', and rejects otherwise. The parameterization of the initial and final states of N_b will be needed in bullet (4).

We define $\delta_B((a, q, \text{check}), a)$ as the union of all sets $R(u)$ where u is a node in $\text{rhs}(q,a)$ and each $R(u)$ is defined as follows:

$$\big(a_1, (q_1, \ell^b_{1,1}, r^b_{1,1}), \ldots, (q_M, \ell^b_{M,1}, r^b_{M,1})\big) \cdots$$
$$\cdots \big(a_n, (q_1, \ell^b_{1,n}, r^b_{1,n}), \ldots, (q_M, \ell^b_{M,n}, r^b_{M,n})\big)$$

such that

- $a_1 \cdots a_n \in d_{\text{in}}(a)$;
- the string
$$z_0(\ell^b_{1,1}, r^b_{1,1}) \cdots (\ell^b_{1,n}, r^b_{1,n}) z_1 \cdots z_{k-1}(\ell^b_{k,1}, r^b_{k,1}) \cdots (\ell^b_{k,n}, r^b_{k,n}) z_k$$
is accepted by $N_b(I_b, Q_b - F_b)$;
- $q_1, \ldots, q_k$ are the states as occurring in $s_u = z_0 q_1 z_1 \cdots q_k z_k$; and
- for $i = k+1, \ldots, M$, $j = 1, \ldots, n$: $(q_i, \ell^b_{i,j}, r^b_{i,j}) = (\#, \#, \#)$.

We compute an upper bound for the size of the NFA for $\delta_B((a, q, \text{check}), a)$. The alphabet size of $\delta_B((a, q, \text{check}), a)$ is bounded by $|\Sigma||Q_T|^C|Q_{\text{out}}|^{2C}$, where $Q_{\text{out}} = \bigcup_{b \in \Sigma} Q_b$. Further, for a node u in $\text{rhs}(q,a)$ labeled with b, $R(u)$ can be accepted by a DFA that simulates in parallel one copy of $d_{\text{in}}(a)$ and at most C copies of $\overline{A_b}$. Note that once u is chosen, the states $q_1, \ldots, q_k$ in $R(u)$, with $k \leq C$, are fixed. Hence, $\delta_B((a, q, \text{check}), a)$ can be represented as a union of $|\text{rhs}(q,a)|$ DFAs with $|d_{\text{in}}(a)||d_{\text{out}}|^C$ states, which bounds the total size of the NFA representing $\delta_B((a, q, \text{check}), a)$ by

$$|\Sigma||Q_T|^C|Q_{\text{out}}|^{2C} \times |\text{rhs}(q,a)||d_{\text{in}}(a)||d_{\text{out}}|^C.$$

4. Finally, we define $\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$. Let m be the smallest index such that for all $m' > m$, $p^{m'} = \#$. Intuitively, when B arrives at a node v in state $(a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \underline{\ell^b_M, r^b_M}))$, then it should verify that for every $i = 1, \ldots, m$, $\text{top}(T^{p_i}(t/v))$ takes $\overline{A_b}$ from ℓ^b_i to r^b_i. For every $i = 1, \ldots, m$, let $\text{top}(\text{rhs}(p_i, a))$ be of the form $z_{i,0} q_{i,1} z_{i,1} \cdots q_{i,k_i} z_{i,k_i}$ where $z_{i,j} \in \Sigma^*$ and $q_{i,j} \in Q_T$. When $k_i > 0$ B has to replace (p_i, ℓ^b_i, r^b_i) with a new sequence in $(Q_T \times A_b \times A_b)^*$.

   So, $\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$ accepts the strings

$$\big(a_1, (q_1, \ell^b_{1,1}, r^b_{1,1}), \ldots, (q_M, \ell^b_{M,1}, r^b_{M,1})\big) \cdots$$
$$\cdots \big(a_n, (q_1, \ell^b_{1,n}, r^b_{1,n}), \ldots, (q_M, \ell^b_{M,n}, r^b_{M,n})\big)$$

   such that

- $a_1 \cdots a_n \in d_{\text{in}}(a)$; and
- for all $i \leq m$, $q_{j+1} \cdots q_{j+k_i} = q_{i,1} \cdots q_{i,k_i}$, where $j = \Sigma_{x=1}^{i-1} k_x$; and
- for all $i \leq m$, the string

$$z_{i,0}(\ell^b_{j+1,1}, r^b_{j+1,1}) \cdots (\ell^b_{j+1,n}, r^b_{j+1,n}) z_{i,1} \cdots \\ \cdots z_{i,k_i-1}(\ell^b_{j+k_i,1}, r^b_{j+k_i,1}) \cdots (\ell^b_{j+k_i,n}, r^b_{j+k_i,n}) z_{i,k_i}$$

  is accepted by $N_b(\{\ell^b_i\}, \{r^b_i\})$, where $j = \Sigma_{x=1}^{i-1} k_x$; and,
- for $i = (1 + \Sigma_{x=1}^m k_x), \ldots, M$, $j = 1, \ldots, n$: $(q_i, \ell_{i,j}, r_{i,j}) = (\#, \#, \#)$.

We need to argue that at all times, $\Sigma_{x=1}^m k_x \leq M$. Let for an input tree t, $v \in \text{Nodes}(t)$ be the node that is visited in state (a, q, check) by B and let $u \in \text{rhs}(q, a)$ be the node selected in step (3), labeled with b. Assume first that q is a state with bounded deletion depth. To produce the string s that must be tested for membership in $\overline{A_b}$, T visits v's children in at most C states. Let $q, q^1, \ldots, q^\ell$ be an arbitrary deletion path in T, and let for each q^i, D_i be the deletion width of q_i. Then, the nodes at depth i in t/v are visited by T in at most $C \cdot D_1 \cdots D_{i-1}$ states of T. So, every node in t/v is visited by T in at most $C \times K = M$ states to produce s. Hence, M is an upper bound for $\Sigma_{x=1}^m k_x$. When q has unbounded deletion depth, only states that do not copy can occur multiple times. These cannot increase the number of states B needs to remember.

We compute an upper bound for the size of

$$\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a).$$

The alphabet size of $\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$ is bounded from above by $|\Sigma||Q_T|^M|Q_{\text{out}}|^{2M}$. Further, $\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$ simulates one copy of $d_{\text{in}}(a)$ and at most M copies of $\overline{A_b}$ in parallel. Note that the sequence $q_1 \cdots q_M$ is uniquely determined by a and $p_1 \cdots p_m$. Hence,

$$\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$$

is a DFA of size

$$|\Sigma||Q_T|^M|Q_{\text{out}}|^{2M} \times |d_{\text{in}}(a)||A_b|^M.$$

We compute the size of B. The size of every NFA in B is

$$\mathcal{O}(|d_{\text{in}}|^2|Q_T|^{M+1}|d_{\text{out}}|^{3M}).$$

Further, B has $\mathcal{O}(|\Sigma||Q_T|^M|d_{\text{out}}|^{2M})$ states. Hence the size of B is

$$\mathcal{O}(|d_{\text{in}}|^3|Q_T|^{3M+1}|d_{\text{out}}|^{5M}).$$

As emptiness of NTA(NFA)s is in PTIME (Proposition 3.18), we get our upper bound. □

From Lemma 6.3, we immediately obtain that the typechecking problem with respect to DTD(DFA)s is tractable for all classes of tree transducers with a bounded deletion path width:

Theorem 6.4. *$TC[\mathcal{T}_{trac}, DTD(DFA)]$ is* PTIME*-complete.*

The lower bound follows from Table 4.1.

Not only do we obtain a PTIME algorithm, Lemma 6.3 also provides a clear view on the concrete complexity in terms of the different parameters. Although the parameters C and K occur in the exponent, we believe these numbers to be small in practical transformations. It is important to point out that the presence of non-copying recursively deleting states do not affect the parameter K. Hence, there is no penalty for the recursive deletion without copying that occurs in many filtering transformations. In contrast to the results in Chapters 4 and 5 that abandoned deletion completely, the present result shows that transformations with small K but arbitrary deletion without copying can still be efficiently typechecked.

Proposition 6.5. *Let T be a tree transducer. The smallest numbers C and K such that $T \in \mathcal{T}_{trac}^{C,K}$ can be computed in* PTIME*.*

Proof. It is obvious that C can be computed in PTIME. We only need to count the maximum number of states that occur as siblings in a rhs in T.

The computation of K is somewhat more complicated. We reduce this problem to the problem of finding a *longest path* (or a path with the *highest cost*) in a directed acyclic graph. The latter problem can be solved in polynomial time (see problem ND29 in [GJ79]). Given a tree transducer $T = (Q_T, \Sigma, q_T^0, R_T)$, we define the *deletion path graph* $G_T = (V_T, E_T)$ — which can still contain cycles — as follows. The set of nodes $V_T = \{(q, a) \mid q \in Q_T, a \in \Sigma\}$. For a node (q, a), the set of outgoing edges is defined as $\{((q, a), (q', a')) \mid a' \in \Sigma, q' \in Q_{q,a}\}$, where $Q_{q,a}$ is the set of states occurring in top(rhs(q, a)). Note that these edges can be computed in PTIME. To every edge $e = ((q, a), (q', a'))$ we associate a *cost*, denoted cost(e), which is the number of states occurring at top(rhs(q, a)). The cost of a *path* p in G_T is the product of the costs of the edges occurring in p. Note that by definition, all costs of edges are at least one and that the deletion path width of T is equal to the largest cost of a path in G_T.

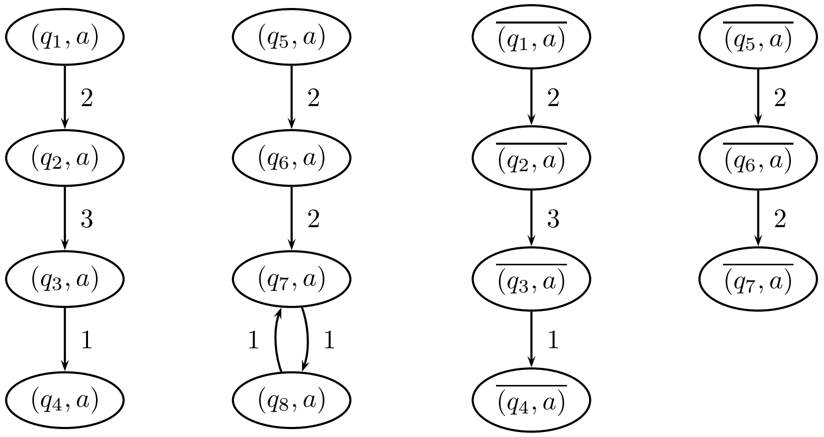
We now transform G_T into an acyclic graph as follows. Assume that there is at least one edge with cost two, otherwise, we immediately know that $K = 1$. First we investigate, for every edge $e = ((q, a), (q', a'))$, if it is part of a cycle. This can be done in NLOGSPACE, and, hence, also in PTIME. If there exists an e which is part of a cycle and cost$(e) > 1$, then we can immediately halt the algorithm and conclude that K cannot be bounded. Therefore, assume now that every edge occurring in a cycle has cost one. Since cycles with cost one have no effect on the cost of the longest path in G_T, we remove these cycles from G_T by joining the nodes that they connect.

Formally, we define an equivalence relation $\equiv$ between nodes of G_T. For two nodes v and v', we say that $v \equiv v'$ if *(1)* $v = v'$; or *(2)* v and v' occur in the same cycle (that is, there exists a directed path from v to v' and from v' to v). For a node v, we denote by $\overline{v}$ the set of nodes which are equivalent to v. We now define the graph $G'_T = (V'_T, E'_T)$, where $V'_T = \{\overline{v} \mid v \in V_T\}$ and $E'_T = \{(\overline{(q, a)}, \overline{(q', a')}) \mid ((q, a), (q', a')) \in E_T$ and $\overline{(q, a)} \neq \overline{(q', a')}\}$.

Since G'_T is a DAG, we can compute the longest path in G'_T in PTIME. Note that, in the longest path problem, we want to maximize the sum of the costs of the edges, whereas we want to maximize their product. However, this can directly be incorporated in the longest path algorithm, as our costs are always positive integers. It is easy to see that the maximum possible intermediate cost is always bounded by $|T|^{|T|}$. This number can be represented using $|T| \cdot \lceil \log |T| \rceil$ bits, which is polynomial. □

We illustrate how the algorithm in Proposition 6.5 computes C and K for the tree transducer of Example 6.1.

Example 6.6. Let T be the tree transducer defined in Example 6.1. It is immediate that $C = 3$. The deletion path graph $G_T = (V_T, E_T)$ is graphically represented in Figure 6.1(a). The graph G'_T, which is obtained from G_T by eliminating the cycles, is shown in Figure 6.1(b). The path $\overline{(q_1, a)}\ \overline{(q_2, a)}\ \overline{(q_3, a)}\ \overline{(q_4, a)}$ in G'_T has a cost of 6, which is the highest possible cost in G'_T.[2] Therefore, $K = 6$. ◇



(a) The deletion path graph G_T.

(b) The graph G'_T.

Figure 6.1: The deletion path width graphs G_T and G'_T of the transducer T from Example 6.1

6.1.2 Lower Bounds for Extensions

We show that the scenario of the previous section cannot be enlarged in an obvious way without rendering the typechecking problem intractable. The PTIME result of the previous section is obtained for those classes of transducers that can bound their deletion path width and their copying width by a constant. The restriction on copying width cannot be relaxed: even TC[$\mathcal{T}_{\text{nd,uc}}$,DTD(DFA)] is PSPACE-hard with fixed

[2] Recall that the cost of a path is defined to be the *product* of the costs of its edges.

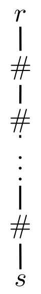


Figure 6.2: Structure of the trees defined by the input schema in the proof of Theorem 6.7.

input or output schema (cf. Tables 4.1 and 5.1). What about the constraint on the bounded deletion path width? A slight relaxation of this constraint is to require that the deletion path width is finite for each transducer in the class but not necessarily bounded by a predetermined constant. We denote by $\mathcal{T}_{\text{dw=2,cw=2,fdpw}}$ the class of such transducers with the additional constraint that the deletion width and copying width of states is restricted to two. The next theorem shows that typechecking in this scenario is intractable.

Theorem 6.7. *$TC[\mathcal{T}_{dw=2,cw=2,fdpw}, DTD(DFA)]$ is* PSPACE*-hard.*

Proof. We reduce the intersection emptiness problem of an arbitrary number of DFAs, which is PSPACE-hard (Proposition 3.9), to the typechecking problem. The intersection emptiness problem for DFAs asks whether $\bigcap_{i=1}^{n} L(A_i) = \emptyset$ for a given sequence of DFAs $A_1, \ldots, A_n$.

For $i = 1, \ldots, n$, let $A_i = (Q_i, \Delta, \delta_i, I_i, F_i)$ be a DFA. Define $\Sigma = \Delta \cup \{\#, r, \text{ok}\}$. We construct two DTD(DFA)s d_{in} and d_{out}, and a tree transducer T with deletion and copying width two, and deletion depth $\lceil \log n \rceil$, such that T typechecks with respect to d_{in} and d_{out} if and only if $\bigcap_{i=1}^{n} L(A_i) = \emptyset$.

The input DTD d_{in} with start symbol r is defined as follows: $d_{\text{in}}(r) = \#$ and $d_{\text{in}}(\#) = \# + \Delta^*$. Then, every allowed tree is of the form as depicted in Figure 6.2, where $s \in \Delta^*$. We define the tree transducer $T = (Q_T, \Sigma, q_T^0, R_T)$ where $Q_T = \{q_T^0, q^1, \ldots, q^{\lceil \log n \rceil}\}$ and R_T consists of the following rules:

- $(q_T^0, r) \rightarrow r(q^1 \# q^1)$;
- $(q^{i-1}, \#) \rightarrow q^i \# q^i$ for $i = 2, \ldots, \lceil \log n \rceil$;
- $(q^i, a) \rightarrow \text{ok}$ for $i < \lceil \log n \rceil$ and $a \in \Delta$;
- $(q^{\lceil \log n \rceil}, \#) \rightarrow \text{ok}$; and
- $(q^{\lceil \log n \rceil}, a) \rightarrow a$ for all $a \in \Sigma$.

Note that T produces a tree of the form $r(w)$ with $w \in (\Delta \cup \{\#, \text{ok}\})^*$. When the depth of the input tree is different from $\lceil \log n \rceil$, w contains the symbol *ok*. Otherwise, w consists of at least n copies of the Δ-string s.

It remains to define the DFA specifying $d_{\text{out}}(r)$. The automaton starts by simulating A_1. Further, when the DFA encounters the ith occurrence of a #, the simulation of A_{i+1} is started. The DFA accepts when at least one A_i rejects, or when the symbol *ok* appears in the output.

So, for all $t \in L(d_{\text{in}})$ with depth $\lceil \log n \rceil$, we have that $T(t) \in L(d_{\text{out}})$ if and only if $\bigcap_{i=1}^{n} L(A_i) = \emptyset$. As for all other trees $t \in L(d_{\text{in}})$ we have that $T(t) \in L(d_{\text{out}})$, this instance typechecks if and only if $\bigcap_{i=1}^{n} L(A_i) = \emptyset$. □

For completeness, we also mention here that typechecking is EXPTIME-hard for deleting tree transducers with a deletion and copying width of two. This hardness even holds for a fixed input and output schema (Theorem 5.1).

6.1.3 Tree Automata

In this section, we turn to schemas defined by unranked tree automata. We show that when every right hand side of a rewrite rule contains at most 1 state, recursively deleting of width one remains tractable in the presence of DTA^c(DFA)s. The latter is the class of bottom-up deterministic complete tree automata that use DFAs to represent transition functions. Such transformations are mild generalizations of relabelings and we therefore denote the class of these transducers by $\mathcal{T}_{\text{del-relab}}$. It is hence not surprising that the output type of a transducer in $\mathcal{T}_{\text{del-relab}}$ can be exactly captured by a tree automaton. The latter observation is a generalization of the corresponding result for ranked tree transducers (Proposition 7.8(b) in [GS97]). We only have to show that the construction of the unranked tree automaton can be done in PTIME. Typechecking then reduces to containment checking of NTA(NFA)s in DTA^c(DFA)s.

We make use of the following Lemma.

Lemma 6.8. *Let A be an NTA(NFA) and T be a non-deleting tree transducer for which every rhs contains at most one state. Then we can construct in polynomial time an NTA(NFA) B such that $L(B) = T(L(A))$.*

Proof. Let $A = (Q_A, \Sigma, \delta_A, F_A)$ be an NTA(NFA) and let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a tree transducer such that every rule in R_T is of the form $(q, a) \to b(h)$, where h contains at most one state. We construct a NTA(NFA) $B = (Q_B, \Sigma, \delta_B, F_B)$ such that $L(B) = T(L(A))$. Intuitively, when B processes a tree t, it guesses the tree t' such that $T(t') = t$ and verifies whether $t' \in L(A)$.

The automaton B is defined as follows:

$$Q_B = \Sigma \times Q_A \times Q_T \times \cup_{(q,a)\in Q_T \times \Sigma} \text{Nodes}(\text{rhs}(q, a));$$

$F_B = \Sigma \times F_A \times \{q_T^0\} \times \{\varepsilon\}$. Intuitively, when $t \in L(B)$, it means that there is some tree $t' \in L(A)$ such that $T(t') = t$. If a node $v \in \text{Nodes}(t)$ is labeled with (a, q_A, q_T, u) in an accepting run of B, it intuitively means that there is a node v' in t' for which

- the label of v' is a;
- $\lambda(v') = q_A$ in some accepting run λ of A on t';
- v' was visited by T in state q_T; and

- v was constructed by T from node u in $\text{rhs}(q_T, a)$.

Formally, for any $a \in \Sigma$, $q_A \in Q_A$ and $q_T \in Q_T$, let $t_1 = \text{rhs}(q, a)$ and let $i_1 \cdots i_k \in \mathbb{N}^*$ be the unique node in t_1 labeled by a state, if it exists. Then, for every node $u \in \text{Nodes}(t_1)$ different from $i_1 \cdots i_{k-1}$ or $i_1 \cdots i_k$, with children $u1, \ldots, un$, we define

$$\delta_B((a, q_A, q_T, u), \text{lab}^t(u)) := \{(a, q_A, q_T, u1) \cdots (a, q_A, q_T, un)\}.$$

It is trivial to construct an NFA of size n that accepts this singleton. Note that this language contains only the empty string if u is a leaf.

Denote by v the node $i_1 \cdots i_{k-1}$ and suppose that v has m children. Then, to define $\delta_B((a, q_A, q_T, v), \text{lab}^{t_1}(v))$, let $D = (Q_D, Q_A, \delta_D, I_D, F_D)$ be the NFA representing $\delta(q_A, a)$ and let q'_T be the state in $\text{rhs}(q_T, a)$. Then,

$$\delta_B((a, q_A, q_T, v), \text{lab}^{t_1}(v))$$

is the NFA accepting the language

$$(a, q_A, q_T, v1) \cdots (a, q_A, q_T, v(i_k - 1))L(D')(a, q_A, q_T, v(i_k + 1)) \cdots (a, q_A, q_T, vm)$$

where D' is obtained from D by replacing every transition $\delta_D(p_1, q'_A) = \{p_2\}$ by

- the transitions $\delta_D(p_1, (c, q'_A, q'_T, \varepsilon)) = \{p_2\}$ for every $c \in \Sigma$ when $\text{rhs}(q'_T, c)$ is a tree; and by
- the transitions

  $$\delta_D\big(p_1, (c, q'_A, q'_T, 1)\big) = \{p_1^{q'_T,c,1}\},$$
  $$\delta_D\big(p_1^{q'_T,c,1}, (c, q'_A, q'_T, 2)\big) = \{p_1^{q'_T,c,2}\}, \ldots$$
  $$\ldots, \delta_D\big(p_1^{q'_T,c,\ell-1}, (c, q'_A, q'_T, \ell)\big) = \{p_2\}$$

  when $\text{rhs}(q'_T, c)$ is a hedge consisting of $\ell > 1$ trees. In the above definitions, the states $p_1^{q'_T,c,1}, \ldots, p_1^{q'_T,c,\ell-1}$ are new states not occurring in the state set of D.

In other words, B guesses a string of children of node v' in t', continues with the simulation of T by remembering q'_T and continues with the simulation of A on t' by running D over the states of A.

So, B has $\mathcal{O}(|\Sigma||A||T|)$ states, and for each such state, the size of B's transition function is $\mathcal{O}(|\Sigma||A||T|)$. □

We are now ready to prove the following theorem.

Theorem 6.9. *$TC[\mathcal{T}_{\text{del-relab}}, DTA^c(DFA)]$ is* PTIME*-complete.*

Proof. The lower bound is immediate from Lemma 3.19.

For the upper bound, we reduce the typechecking problem to the emptiness problem of the intersection of two NTA(NFA)s, which is in PTIME (Proposition 3.18). To this end, let A_{in} and A_{out} be the input and output tree automaton, respectively.

We construct a non-deleting tree transducer T' from T by replacing every deleting state q in a rhs of T by $\#(q)$. So, T' outputs a $\#$ whenever T would process a deleting state. We assume that $\# \not\in \Sigma$. We now construct an NTA(NFA) B_{in} such that $L(B_{\text{in}}) = T'(L(A_{\text{in}}))$. According to Lemma 6.8, B_{in} can be computed in time polynomial in the size of A_{in} and T'.

As A_{out} is a complete DTA(DFA), the complement $\overline{A_{\text{out}}}$ can easily be computed by switching the final and non-final states. Note that the size of $\overline{A_{\text{out}}}$ is linear in the size of A_{out}.

Define the $\#$-eliminating function γ as follows: $\gamma(a(h))$ is $\gamma(h)$ when $a = \#$ and $a(\gamma(h))$ otherwise; further, $\gamma(t_1 \cdots t_n) := \gamma(t_1) \cdots \gamma(t_n)$. We construct the NTA(NFA) B_{out}, such that B_{out} accepts a tree $t \in \mathcal{T}_{\Sigma \cup \{\#\}}$ if and only if $\gamma(t)$ is accepted by $\overline{A_{\text{out}}}$.

According to the proof of Theorem 4.1(1) in Chapter 4, we can construct B_{out} in LOGSPACE. The instance then typechecks if and only if $L(B_{\text{in}} \cap B_{\text{out}}) = \emptyset$. □

For completeness, we note that typechecking with respect to DTA(DFA)s already turns EXPTIME-hard for tree transducers with a copying width of one, and for which the right-hand sides of rewrite rules are allowed to contain at most two states (see Theorem 4.5(2)). In the reduction, both the input and output schemas are fixed.

6.2 XPath Patterns

An approach complementary to deletion, is the use of XPath patterns to skip nodes of the input tree [CD99]. As XPath patterns are very likely to occur in practical transformations, it is important to study the complexity of the typechecking problem for tree transducers that allow the use of XPath patterns. We only consider XPath patterns for downward navigation and therefore restrict attention to the following axes and operations: child (/), descendant (//), wildcard ($*$), disjunction (|), and filter ([ ]). We allow element tests and either the child or descendant axis in every fragment of XPath we consider.

Definition 6.10. An *XPath*$\{/, //, [\,], |, *\}$ *pattern* P is an expression $\cdot/\phi$ or $\cdot//\phi$ where ϕ is defined by the following grammar:

$$\begin{array}{rlr} \phi := & \phi_1|\phi_2 & \text{(disjunction)} \\ & |\ \phi_1/\phi_2 & \text{(child)} \\ & |\ \phi_1//\phi_2 & \text{(descendant)} \\ & |\ \phi_1[P] & \text{(filter)} \\ & |\ a & \text{(element test)} \\ & |\ * & \text{(wildcard)} \end{array}$$

◇

An example of an XPath$\{/, //, [\,], |, *\}$ pattern is $\cdot/(a|b)//c[\cdot//e]/*$.

Note that in our framework, we only use XPath patterns that start with "$\cdot$", that is, they always start from the context node. We use the following notational convention: for a sequence X of axes and operations, we denote by XPath$\{X\}$ the XPath patterns that only use the axes and operations in $\{X\}$. For instance, XPath$\{/, |\}$ denotes the fragment of XPath$\{/, //, [\,], |, *\}$ where only element test and the child and disjunction axes are used. An XPath pattern P defines a function $f_P : t \times \text{Nodes}(t) \to 2^{\text{Nodes}(t)}$. We inductively define f_P as follows.

- $f_{./\phi}(t,u) := \{v \mid \exists z \in \mathbb{N} : v \in f_\phi(t,uz)\}$;
- $f_{.//\phi}(t,u) := \{v \mid \exists z \in \mathbb{N}^* - \{\varepsilon\} : v \in f_\phi(t,uz)\}$;
- $f_{\phi_1|\phi_2}(t,u) := f_{\phi_1}(t,u) \cup f_{\phi_2}(t,u)$;
- $f_{\phi_1/\phi_2}(t,u) := \{v \mid \exists w \in \text{Nodes}(t), z \in \mathbb{N} : w \in f_{\phi_1}(t,u) \text{ and } v \in f_{\phi_2}(t,wz)\}$;
- $f_{\phi_1//\phi_2}(t,u) := \{v \mid \exists w \in \text{Nodes}(t), z \in \mathbb{N}^* - \{\varepsilon\} : w \in f_{\phi_1}(t,u) \text{ and } v \in f_{\phi_2}(t,wz)\}$;
- $f_{\phi_1[P]}(t,u) := \{v \mid v \in f_{\phi_1}(t,u) \text{ and } f_P(t,v) \neq \emptyset\}$;
- $f_a(t,u) := \begin{cases} \{u\} & \text{if } \text{lab}^t(u) = a; \\ \emptyset & \text{otherwise;} \end{cases}$
- $f_*(t,u) := \{u\}$;

When a node u is in $f_P(t,\varepsilon)$, we say that P *selects* u in t.

Let $\mathcal{P} \subseteq \text{XPath}\{/,//,[\,],|,*\}$ be a set of XPath patterns. We explain how the syntax and the semantics of transducers is extended to patterns in $\mathcal{P}$. We denote the latter fragment by $\mathcal{T}^{\mathcal{P}}$. Rules are now of the form $(q,a) \to h$ where $h \in \mathcal{H}_\Sigma((Q \times \mathcal{P}) \cup Q)$. That is, state-pattern pairs $\langle q, P \rangle$ can now also occur at leaves. Previously, all children of the current node were processed; now, only the nodes selected by P starting from the current node. These nodes are processed in document order, that is, the order in which they would occur in a depth-first left to right traversal of the tree. We denote state-pattern pairs with angled parentheses to avoid confusion in the string representation of trees.

If T is a tree transducer, $t = a(t_1 \cdots t_n)$ and there is a rule $(q,a) \to h \in R_T$ then $T^q(t)$ is obtained from h by replacing every node u in h labeled with $\langle p, P \rangle$ by the hedge $T^p(t/u_1) \cdots T^p(t/u_m)$ where $f_P(t,\varepsilon) = \{u_1, \ldots, u_m\}$ and the sequence $u_1, \ldots, u_m$ occurs in document order. Recall that we denote by t/u the subtree of t rooted at u. Note that the context node is always set to the root of the subtree that is to be processed by T and that every XPath pattern is of the form $\cdot/\phi$ or $\cdot//\phi$. In this way, the context node itself is never selected and the transformation by T always terminates.

Example 6.11. When making use of XPath patterns, we can write the first document transformation in Example 2.15 more succinctly as follows:

$$\begin{array}{lll} (q,\text{book}) & \to & \text{book}(q) \\ (q,\text{chapter}) & \to & \text{chapter } \langle q, \cdot//\text{title} \rangle \\ (q,\text{title}) & \to & \text{title} \end{array} \qquad \diamond$$

Via a reduction to Theorem 6.4, we show that for very simple XPath patterns added to the formalism typechecking remains in PTIME.

Theorem 6.12. *$TC[\mathcal{T}_{trac}^{XPath\{/,*\}}, DTD(DFA)]$ is* PTIME*-complete.*

Proof. The lower bound is immediate from Theorem 6.4. We prove the upper bound. In particular, we will show that for any tree transducer $T \in \mathcal{T}_{\text{trac}}^{\text{XPath}\{/,*\}}$, we can construct an equivalent tree transducer $T' \in \mathcal{T}_{\text{trac}}$ which has size polynomial in the size of T and the same copying and deletion path width as T. Intuitively, we convert every XPath-pattern P occurring in T to a DFA, which we simulate by using deleting states in T'. The simulation of such DFAs only introduces non-recursively deleting states of deleting width one, hence, unaffecting the deletion path width.

Formally, let $T = (Q_T, \Sigma, q_T^0, R_T)$ and let $\mathcal{P}_T$ be the set of XPath patterns occurring in T. For every XPath-pattern $P \in \mathcal{P}_T$, we can easily construct a DFA $A_P = (Q_P, \Sigma, \delta_P, \{q_P^I\}, \{q_P^F\})$ accepting all strings $a_1 \cdots a_n$ such that P selects the a_n-labeled node in the tree $r(a_1(\cdots(a_n)))$ when evaluated from the root. Moreover, each A_P has a linear number of states in the number of symbols of P and at most a quadratic number of transitions. Further, A_P is acyclic, only accepts a finite language, and all strings in $L(A_P)$ are of the same length. Without loss of generality, we assume that the sets Q_P are pairwise disjoint and disjoint from Q_T.

We construct $T' = (Q_{T'}, \Sigma, q_T^0, R_{T'})$ as follows. Its state set is $Q_T \cup \bigcup_{P \in \mathcal{P}_T}(Q_T \times Q_P)$. For every rule $(q,a) \to h$ in R_T, and for every $\langle p, P\rangle$ occurring in h we have the following set of rules in $R_{T'}$:

- $(q,a) \to h'$ where h' is the hedge obtained from h by replacing every occurrence of $\langle p, P\rangle$ by (p, q_P^I);
- $((p,q_P),b) \to (p, \delta_P, q_P, b))$ for every $q_P \in Q_P$ and $b \in \Sigma$ such that $\delta_P(q_P, b) \neq q_P^F$; and
- $((p,q_P),b) \to \text{rhs}(p,b)$ for every $q_P \in Q_P$ and $b \in \Sigma$ such that $\delta_P(q_P,b) = q_P^F$.

Note that the final state of A_P itself does not occur in the rewrite rules.

We only need to argue that the XPath patterns in T are evaluated correctly in T'. To this end, it easy to see that we only use deleting states for nodes that are skipped in the input tree by the XPath patterns, and that we continue in the correct state in Q_T in the nodes that are selected by the XPath patterns. Further, only deleting states of width one are introduced. So, $T' \in \mathcal{T}_{\text{trac}}^{C,K}$ whenever $T \in \mathcal{T}_{\text{trac}}^{C,K}$. □

Although the fragment XPath$\{/,*\}$ is very limited, we show in Theorem 6.16 that there is not much room for improvement. The lower bounds in bullet (1) follow from a reduction from XPath containment in the presence of DTDs with DFAs [NS03, Woo03]. This problem is defined as follows: given a DTD(DFA) d and XPath patterns P_1 and P_2, is it true that $f_{P_1}(t,\varepsilon) \subseteq f_{P_2}(t,\varepsilon)$ for all trees t satisfying d.

In the statements of Theorem 6.13 and Lemma 6.15, let XPath$\{X\}$ denote any fragment XPath$\{/,|\}$, XPath$\{//,|\}$, XPath$\{/,[\,]\}$ or XPath$\{//,[\,]\}$.

Theorem 6.13 ([NS03, Woo01, Woo03]). *XPath$\{X\}$ containment in the presence of DTD(DFA)s is co*NP*-hard.*

We note that Wood used DTDs with DFAs in his coNP-hardness proof of the inclusion problems of XPath$\{/,[\,]\}$ and XPath$\{//,[\,]\}$ [Woo].

We also make use of the following lemma. The proof uses the notion of *selecting literals* of an XPath pattern. Intuitively, an element test or a wildcard in an XPath pattern is a *selecting literal* if it used for selecting nodes in the document rather than for navigation in the document. In the following definition, we denote by ℓ an arbitrary $a \in \Sigma$ or a wildcard.

- ℓ is a selecting literal in $\cdot/\phi_2$, in $\cdot//\phi_2$, in ϕ_1/ϕ_2, in $\phi_1//\phi_2$ or in $\phi_2[P]$ if it is a selecting literal in ϕ_2.
- ℓ is a selecting literal in $\phi_1|\phi_2$ if it is a selecting literal in ϕ_1 or in ϕ_2.
- ℓ is a selecting literal in ℓ.

Example 6.14. We provide some examples.

- The selecting literals of $\cdot//a/b/((c/d)|(b/e))$ are labeled d and e.
- The selecting literal of $\cdot/a[\cdot/c]//*[\cdot/(b|c)]$ is labeled $*$. ◇

Lemma 6.15 ([MS04, NS03]). *Given a DTD(DFA) d and XPath$\{X\}$ patterns P_1 and P_2, we can construct a DTD(DFA) d' and XPath$\{X\}$ patterns P_1' and P_2' in* LOGSPACE *such that deciding whether*

$$f_{P_1}(t,\varepsilon) \subseteq f_{P_2}(t,\varepsilon) \text{ for all trees } t \text{ satisfying } d,$$

is equivalent to deciding whether for all trees t satisfying d',

$$\text{if } P_1' \text{ selects an } x_1\text{-labeled node in } t, \text{ then } P_2' \text{ selects an } x_2\text{-labeled node in } t.$$

Proof sketch. The DTD d' is identical to d, except that d' also requires that every node has a child leaf labeled with x_1 and one with x_2.

For $i = 1, 2$, pattern P_i' is constructed from P_i by replacing for every selecting literal ℓ

(a) subpatterns $/\ell[\phi_1]\cdots[\phi_n]$ by $/\ell[\phi_1]\cdots[\phi_n]/x_i$; and

(b) subpatterns $//\ell[\phi_1]\cdots[\phi_n]$ by $//\ell[\phi_1]\cdots[\phi_n]//x_i$,

where $[\phi_1]\cdots[\phi_n]$ is a (possibly empty) sequence of filter operations. □

The lower bound in bullet (2) of Theorem 6.16 follows from a reduction from the INTERSECTION EMPTINESS problem for DFAs over a unary alphabet. Given an arbitrary number of DFAs $A_1, \ldots, A_n$ over alphabet $\{a\}$, this problem asks whether $\bigcap_{i=1}^n L(A_i) = \emptyset$.

Theorem 6.16. *The following problems are* coNP*-hard.*

(1) $TC[\mathcal{T}_{nd,bc}^{XPath\{X\}}, DTD(DFA)]$, for XPath$\{X\}$ among

- *XPath$\{/, |\}$;*
- *XPath$\{//, |\}$;*
- *XPath$\{/, [\,]\}$ and*
- *XPath$\{//, [\,]\}$.*

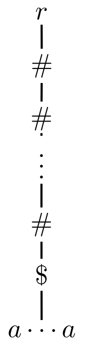


Figure 6.3: Structure of the trees defined by the input schema in the proof of Theorem 6.16(2).

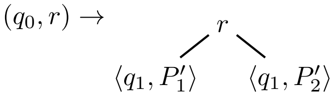
(2) $TC[\mathcal{T}_{trac}^{XPath\{//\}}, DTD(DFA)]$.

Proof. (1) In all four cases, we can do a reduction from the XPath$\{X\}$ containment problem in the presence of DTD(DFA)s, which is coNP-hard according to Theorem 6.13.

To this end, let P_1 and P_2 be two XPath$\{X\}$ patterns and let (Σ, d, s) be a DTD(DFA). We construct an instance of the typechecking problem that typechecks if and only if $P_1(t, \varepsilon) \subseteq P_2(t, \varepsilon)$ for every $t \in (\Sigma, d, s)$.

The input DTD $(\Sigma \uplus \{r, x_1, x_2\}, d_{\text{in}}, r)$ is identical to $(\Sigma \uplus \{x_1, x_2\}, d', s)$ as constructed in the proof of Lemma 6.15, except that r is a new alphabet symbol and $d_{\text{in}}(r) = s$. Let P'_1 and P'_2 be the XPath$\{X\}$ patterns as constructed in the proof of Lemma 6.15.

We define the tree transducer $T = (\{q_T^0, q_1\}, \Sigma, q_T^0, R_T)$. The set R_T contains the following rule:



For state q_1 we have the rules $(q_1, x_1) \to x_1$ and $(q_1, x_2) \to x_2$, which is the identity transformation on x_1 and x_2. The output DTD d_{out} has start symbol r and $d_{\text{out}}(r) = x_2^* + (x_1 x_1^* x_2 x_2^*)$. The latter checks that x_1 does not appear or appears together with x_2. The correctness now follows from the statement of Lemma 6.15.

(2) We reduce the INTERSECTION EMPTINESS problem for DFAs $A_1, \ldots, A_n$ over over alphabet $\{a\}$ to the present problem. The INTERSECTION EMPTINESS problem is coNP-hard (Lemma 3.15).

The input DTD d_{in} has start symbol r and is defined as follows: $d_{\text{in}}(r) = \#$, $d_{\text{in}}(\#) = \# + \$$, and $d_{\text{in}}(\$) = a^*$. So, trees are of the form as depicted in Figure 6.3.

We define the tree transducer $T = (\{q_T^0, q_1, q_2, q_3\}, \{a, r, \#, \$\}, q_T^0, R_T)$ with the following rewrite rules:

$$\begin{array}{ll} (q_T^0, r) \to r(\langle q_1, \cdot//\#\rangle) & (q_1, \#) \to \langle q_2, \cdot//\$\rangle \\ (q_2, \$) \to \langle q_3, \cdot//a\rangle\$ & (q_3, a) \to a \end{array}$$

The transducer starts by selecting every #-labeled node. For each of those (say there are k) it selects the single \$-labeled descendant node. So, k copies of the input string in $L(a^*)$ are made, separated by the \$-symbol.

The output DTD simulates the ith DFAs on the ith copy and accepts if one of them rejects or if there are less than n copies. So, the instance typechecks if the intersection is empty. Note that the copying width (C) and the deletion path width (K) are both one. □

The previous results show that, to retain tractability of typechecking, only very restricted XPath patterns can be added to $\mathcal{T}_{\text{trac}}$, or even $\mathcal{T}_{\text{nd,bc}}$. Next, we look at transducers where patterns are specified by DFAs (rather than by XPath patterns). We denote this fragment by $\mathcal{T}^{DFA}$. The semantics of such selecting DFAs is as follows: given a DFA A and a context node u, a descendant v of u is selected by A if and only if A accepts the string of labels on the path from u to v. From Theorem 6.16(2) it follows that typechecking is already hard when we allow patterns to be specified by DFAs in $\mathcal{T}_{\text{trac}}$ transducers (for instance, every XPath{//}-pattern used in the proof of Theorem 6.16(2) can be translated to an equivalent DFA in linear time). When we completely disallow deletion however, we still have tractability.

Theorem 6.17. *$TC[\mathcal{T}^{DFA}_{nd,bc}, DTD(DFA)]$ is in* PTIME.

Proof. We show that for any tree transducer $T \in \mathcal{T}^{DFA}_{\text{nd,bc}}$, we can construct an equivalent tree transducer $T' \in \mathcal{T}_{\text{trac}}$ with size linear in the size of T, and the same copying and deletion path width as T.

The proof is quite analogous to the proof of Theorem 6.12. We simulate every DFA-pattern in T by deleting states in T'. The simulation of such DFAs only introduces deleting states of deletion width one.

Formally, let T be the transducer $(Q_T, \Sigma, q_T^0, R_T)$. Let $A_x = (Q_x, \Sigma, \delta_x, \{q_x^I\}, \{q_x^F\})$, $x \in X$ be the sets of selecting DFAs in T, where X is a set of indices. Without loss of generality, we assume that the sets Q_x are pairwise disjoint and disjoint from Q_T.

We construct $T' = (Q_{T'}, \Sigma, q_T^0, R_{T'})$ as follows. Its state set is $Q_T \cup \bigcup_{x \in X}(Q_T \times Q_x)$. For every rule $(q, a) \to h$ in R_T, and for every $\langle p, A_x \rangle$ occurring in h we have the following set of rules in $R_{T'}$:

- $(q, a) \to h'$ where h' is the hedge h where every $\langle p, A_x \rangle$ is replaced by (p, q_x^I);
- $((p, q_x), b) \to (p, \delta_x(q_x, b))$ for every $p_x \in Q_x$ and $b \in \Sigma$ such that $\delta_x(p_x, b) \neq q_x^F$; and
- $((p, q_x), b) \to \text{rhs}(p, b)\ (p, q_x^F)$ for every $p_x \in Q_x$ and $b \in \Sigma$ such that $\delta_x(p_x, b) = q_x^F$. Since T is non-deleting, no states occur in $\text{top}(\text{rhs}(p, b))$ and hence, (p, q_x) has deletion path width one.

The main difference with Theorem 6.12 is that when we arrive in a final state of A_x, the simulation of A_x still needs to go on. This is shown in the third bullet. There, the output hedge consists of the output generated by the selection of the current node, followed by the output generated by selecting descendant nodes of the current node by A_x. Hence, the document order is respected. Again, $T' \in \mathcal{T}^{C,K}_{\text{trac}}$ whenever $T \in \mathcal{T}^{C,K}_{\text{trac}}$. □

As shown by Green et al., any XPath pattern in XPath$\{/, //, *\}$ for which the number of wildcards occurring between two descendant axes is bounded from above by c, can be translated to an equivalent DFA of size $\mathcal{O}(n^c)$, where n is the size of the pattern [GGM$^+$04]. Hence, typechecking is in PTIME for $\mathcal{T}_{\text{nd,bc}}^{\text{XPath}\{/,//,*\}}$ where patterns are such that c is bounded by a constant.

It remains open whether typechecking for $\mathcal{T}_{\text{nd,bc}}^{\text{XPath}\{/,//,*\}}$ is in PTIME in general.

6.3 Deletion, Unbounded Copying, and RE$^+$

All tractable fragments of the previous setting assume a uniform bound on the copying and deletion width of a transducer. Although in practice these bounds will usually be small and Lemma 6.3 provides a detailed account of their effect, the restrictions remain somewhat artificial. In the present section, we therefore investigate fragments in which there are no restrictions on the copying or deletion power of the transducer. As the typechecking problem is already PSPACE-hard when we use DTD(DFA)s, we have to restrain the schemas, for example, by restricting the regular expressions in rules.

We consider the following regular expressions. Let RE$^+$ be the set of regular expressions of the form $\alpha_1 \cdots \alpha_k$ where every α_i is ε, a, or a^+ for some $a \in \Sigma$. An example is `title` `author`$^+$ `chapter`$^+$. In this section, we show that typechecking for arbitrary tree transducers with respect to DTD(RE$^+$)s is in PTIME. We note that every DTD(RE$^+$) is either non-recursive (that is, an a-labeled node has no a-labeled descendants) or defines the empty language. However, the tractability of typechecking remains non-trivial, as in general typechecking is already PSPACE-complete when using DTD(DFA)s only defining trees of depth two (Theorem 4.3(4)).

Notice that deciding inclusion and equivalence for RE$^+$-expressions is in PTIME, as every such expression can be transformed to a corresponding DFA in linear time. Moreover, deciding whether the intersection of an arbitrary number of RE$^+$-expressions is empty can also be decided in PTIME (see Chapter 9). We further note that Benedikt, Fan, and Geerts, among other things, obtained that satisfiability of various fragments of XPath is tractable in the presence of a DTD(RE$^+$) [BFG05].

We present the typechecking algorithm and show its correctness. For the rest of this section, let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a tree transducer and denote the input and output DTD by d_{in} and d_{out}, respectively. We introduce some notational shorthands. For an RE$^+$-expression e and DTD d, we denote by d^e the hedge language

$$\{a_1(h_1) \cdots a_n(h_n) \mid a_1 \cdots a_n \in L(e) \text{ and } \forall i = 1, \ldots, n : a_i(h_i) \in L((d, a_i))\}.$$

So, if $t_1 \cdots t_n \in d^e$ then $\text{top}(t_1) \cdots \text{top}(t_n) \in L(e)$ and every t_i is a derivation tree of $(d, \text{top}(t_i))$. Recall that (d, a_i) denotes the DTD d with start symbol a_i (page 13). For a state $q \in Q_T$ and an alphabet symbol $a \in \Sigma$, we say that the pair (q, a) is *reachable* if there exists a tree t in $L(d_{\text{in}})$ such that T processes at least one node of t labeled with a in state q. The set of reachable pairs can be computed in PTIME.

To verify that the instance typechecks, we have to check that for every reachable pair (q, a) and for every node u in $\text{rhs}(q, a)$ that

$$\{z_0 \text{top}(T^{q_1}(h)) z_1 \cdots z_{k-1} \text{top}(T^{q_k}(h)) z_k \mid h \in d_{\text{in}}^e\} \subseteq d_{\text{out}}(\sigma),$$

where $e = d_{\text{in}}(a)$, $z_0q_1z_1 \cdots z_{k-1}q_kz_k$ is the concatenation of u's children, and σ is the label of u. In the above, for $h = t_1 \cdots t_n$, we denoted by $T^q(h)$ the hedge $T^q(t_1) \cdots T^q(t_n)$. We denote the above language

$$\{z_0\text{top}(T^{q_1}(h))z_1 \cdots z_{k-1}\text{top}(T^{q_k}(h))z_k \mid h \in d_{\text{in}}^e\}$$

by $L_{q,a,u}$. Note that the latter language is not necessarily regular, or even context-free.

We construct an extended context-free grammar $G_{q,a,u}$ such that $L(G_{q,a,u}) \subseteq d_{\text{out}}(\sigma)$ if and only if $L_{q,a,u} \subseteq d_{\text{out}}(\sigma)$. More specifically, define $G_{q,a,u} = (V, \Sigma, P, S)$, where $V = \{\langle p, b\rangle \mid p \in Q_T, b \in \Sigma\}$ is the set of non-terminals, Σ is the set of terminals, P is the set of production rules and $S = \langle q, a\rangle$ is the start symbol. Each non-terminal $\langle p, b\rangle$ corresponds to the string language $\{\text{top}(T^p(t)) \mid t \in L((d_{\text{in}}, b))\}$. It remains to define the production rules P. For the start symbol $\langle q, a\rangle$, we have the rule

$$\langle q, a\rangle \to z_0\langle q_1, e_1\rangle^{\theta_1} \cdots \langle q_1, e_n\rangle^{\theta_n} z_1 \cdots z_{k-1}\langle q_k, e_1\rangle^{\theta_1} \cdots \langle q_k, e_n\rangle^{\theta_n} z_k,$$

where $e = e_1^{\theta_1} \cdots e_n^{\theta_n}$, every $e_i \in \Sigma$ and θ_i is either $+$ or the empty string. For a non-terminal $\langle p, b\rangle$ let $d_{\text{in}}(b) = b_1^{\alpha_1} \cdots b_m^{\alpha_m}$ and let $\text{top}(\text{rhs}(p, b)) = s_0p_1s_1 \cdots p_\ell s_\ell$. Then we add the rule

$$\langle p, b\rangle \to s_0\langle p_1, b_1\rangle^{\alpha_1} \cdots \langle p_1, b_m\rangle^{\alpha_m} s_1 \cdots s_{\ell-1}\langle p_\ell, b_1\rangle^{\alpha_1} \cdots \langle p_\ell, b_m\rangle^{\alpha_m} s_\ell$$

to P. If there is no $\text{rhs}(p, b)$ in R_T, we add $\langle p, b\rangle \to \varepsilon$ to P. Note that $G_{q,a,u}$ is an extended context-free grammar, polynomial in the size of d_{in} and T. It is easy to see that since d_{in} is non-recursive, $G_{q,a,u}$ is also non-recursive and that $L_{q,a,u} \subseteq L(G_{q,a,u})$.

Our next goal is to prove the following theorem, which states that typechecking reduces to checking inclusion of the language defined by the constructed grammar in the language defined by an RE$^+$-expression.

Theorem 6.18. *For every $q \in Q$, $a \in \Sigma$ and $u \in rhs(q, a)$,*

$$L_{q,a,u} \subseteq L(d_{out}(\sigma)) \textit{ if and only if } L(G_{q,a,u}) \subseteq L(d_{out}(\sigma)),$$

where σ is the label of u.

So, typechecking reduces to testing whether $L(G_{q,a,u}) \subseteq L(d_{\text{out}}(\sigma))$. The latter can be reduced to emptiness testing of the cross-product of the pushdown automaton equivalent to $G_{q,a,u}$ and the DFA accepting the complement of $L(d_{\text{out}}(\sigma))$. All applied constructions and the emptiness test can be done PTIME [HMU01, Sip97].

We now prove Theorem 6.18 in a series of lemmas. The theorem immediately follows from Lemma 6.24. We fix a transducer T and an input and output schema d_{in} and d_{out}.

First, we introduce some additional notation and concepts. We bring an RE$^+$-expressions e in *normal form* as follows. In e, we replace every occurrence of a symbol a and a^+ by $a^{=1}$ and $a^{\geq 1}$, respectively. Next, we repeatedly combine successive terms $a^{=i}a^{=j}$ as $a^{=i+j}$, and $a^{\geq i}a^{=j}$, $a^{=i}a^{\geq j}$ or $a^{\geq i}a^{\geq j}$ as $a^{\geq i+j}$. When no combinations can be made anymore, we say that the resulting expression is *normalized.*

For a normalized RE$^+$-expression $e = a_1^{\theta_1 x_1} \cdots a_n^{\theta_n x_n}$, we denote by $e_{\min}$ the minimal string $a_1^{x_1} \cdots a_n^{x_n}$. A string is *vast with respect to* e, or *e-vast*, when it is of the form $a_1^{y_1} \cdots a_n^{y_n}$ where for every $i = 1, \ldots, n$, $y_i > x_i$ if θ_i is $\geq$ and $y_i = x_i$ otherwise. Note that when $L(e)$ is a singleton, the minimal string is e-vast.

We call two string languages L_1 and L_2 RE$^+$-equivalent, denoted $L_1 \equiv L_2$, if for every RE$^+$-expression e, $L_1 \subseteq L(e) \Leftrightarrow L_2 \subseteq L(e)$. Obviously, this is an equivalence relation.

Lemma 6.19. *For any RE$^+$-expression e and e-vast string e_{vast},*

$$L(e) \equiv \{e_{min}, e_{vast}\}.$$

Proof. Let e be of the form $a_1^{\theta_1 x_1} \cdots a_n^{\theta_n x_n}$. Let f be an arbitrary RE$^+$-expression such that $\{e_{\min}, e_{\text{vast}}\} \subseteq L(f)$. As $e_{\min} \in L(f)$, f is of the form $a_1^{\theta'_1 y_1} \cdots a_n^{\theta'_n y_n}$, where $y_i \leq x_i$ for every $i = 1, \ldots, n$. Moreover, when θ'_i is $=$, then $y_i = x_i$. Since $e_{\text{vast}} = a_1^{z_1} \cdots a_n^{z_n} \in L(f)$, for every $i = 1, \ldots, n$, θ'_i is $\geq$ whenever $z_i > x_i$, and consequently, when θ_i is $\geq$. Therefore, $L(e) \subseteq L(f)$. Clearly, $\{e_{\min}, e_{\text{vast}}\} \subseteq L(f)$ when $L(e) \subseteq L(f)$. This proves the lemma. □

Corollary 6.20. *Let e, f be RE$^+$-expressions. If $L(e) \not\subseteq L(f)$ then either $e_{min} \notin L(f)$ or $e_{vast} \notin L(f)$ for any e-vast string $e_{vast} \in L(e)$.*

Lemma 6.21. *Let e be an RE$^+$-expression and e_{vast} an e-vast string. For any $L \subseteq \Sigma^*$, if $\{e_{min}, e_{vast}\} \subseteq L \subseteq L(e)$ then $L \equiv L(e)$.*

Proof. Let f be an arbitrary RE$^+$-expression such that $L \subseteq L(f)$. Towards a contradiction, assume that $L(e) \not\subseteq L(f)$. But then, according to Corollary 6.20, either $e_{\min} \notin L(f)$ or $e_{\text{vast}} \notin L(f)$. This leads to the desired contradiction. The other direction is trivial since $L \subseteq L(e)$. □

A string language L is *bounded* when there is an RE$^+$-expression $e = a_1^+ \cdots a_\ell^+$ where $a_i \neq a_{i+1}$ for each $i = 1, \ldots, \ell - 1$ such that $L \subseteq L(e)$. We refer to e as a *witness*. Two bounded languages are *bound equivalent* when they share the same witness expression. A language is *unbounded* when it is not bounded.

For every $p \in Q_T$ and $b \in \Sigma$, define $R_{p,b}$ to be the set of strings $\{\text{top}(T^p(t)) \mid t \in L((d_{\text{in}}, b))\}$. Consider the grammar $G_{q,a,u} = (V, \Sigma, P, S)$ as defined earlier in this section. Denote by $L(\langle p, b\rangle)$ the language accepted by $(V, \Sigma, P, \langle p, b\rangle)$ for every non-terminal $\langle p, b\rangle \in V$. That is, $L(\langle p, b\rangle)$ is the grammar $G_{q,a,u}$, but with start symbol $\langle p, b\rangle$. Note that, by definition of $G_{q,a,u}$, for each $p \in Q_T$, $b \in \Sigma$ we have that $R_{p,b} \subseteq L(\langle p, b\rangle)$, and in particular, $L_{q,a,u} \subseteq L(G_{q,a,u})$. Hence, the next lemma immediately follows.

Lemma 6.22.

(1) *For every $p \in Q_T$, $b \in \Sigma$, if $L(\langle p, b\rangle)$ is bounded, then $R_{p,b}$ and $L(\langle p, b\rangle)$ are bound equivalent.*

(2) *If $L(G_{q,a,u})$ is bounded, then $L(G_{q,a,u})$ and $L_{q,a,u}$ are bound equivalent.*

We now show that the languages defined by the constructed grammars are bounded if and only if $R_{p,b}$ and $L_{q,a,u}$ are bounded, respectively.

Lemma 6.23.

(1) For every $p \in Q_T$, $b \in \Sigma$, $L(\langle p, b\rangle)$ is bounded if and only if $R_{p,b}$ is bounded.

(2) $L(G_{q,a,u})$ is bounded if and only if $L_{q,a,u}$ is bounded.

Proof. We only prove (1) as the proof of (2) is similar. As $G_{q,a,u} = (V, \Sigma, P, \langle q, a\rangle)$ is non-recursive, we can prove this lemma by induction on the maximum depth d of derivation trees in $(V, \Sigma, P, \langle p, b\rangle)$.

When $d = 1$, then $\langle p, b\rangle \to w$ is a rule in P for some $w \in \Sigma^*$.

By definition of $G_{q,a,u}$ and $R_{p,b}$, we then have that $L(\langle p, b\rangle) = \{w\} = R_{p,b}$. So, the statement of the lemma follows.

We turn to the induction step. Assume $d > 1$. Let

$$\langle p, b\rangle \to s_0\langle p_1, b_1\rangle^{\alpha_1} \cdots \langle p_1, b_m\rangle^{\alpha_m} s_1 \cdots s_{\ell-1}\langle p_\ell, b_1\rangle^{\alpha_1} \cdots \langle p_\ell, b_m\rangle^{\alpha_m} s_\ell$$

be a rule in P. Then, $R_{p,b}$ is the set

$$\big\{s_0\text{top}\big(T^{p_1}(t_1)\cdots T^{p_1}(t_n)\big)s_1 \cdots s_{\ell-1}\text{top}\big(T^{p_\ell}(t_1)\cdots T^{p_\ell}(t_n)\big)s_\ell \\ \mid t_1 \cdots t_n \in d^{b_1^{\alpha_1}\cdots b_m^{\alpha_m}}\big\}.$$

The latter is equal to

$$\big\{s_0\text{top}\big(T^{p_1}(b_1(h_1^1))\cdots T^{p_1}(b_1(h_1^{k_1}))\cdots T^{p_1}(b_m(h_m^1))\cdots T^{p_1}(b_m(h_m^{k_m}))\big)s_1 \cdots \\ \cdots s_{\ell-1}\text{top}\big(T^{p_\ell}(b_1(h_1^1))\cdots T^{p_\ell}(b_1(h_1^{k_1}))\cdots T^{p_\ell}(b_m(h_m^1))\cdots T^{p_\ell}(b_m(h_m^{k_m}))\big)s_\ell \\ \mid b_1(h_1^1)\cdots b_1(h_1^{k_1})\cdots b_m(h_m^1)\cdots b_m(h_m^{k_m}) \in d^{b_1^{\alpha_1}\cdots b_m^{\alpha_m}}\big\}.$$

As $R_{p,b} \subseteq L(\langle p, b\rangle)$, $R_{p,b}$ is bounded when $L(\langle p, b\rangle)$ is. We next show that if $L(\langle p, b\rangle)$ is unbounded then $R_{p,b}$ is unbounded. We distinguish two cases.

(i) There is an $L(\langle p_i, b_j\rangle)$ which is unbounded. By induction,

$$R_{p_i,b_j} = \{\text{top}(T^{p_i}(t)) \mid t \in d_{\text{in}}^{b_j}\}$$

is unbounded. As for every string $w \in R_{p_i,b_j}$, there are strings w_1, w_2 such that $w_1ww_2 \in R_{p,b}$, we have that the latter language is also unbounded.

(ii) Every $L(\langle p_i, b_j\rangle)$ is bounded, but there are a ℓ, m such that $L(\langle p_\ell, b_m\rangle)$ contains a string with at least two different alphabet symbols and α_m is $+$. Clearly, $L(\langle p, b\rangle)$ is unbounded. By induction, R_{p_ℓ,b_m} is bounded. By Lemma 6.22(1), $L(\langle p_\ell, b_m\rangle)$ and R_{p_ℓ,b_m} are bound equivalent. Therefore, since $L(\langle p_\ell, b_m\rangle)$ contains a string with at least two different alphabet symbols, every string

$$\text{top}(T^{p_\ell}(b_m(h_m^1))), \ldots, \text{top}(T^{p_\ell}(b_j(h_m^{k_m})))$$

contains at least two different alphabet symbols. As k_m can be arbitrarily large, $R_{p,b}$ is unbounded. □

For every $a \in \Sigma$, we define trees $t_a^{\min}$ and t_a^{vast} in $L(d_{\text{in}})$ as follows:

- when $d_{\text{in}}(a) = \varepsilon$ then $t_a^{\min} = t_a^{\text{vast}} = a$; and
- when $d_{\text{in}}(a) = a_1^{\alpha_1} \cdots a_n^{\alpha_n}$ then
  - (i) $t_a^{\min} = a(t_{a_1}^{\min} \cdots t_{a_n}^{\min})$ and
  - (ii) $t_a^{\text{vast}} = a(h_{a_1} \cdots h_{a_n})$, where for every $i = 1, \ldots, n$ we have
    - $h_{a_i} = t_{a_i}^{\text{vast}} t_{a_i}^{\text{vast}}$ when α_i is $+$; and
    - $h_{a_i} = t_{a_i}^{\text{vast}}$, otherwise.

Theorem 6.18 now follows from Lemma 6.24(2).

Lemma 6.24.

(1) For every $p \in Q_T$, $b \in \Sigma$, $L(\langle p, b\rangle) \equiv R_{p,b}$; and

(2) $L(G_{q,a,u}) \equiv L_{q,a,u}$.

Proof. As $G_{q,a,u}$ is non-recursive, we can prove this lemma by induction on the maximum depth d of the derivation trees of $(V, \Sigma, P, \langle p, b\rangle)$.

We prove by induction on d that for any $p \in Q_T$, $b \in \Sigma$,

(IH) if $R_{p,b}$ is bounded, then there is an RE$^+$-expression $r_{p,b}$ such that

1. $L(\langle p, b\rangle) \subseteq L(r_{p,b})$;
2. $(r_{p,b})_{\min} = \text{top}(T^p(t_b^{\min}))$ and $\text{top}(T^p(t_b^{\text{vast}}))$ is $r_{p,b}$-vast. (Note that the latter strings are in $R_{p,b}$.)

We argue that the lemma is proven when (IH) holds. Indeed, by Lemma 6.21 we have that $L(\langle p, b\rangle) \equiv L(r_{p,b}) \equiv R_{p,b}$. When $R_{p,b}$ is unbounded, then so is $L(\langle p, b\rangle)$ (Lemma 6.23(1)). By definition, $L(\langle p, b\rangle) \equiv \langle p, d_{\text{in}}, b\rangle$.

Suppose that $d = 1$, then $\langle p, b\rangle \to w$ for some $w \in \Sigma^*$.

By definition, $L(\langle p, b\rangle) = \{w\} = R_{p,b}$. Define $r_{p,b} = w = r_{p,b}^{\min} = r_{p,b}^{\text{vast}}$. IH now holds.

We turn to the induction step. Assume $d > 1$. Let

$$\langle p, b\rangle \to s_0 \langle p_1, b_1\rangle^{\alpha_1} \cdots \langle p_1, b_m\rangle^{\alpha_m} s_1 \cdots s_{\ell-1} \langle p_\ell, b_1\rangle^{\alpha_1} \cdots \langle p_\ell, b_m\rangle^{\alpha_m} s_\ell$$

be a rule in P. Then

$$R_{p,b} = \{s_0 \text{top}(T^{p_1}(t_1) \cdots T^{p_1}(t_n)) s_1 \cdots s_{\ell-1} \text{top}(T^{p_\ell}(t_1) \cdots T^{p_\ell}(t_n)) s_\ell \\ \mid t_1 \cdots t_n \in d^{b_1^{\alpha_1} \cdots b_m^{\alpha_m}}\}.$$

Assume $R_{p,b}$ is bounded. The latter implies that $L(\langle p, b\rangle)$ is bounded (Lemma 6.23). As the maximum depth of the derivation trees rooted at each $\langle p_i, b_j\rangle$ is $d - 1$, there are corresponding RE$^+$-expressions r_{p_i,b_j} for which the induction hypothesis holds.

Define the RE$^+$-expression $r'_{p,b}$ as

$$s_0 (r_{p_1,b_1})^{\alpha_1} \cdots (r_{p_1,b_m})^{\alpha_m} s_1 \cdots s_{\ell-1} (r_{p_\ell,b_1})^{\alpha_1} \cdots (r_{p_\ell,b_m})^{\alpha_m} s_\ell.$$

We now construct $r_{p,b}$ from $r'_{p,b}$ as follows. For any $i = 1, \ldots, \ell$, $j = 1, \ldots, n$, if α_j is $+$ and $r_{p_i,b_j} = c^{=m}$ or $r_{p_i,b_j} \equiv c^{\geq m}$ then replace $(r_{p_i,b_j})^+$ by $c^{\geq m}$. Finally, normalize the resulting expression. Note that no r_{p_i,b_j} can contain two different alphabet symbols as $L(\langle p_j, b_i\rangle)$ is bounded.

From the construction and the induction hypothesis it follows that $L(\langle p,b\rangle) \subseteq L(r'_{p,b}) \subseteq L(r_{p,b})$, so (1) holds.

It remains to show (2). Clearly,

$$r^{\min}_{p,b} = s_0(r_{p_1,b_1})_{\min} \cdots (r_{p_1,b_m})_{\min} s_1 \cdots s_{\ell-1}(r_{p_\ell,b_1})_{\min} \cdots (r_{p_\ell,b_m})_{\min} s_\ell.$$

Now define

$$(r_{p,b})_{\text{vast}} = s_0(r_{p_1,b_1})^{x_1}_{\text{vast}} \cdots (r_{p_1,b_m})^{x_m}_{\text{vast}} s_1 \cdots s_{\ell-1}(r_{p_\ell,b_1})^{x_1}_{\text{vast}} \cdots (r_{p_\ell,b_m})^{x_m}_{\text{vast}} s_\ell,$$

where for every i we have that $x_i = 1$ if α_i is ε and $x_i = 2$ otherwise. Note that the string $(r_{p,b})_{\text{vast}}$ is $r_{p,b}$-vast.

It remains to show that $(r_{p,b})_{\min} = \text{top}(T^p(t^{\min}_b))$ and $(r_{p,b})_{\text{vast}} = \text{top}(T^p(t^{\text{vast}}_b))$. By induction,

$$\begin{aligned}(r_{p,b})_{\min} &= s_0\text{top}(T^{p_1}(t^{\min}_{b_1}) \cdots T^{p_1}(t^{\min}_{b_m}))s_1 \cdots s_{\ell-1}\text{top}(T^{p_\ell}(t^{\min}_{b_1}) \cdots T^{p_\ell}(t^{\min}_{b_m}))s_\ell \\ &= s_0\text{top}(T^{p_1}(t^{\min}_{b_1} \cdots t^{\min}_{b_m}))s_1 \cdots s_{\ell-1}\text{top}(T^{p_\ell}(t^{\min}_{b_1} \cdots t^{\min}_{b_m}))s_\ell \\ &= \text{top}(T^p(t^{\min}_b))\end{aligned}$$

and we analogously have that

$$\begin{aligned}(r_{p,b})_{\text{vast}} &= s_0\text{top}(T^{p_1}(t^{\text{vast}}_{b_1}))^{x_1} \cdots \text{top}(T^{p_1}(h^{\text{vast}}_{b_m}))^{x_m} s_1 \cdots \\ &\qquad \cdots s_{\ell-1}\text{top}(T^{p_\ell}(h^{\text{vast}}_{b_1}))^{x_1} \cdots \text{top}(T^{p_\ell}(h^{\text{vast}}_{b_m}))^{x_m} s_\ell \\ &= s_0\text{top}(T^{p_1}(h^{\text{vast}}_{b_1}) \cdots T^{p_1}(h^{\text{vast}}_{b_m}))s_1 \cdots \\ &\qquad \cdots s_{\ell-1}\text{top}(T^{p_\ell}(h^{\text{vast}}_{b_1}) \cdots T^{p_\ell}(h^{\text{vast}}_{b_m}))s_\ell \\ &= s_0\text{top}(T^{p_1}(h^{\text{vast}}_{b_1} \cdots h^{\text{vast}}_{b_m}))s_1 \cdots s_{\ell-1}\text{top}(T^{p_\ell}(h^{\text{vast}}_{b_1} \cdots h^{\text{vast}}_{b_m}))s_\ell \\ &= \text{top}(T^p(t^{\text{vast}}_b))\end{aligned}$$

where $h^{\text{vast}}_{b_i} = t^{\text{vast}}_{b_i} t^{\text{vast}}_{b_i}$ when α_i is $+$ and $h^{\text{vast}}_{b_i} = t^{\text{vast}}_{b_i}$ otherwise. □

We have thus obtained the following Theorem:

Theorem 6.25. *$TC[\mathcal{T}_{d,uc}, DTD(RE^+)]$ is in* PTIME.

The simplicity of RE$^+$-expressions seems to be the price to pay for a tractable algorithm for arbitrary transducers. Indeed, the inclusion problem for a class of regular expressions $\mathcal{C}$ can readily be reduced to typechecking with DTD($\mathcal{C}$)s. As it is shown in Chapter 9 that inclusion of obvious extensions of RE$^+$-expressions is coNP-hard, typechecking for the corresponding fragment is coNP-hard. In particular, Chapter 9 discusses expressions of the form $\alpha_1 \cdots \alpha_n$ where all α_i belong to classes (1) a or $a?$, (2) a or a^*, (3) a or $(a_1^+ + \cdots + a_n^+)$, (4) a or $(a_1 \cdots a_n)^+$ (5) a or $(a_1 + \cdots + a_n)^+$ and (6) $(a_1 + \cdots + a_n)$ or a^+. Of course, this argument only holds for setting imposing the same restrictions on input and output schemas.

An interesting question is whether we can also obtain a PTIME typechecking algorithm if we allow expressions of the form α and $\alpha + \varepsilon$ where α is an RE$^+$-expression. This problem remains open.

7

Conclusions and Further Remarks

Motivated by simple transformations obtained by using structural recursion or XSLT, we studied typechecking for top-down XML transformers in the presence of both DTDs and tree automata. Whereas the work on general-purpose XML programming languages like XDuce [HP03] and CDuce [BFC03], studies fast and sound typechecking, we have studied the problem of *complete* typechecking. In our work, we have identified several practically relevant fragments for which complete typechecking can be performed efficiently. As we have also shown that typechecking quickly turns intractable for extensions of our fragments, we have given a quite detailed characterization of the frontier of tractability in our framework. As a result, our work sheds light on when to use fast complete algorithms and when to reside to sound but incomplete ones.

We briefly give an overview of our main results. In our setting, the complexity of the typechecking problem ranges from PTIME to EXPTIME. In particular, when tree automata are used for specifying schema languages, there seems to be not much hope for tractable algorithms. Indeed, the only tractable scenario that we obtained is the one in Theorem 6.8, where we used transformations that are only mild generalizations of relabelings and bottom-up deterministic complete tree automata that use DFAs to represent transition functions. In all other considered scenarios, the typechecking problem remains EXPTIME-hard. The situation differs when we look at DTDs. In an initial study in Chapter 4, we identified three sources of complexity: (1) deletion in the tree transformations; (2) unbounded copying in the tree transformations; and, (3) non-determinism in schema languages. Hence, we only obtained a PTIME typechecking algorithm when no deletion is allowed, the amount of copying is fixed in advance, and when DTD(DFA)s are used to represent schemas.

We considered the complexity of typechecking in the presence of fixed input and/or output schemas in Chapter 5. In comparison with the results in Chapter 4, fixing input and/or output schemas only lowers the complexity in the presence of DTDs and when deletion is disallowed. Here, we see that the complexity is lowered when

1. the input schema is fixed, in the case of DTD(SL)s;
2. the input schema is fixed, in the case of DTD(DFA)s;
3. the output schema is fixed, in the case of DTD(NFA)s; and
4. both input and output schemas are fixed, in all cases.

In all of these cases, the complexity of the typechecking problem is in polynomial time.

It is striking, however, that in many cases, the complexity of typechecking does not decrease significantly when fixing the input and/or output schema, and most cases remain intractable. We have to leave the precise complexity (that is, the PTIME-hardness) of $\text{TC}^i[\mathcal{T}_{nd,uc}, \text{DTD(SL)}]$ as an open problem.

Though the presented results in Chapters 4 and 5 shed some light on precisely which features determine the complexity of typechecking, they fail to identify practically relevant fragments for which typechecking is tractable. Indeed, although it makes sense to limit copying in advance, disallowing deletion completely is not very sensible as deletion occurs in many simple transformations (for example, as in Example 2.15).

Establishing tractable and practically relevant fragments was the topic of Chapter 6. Building further on the results of the previous chapters, we provided a rather complete overview of how the different parameters influence the complexity of the typechecking problem. As the main focus of the chapter is on tractable scenarios, we did not investigate upper bounds for intractable cases.

We identified several practically interesting tractable cases that can be classified depending on the strength of the schema languages. The most liberal setting is where RE^+ expressions suffice to define schema languages: we have PTIME typechecking for all transducers in our framework. Sometimes, however, one needs more expressive regular expressions in schema languages. For instance, to express choice like in (section + table + figure)*. Our results show that there is still a PTIME algorithm when those expressions can be translated in PTIME to DFAs and when one can bound simultaneous copying and deletion. Interestingly, arbitrary deletion without copying *can* be allowed. As copying is usually fairly limited in the simple transformations for which XSLT is used, but unbounded deletion without copying is required for so-called filtering transformations, our result identifies a tractable fragment with potential in practice. Further, we obtained that the XPath axes "/" and "∗" can be added without increasing the complexity. Finally, when deterministic tree automata are required, no copying can be allowed but arbitrary deletion is permitted.

We also showed that none of the above restrictions can be severely relaxed without rendering the typechecking problem intractable. So, for these larger classes of transformations or schema languages, it is more appropriate to develop incomplete or approximate algorithms.

Finding Counterexamples. In practice it is relevant that typechecking algorithms can generate counterexample trees (or a description of them) for instances that it rejects. As the main upper bound theorem in Chapter 6 (Theorem 6.4) reduces the typechecking problem to the emptiness problem for a NTA(NFA) of polynomial size, and since it is possible to generate a description of a tree in the language of an NTA(NFA) in polynomial time (see Proposition 3.18), we can also generate a counterexample tree for the typechecking algorithm in polynomial time.

Further, from the proof of Lemma 6.24 it follows that if an instance of TC[$\mathcal{T}_{d,uc}$, DTD(RE$^+$)] does not typecheck, we either have that t_a^{min} or t_a^{vast} is a counterexample, where a is the start symbol of the DTD. Note that both trees can be easily represented by a polynomial sized extended context free grammar. We have thus obtained the following.

Corollary 7.1. *If an instance of*

- $TC[\mathcal{T}_{trac}, DTD(DFA)]$ *or*
- $TC[\mathcal{T}_{d,uc}, DTD(RE^+)]$

does not typecheck, we can generate a counterexample in PTIME*.*

Notice that, in the case of TC[$\mathcal{T}_{\text{d,uc}}$, DTD(RE$^+$)], testing whether t_a^{min} or t_a^{vast} are counterexamples gives rise to a slightly different typechecking algorithm than the one we exhibited in Section 6.3. However, we believe that algorithms similar to the one in Section 6.3 might also be useful for typechecking with respect to DTDs using other formalisms than RE$^+$-expressions, or for incomplete typechecking algorithms. This is not the case for the algorithm that tests whether t_a^{min} or t_a^{vast} are counterexamples.

Almost Always Typechecking. We end with a note on *almost always typechecking*. We say that an instance of the typechecking problem typechecks *almost always* if the set $\{t \in d_{\text{in}} \mid T(t) \notin d_{\text{out}}\}$ is finite. The latter notion is introduced by Engelfriet and Maneth [EM03]. Since the finiteness problem of NTA(NFA) is decidable in PTIME, according to Proposition 3.18(1), we also have the following.

Corollary 7.2. *Almost always typechecking of $\mathcal{T}_{trac}$ transducers with respect to DTD(DFA)s is in* PTIME*.*

Part II

Foundations of XML Schema Languages

8

Expressive Power of XML Schemas

To date, the most widespread and commonly used XML schemas are DTDs. Their success is mostly of historic nature (DTDs are inherited from XML's predecessor SGML) and partly because of their simplicity. Unfortunately, DTDs are also limited in various ways [DuC02, Jel01, LC00]: they lack modularity, they have few basic types, and the referencing mechanism is quite restricted. Also, specification of unordered data is rather verbose and the expressiveness is severely limited. Many schema languages have been defined to address these shortcomings, to name just a few: XML Schema [SMT05], DSD [KlS00], Relax NG [CM01], Schematron [Jel05]. Among these, XML Schema is the schema language supported by W3C and therefore receives the most attention. Although *XML Schema Definitions (XSDs)* directly address most of the shortcomings of DTDs, and in particular, are more expressive than DTDs, the exact expressiveness of XML Schema, and more importantly, whether the latter is adequate, remains unclear.

The main cause for the limited expressiveness of DTDs is that the content model of an element can not depend on the context of that element but only on the name of its tag. In formal language theoretic terms, DTDs define *local tree languages.* On an abstract level, XML Schema, just like Relax NG, obtains a higher expressive power by extending DTDs with a typing mechanism which allows to define types, possibly recursively, in terms of other types. In particular, and in contrast with DTDs, several types can be associated to the same element name. Whereas Relax NG corresponds to the robust and well-understood formalism of *unranked regular tree languages* (see Section 2.2), XML Schema is less expressive as the XML Schema specification enforces an extra constraint: the *Element Declarations Consistent (EDC)* constraint. It essentially prohibits the occurrence of two different types with the same associated element name in the same content model.

In the present chapter, we provide semantic and syntactical characterizations of the expressive power of schemas with the EDC constraint. This approach is pursued in Section 8.2. In particular, it is shown there that the EDC constraint is intimately connected to the ability to type trees in a top-down fashion. The characterizations provide different viewpoints on the expressiveness of XML Schema. One of them provides a tool that can be used to show that certain constructs are not definable by XML Schema Definitions.

Next, in Section 8.3, we turn to the question whether the EDC constraint is adequate for its purpose. For this, it is important to note that computing the semantics of a schema with respect to a document conceptually involves two tasks: (1) checking conformance with respect to the underlying grammar; and (2) assignment of types (also referred to as schema-validity assessment in [TBMM04]). In the case of XML Schema, the two tasks are a bit entwined as types do not occur in the input document but have to be inferred by the schema validator. The EDC constraint is imposed to facilitate both tasks. Indeed, for a schema admitting EDC, there is a very simple one-pass top-down strategy to validate a document against that schema. Moreover, that strategy assigns a unique type to every element name. So, ambiguous typing (the possibility that there are several valid type assignments) is avoided. From a scientific viewpoint, however, it is not clear whether EDC is the most liberal constraint that allows for efficient validation and unique typing. Therefore, we investigate two more liberal constraints which we discuss next: one-pass preorder typing and top-down typing.

In a streaming environment, one might argue that the most liberal notion is to require that, when processing the document in document order, the type of an element is assigned when its opening tag is met. We refer to the latter requirement as *1-pass preorder typing (1PPT)*. Although EDC ensures 1-pass preorder typing, it is not a necessary condition. More interestingly, it turns out that 1-pass preorder typing is a very robust notion with various clean semantical and syntactical characterizations. In particular, it can be defined in terms of XSDs with restrained competition regular expressions (introduced by Murata et al. [MLMK05]) and by an equivalent syntactical formalism based on contextual patterns.

When documents are not validated in a streaming manner, but in a top-down manner, we could say that in the most liberal notion of typeability, the type of a node does *not* depend on any of its descendants, or on any of the descendants of its siblings. We refer to this requirement as *top-down typeability (TDT)*. We show that the latter notion is strictly more expressive than 1PPT, and we provide various clean semantical and syntactical characterizations. Although the notion of top-down typeability seems quite powerful for its purpose, we show that there exists a natural generalization of top-down deterministic tree automata which has the same expressive power.

In Section 8.5, we turn to static analysis and optimization of schemas. In particular, we consider the complexity of and provide algorithms for the following problems:

1. RECOGNITION: Given an unrestricted XSD, check whether it admits EDC, 1PPT, or TDT.

2. SIMPLIFICATION: Given an unrestricted XSD, check whether it has an equivalent

XSD of a restricted type and compute it, (restricted types being DTD or XSD admitting EDC, 1PPT, or TDT).

The above problems have direct practical applications to optimize schemas and to implement schema validators. Especially, our algorithm for SIMPLIFICATION could be used in schema translator software (like, for instance, Trang [Cla02]), to check whether a given schema can in fact be translated into an equivalent schema in another schema language. To date, Trang, for instance, translates any Relax NG schema directly into an equivalent unrestricted XSD even when the resulting XSD does not admit EDC. Sometimes, however, a more clever translation has to be used to get an equivalent XSD that admits EDC. The complexity of other important static analysis problems for schemas, such as INCLUSION and MINIMIZATION are the topic of Chapters 9 and 10, respectively.

In Section 8.6, we discuss one-pass post-order typing: the type of an element is assigned when visiting its closing tag in a streaming fashion. We show that any unrestricted XSD can be rewritten into an equivalent one that admits one-pass post-order typing.

Related Work. The analysis in the present work is in the same spirit as the one by Brüggemann-Klein and Wood who formalized the determinism constraint of SGML DTDs and provided a workable definition [BKW98]. It is also closely related to the investigations of Murata et al. [MLMK05] who defined the concepts of single-type and restrained competition grammars and provided corresponding validation algorithms but did not discuss semantical characterizations or optimization problems.

8.1 Definitions

8.1.1 XML Schema Languages

As discussed in the introduction, the expressive power of DTDs can be extended by adding types, as, for example, in XML Schema [TBMM04] and Relax NG [CM01]. Types are always from a finite set and each type is associated with a unique element name. The designated start symbol has only one possible type. In the subsequent chapters, we use the notion of *extended DTDs (EDTDs)* to model such schema languages. The notion of extended DTDs was introduced[1] by Papakonstantinou and Vianu [PV00].

Analogously to DTDs, we parameterize the definition of EDTDs by a class of representations $\mathcal{M}$ of regular string languages.

Definition 8.1 ([PV00, BPV04])**.** Let $\mathcal{M}$ be a class of representations of string languages over Δ. An *extended DTD (EDTD)* is a tuple $D = (\Sigma, \Delta, d, s_d, \mu)$, where Δ is a finite set of *types*, (Δ, d, s_d) is a DTD($\mathcal{M}$), and μ is a mapping from Δ to Σ. A tree t is *valid with respect to D* (or *satisfies* D) if $t = \mu(t')$ for some $t' \in L(d)$ (where μ is extended to trees). We call t' a *witness* for t. Again, we denote the set of trees satisfying D by $L(D)$. ◇

[1] Papakonstantinou and Vianu used the term *specialized DTD* as types *specialize* tags. We prefer the term *extended DTD* as it expresses more clearly that the power of the schemas is amplified.

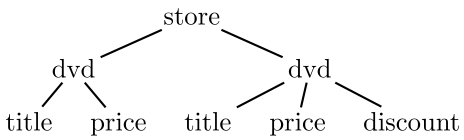


Figure 8.1: Illustration of a tree defined by the EDTD of Example 8.2

Intuitively, a tree satisfies an EDTD if there exists an assignment of types to all nodes such that the typed tree is a derivation tree of the underlying grammar. By EDTD($\mathcal{M}$) we denote the class of EDTDs in which the internal DTD is a DTD($\mathcal{M}$). In the present chapter, we assume that every EDTD is an EDTD(RE), unless otherwise stated.

For notational simplicity, we mostly assume in proofs and formal statements that $\Delta = \{a^i \mid 1 \leq i \leq k_a, a \in \Sigma, i \in \mathbb{N}\}$ for some $k_a \in \mathbb{N}$ and that $\mu(a^i) = a$. When the validity of a tree t is witnessed by a tree t' then we call the label of a node v in t' its *type* with respect to this validation. Analogously to DTDs, we denote by (D, a^i) the extended DTD D, where we replace the DTD (Δ, d, s_d) by (Δ, d, a^i). We say that an EDTD D is *reduced* if d is a reduced DTD.[2] Note that $L((d, a^i))$ contains trees over alphabet Δ, whereas $L((D, a^i))$ contains Σ-trees. The *size* $|D|$ of an EDTD D is defined as $|\Sigma|$ plus the size of the DTD d.

In examples, we denote EDTDs as a set of rules, just as we did before with DTDs.

Example 8.2. The following example displays an EDTD for the tree in Figure 8.1.

$$\begin{array}{lcl} \text{store} & \to & (\text{dvd}^1 + \text{dvd}^2)^*\text{dvd}^2(\text{dvd}^1 + \text{dvd}^2)^* \\ \text{dvd}^1 & \to & \text{title price} \\ \text{dvd}^2 & \to & \text{title price discount} \end{array}$$

Here, *dvd*^{1} and *dvd*^{2} are types that are associated to *dvd* elements, while all other types are associated with the element of the same name: for example, the type *store* corresponds to a *store* element.

Intuitively, *dvd*^{1} defines ordinary DVDs while *dvd*^{2} defines DVDs on discount. The first rule specifies that there has to be at least one DVD on discount. The tree in Figure 8.1 satisfies this EDTD as assigning *dvd*^{1}, and *dvd*^{2} to the left, and right *dvd*-node, respectively, gives a derivation tree of the grammar. ◇

Note that EDTDs have a single root type; only labels below the root can have multiple types. The EDTD of Example 8.2 has the types *dvd*^{1}, *dvd*^{2}, *store*, *title*, *price*, and *discount*. Further, $\mu(\text{dvd}^1) = \mu(\text{dvd}^2) = \text{dvd}$ and μ is the identity, otherwise.

In Figure 8.2, a fragment of an XSD corresponding to the rule for *store* is depicted. We note that the XSD is not syntactically correct because it violates the *Element Declarations Consistent (EDC)* constraint. Roughly, the latter constraint forbids the occurrence of different types associated to the same element name in the same content model. So, the occurrence of both `dvd1` and `dvd2` associated to the same element name `dvd` is a clear violation of this constraint. We formalize this constraint in Definition 8.3 and provide several equivalent characterizations in Theorem 8.16.

[2] Recall the definition of a reduced DTD from page 14

```
<xs:element name="store">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="dvd" type="dvd1"/>
        <xs:element name="dvd" type="dvd2"/>
      </xs:choice>
      <xs:element name="dvd" type="dvd2"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="dvd" type="dvd1"/>
        <xs:element name="dvd" type="dvd2"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 8.2: A fragment of an XSD (violating EDC) corresponding to the EDTD of Example 8.2.

We call a tree language *homogeneous* if all its trees have the same root label. It should be clear that EDTDs, just as DTDs, can only express homogeneous tree languages. From a structural perspective, EDTDs express exactly the *homogeneous regular tree languages*, a similarly robust class as the regular string languages [BKMW01]. In particular, EDTDs are as expressive as unranked tree automata. For definitions of such automata we refer the reader to Chapter 2. It should be noted that the formal underpinnings of the schema language Relax NG are also based upon regular tree languages. As we will only talk about homogeneous tree languages in the present chapter, we will mostly drop the term homogeneous. For the purpose of this dissertation, whenever we say *(homogeneous) regular tree language* T in the sequel, it can be interpreted as *there is an EDTD* D *such that* $L(D) = T$.

Murata et al. [MLMK05] presented a formalization of the EDC rule, which we state here in terms of EDTDs.[3] Roughly, the constraint forbids the occurrence in the same definition of elements with the same name but different types. For instance, the XSD of Figure 8.2 is not allowed as the two types `dvd1` and `dvd2` occur in the same rule.

Definition 8.3. Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD. A regular language L over Δ is *single-type* if there do not exist strings $w_1 a^i v_1$ and $w_2 a^j v_2$ in L, with $\mu(a^i) = \mu(a^j)$ and $i \neq j$. We say that D is a *single-type EDTD (EDTD*st*)* when every regular language $L(d(a^i))$ is single-type. ◇

Hence, when L is a language defined by a regular expression r, we have that L is single-type if and only if no two symbols a^i and a^j occur in r with $i \neq j$. We say that a regular expression is single-type if it defines a single-type language.

[3] Murata et al. used the equivalent model of tree grammars instead of EDTDs [MLMK05].

Example 8.4. The EDTD from Example 8.2 is not single-type as both dvd^1 and dvd^2 occur in the rule for *store*. An example of a single-type EDTD is given below:

$$\begin{array}{lcl}
\text{store} & \to & \text{regulars discounts} \\
\text{regulars} & \to & (\text{dvd}^1)^* \\
\text{discounts} & \to & \text{dvd}^2\ (\text{dvd}^2)^* \\
\text{dvd}^1 & \to & \text{title price} \\
\text{dvd}^2 & \to & \text{title price discount}
\end{array}$$

Although there are still two element definitions dvd^1 and dvd^2, they can only occur in different rules. ◇

We recall the definition of *restrained competition* EDTDs introduced by Murata et al. [MLMK05]. They can be seen as a generalization of single-type EDTDs.

Definition 8.5. Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD. A regular language L over Δ *restrains competition* if there are no strings $w_1 a^i v_1$ and $w_2 a^j v_2$ in L with $\mu(w_1) = \mu(w_2)$, $\mu(a^i) = \mu(a^j)$, and $i \neq j$. An EDTD is *restrained competition (EDTD*^{rc}*)* when every regular language $L(d(a^i))$ restrains competition. ◇

A regular expression over Δ restrains competition if it defines a language which restrains competition.

Intuitively, a restrained competition regular language ensures that, when visiting the children of a node from left to right, it is always clear which type is associated to each node without seeing its right siblings. So, single-type implies restrained competition.

Example 8.6. The following is an example of a restrained competition EDTD that is not single-type nor has an equivalent single-type EDTD.

$$\begin{array}{lcl}
\text{store} & \to & (\text{dvd}^1)^*\ \text{discounts}\ (\text{dvd}^2)^* \\
\text{discounts} & \to & \varepsilon \\
\text{dvd}^1 & \to & \text{title price} \\
\text{dvd}^2 & \to & \text{title price discount}
\end{array}$$

The expression $(\text{dvd}^1)^*$ discounts $(\text{dvd}^2)^*$ is restrained competition as types can be assigned from left to right: each time a *dvd*-element is read, it has type dvd^1 when *discounts* has not been met yet, and type dvd^2, otherwise.

In contrast, the expression $(\text{dvd}^1 + \text{dvd}^2)^*\text{dvd}^2(\text{dvd}^1 + \text{dvd}^2)^*$ of Example 8.2 is not restrained competition as the strings dvd^2 and dvd^1dvd^2 are both defined by the regular expression but dvd^1 and dvd^2 are associated to the same element name. Here, $w_1 = \varepsilon$, $a^i = \text{dvd}^2$, $a^j = \text{dvd}^1$, $v_1 = \varepsilon$, and $v_2 = \text{dvd}^2$. ◇

We extend the restrained competition restriction to a natural generalization of the notion of top-down determinism for tree automata. However, it should be noted that, in the straightforward generalization of top-down determinism, the automaton only uses the label and the state of the current symbol to assign states to the children [BKMW01]. Here, we also take the labeling of the children into account.

Definition 8.7. Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD. A regular language L over Δ is *unambiguously typed* if there are no strings $v \neq w \in L$ with $\mu(v) = \mu(w)$. An EDTD is *top-down deterministic (EDTD^{td})* when every regular language $d(a^i)$ is unambiguously typed. $\diamond$

Notice that L is unambiguously typed if and only if there are no strings $w_1 a^i v_1$ and $w_2 a^j v_2$ in L with $\mu(w_1) = \mu(w_2)$, $\mu(a^i) = \mu(a^j)$, $\mu(v_1) = \mu(v_2)$ and $i \neq j$. Hence, the unambiguously typed restriction is a natural extension of the restrained competition restriction.

A regular expression over Δ is unambiguously typed if it defines a language which is unambiguously typed.

Example 8.8. The following is an example of a top-down deterministic EDTD that is not restrained competition nor has an equivalent restrained competition EDTD.

$$\begin{array}{lcl} \mathrm{store} & \to & (\mathrm{dvd}^1)^*\mathrm{dvd}^2 \\ \mathrm{dvd}^1 & \to & \mathrm{title\ price} \\ \mathrm{dvd}^2 & \to & \mathrm{title\ price\ discount} \end{array}$$

The expression $(\mathrm{dvd}^1)^*\mathrm{dvd}^2$ is unambiguously typeable as types can always be uniquely assigned once the entire sequence of elements is read: all the *dvd*-elements have type dvd^1, excepts the last one, which has type dvd^2.

In contrast, the expression $(\mathrm{dvd}^1 + \mathrm{dvd}^2)^*\mathrm{dvd}^2(\mathrm{dvd}^1 + \mathrm{dvd}^2)^*$ of Example 8.2 is not unambiguously typeable: the strings $\mathrm{dvd}^1\mathrm{dvd}^2$ and $\mathrm{dvd}^2\mathrm{dvd}^1$ are both defined by the regular expression but dvd^1 and dvd^2 are associated to the same element name. Here, $w_1 = \varepsilon$, $a^i = \mathrm{dvd}^1$, $a^j = \mathrm{dvd}^2$, $v_1 = \mathrm{dvd}^2$, and $v_2 = \mathrm{dvd}^1$. $\diamond$

Our top-down deterministic EDTDs are in fact equally expressive (on homogeneous tree languages) as the top-down deterministic tree automata recently defined by Cristau et al. [CLT05]. This follows from one of the characterizations in Theorem 8.22.

The classes of tree languages defined by the grammars introduced above are included as follows: DTD $\subsetneq$ EDTD$^{\mathrm{st}}$ $\subsetneq$ EDTD$^{\mathrm{rc}}$ $\subsetneq$ EDTD$^{\mathrm{td}}$ $\subsetneq$ EDTD (see [MLMK05] and the remainder of this chapter).

8.1.2 Properties of DTDs

In this section, we reconsider some simple properties of DTDs. In particular, we discuss validation and a closure property. The latter property provides a tool to prove that certain tree languages are not definable by DTDs, and, hence, gives insight into the expressiveness of the latter. In Section 8.2, Section 8.3, and Section 8.4, we discuss similar closure properties for the restrictions on EDTDs defined in Section 8.1.1.

Validation of DTDs

Validation of a document against a DTD d simply boils down to testing local consistency: does the string formed by the labels of the children of every a-labeled element satisfy the associated regular expression $d(a)$? No notion of typing is available. To ensure efficient validation, regular expressions in right-hand sides of rules are required to be *deterministic* (Appendix E in [BPSM$^+$04]). We note that the latter notion is also referred to as *one-unambiguous* [BKW98]. Intuitively, a regular expression is deterministic if, when processing the input from left to right, it is always determined which symbol in the expression matches the next input symbol. We discuss the latter notion a bit more formally as it returns in the specification of XML Schema in the form of the *Unique Particle Attribution (UPA)* rule. For a regular expression r over elements, we denote by $\overline{r}$ the regular expression obtained from r by replacing, for each i, the ith a-element in r (counting from left to right) by a_i. For example, when $r = (a+b)^*ac(b+c)^*$, then $\overline{r}$ is $(a_1+b_1)^*a_2c_1(b_2+c_2)^*$.

Definition 8.9. A regular expression r is *one-unambiguous* if there are no strings wa_iv and wa_jv' in $L(\overline{r})$ such that $i \neq j$. ◇

Example 8.10. The regular expression $ab + aa$ is not one-unambiguous. Indeed, $L(a_1b_1 + a_2a_3)$ contains the strings a_1b_1 and a_2a_3, and $1 \neq 2$. Here, w is the empty string. The expression $a(b+a)$ on the other hand, is one-unambiguous. Indeed, $L(a_1(b_1+a_2))$ only contains the strings a_1b_1 and a_1a_2, and it is easy to verify that the above condition is not violated. Note that $a(b+a)$ denotes the same language as $ab + aa$. ◇

In contrast to what the previous example might suggest, Brüggemann-Klein and Wood showed that not every regular expression can be rewritten into an equivalent one-unambiguous one [BKW98]. So the allowed class of regular expressions is a strict subset of the class of all regular languages.

Subtree exchange

Papakonstantinou and Vianu [PV00] provided a characterization of the structural expressive power of DTDs. They considered a more relaxed notion of DTDs without the requirement of one-unambiguous regular expressions. They show that a regular tree language T of trees is definable by such a DTD if and only if T has the following closure property: if two trees t_1 and t_2 are in T, and there are two nodes v_1 in t_1 and v_2 in t_2 with the same label, then the trees obtained by exchanging the subtrees rooted at v_1 and v_2 are also in the set T. We refer to this property as *label-guarded subtree exchange* and we illustrate it in Figure 8.3.

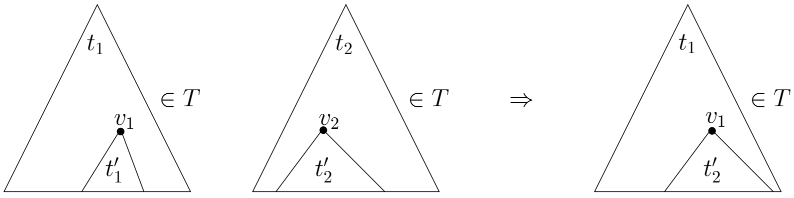


Figure 8.3: Label-guarded subtree exchange. Nodes v_1 and v_2 are both labeled with the same label.

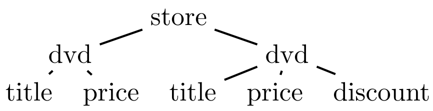
Because of this characterization, the classes of XML documents defined by DTDs are also referred to as *local classes* (see, for example, [MLMK05]): the content of a node only depends on the label of that node and hence, the dependency is local. The characterization can be used to prove that certain languages can *not* be expressed by a DTD as explained in the following example.

Example 8.11. Consider the following DTD consisting of the following rules.

$$\begin{array}{lll} \text{store} & \to & \text{dvd dvd}^* \\ \text{dvd} & \to & \text{title price (discount} + \varepsilon) \end{array}$$

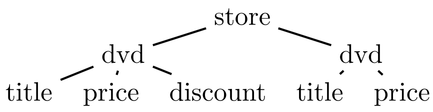
Suppose that we want to put the extra constraint on the DTD requiring the presence of at least one DVD on discount. Then we get a language that is *not* definable by a DTD anymore. We can prove this by applying the above characterization. Indeed, the trees

$t_1 :=$

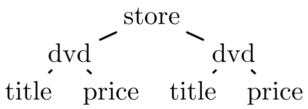


and

$t_2 :=$



are in the language, but the tree



which is obtained from t_1 by replacing its second subtree by the second subtree of t_2, is *not* in the language. ◇

In the remainder of the chapter, we will characterize single-type, restrained competition, and top-down deterministic EDTDs in an analogous manner.

8.1.3 Characterizations

The present section introduces some equivalent characterizations of the previously defined schema languages. We will prove the equivalence in Sections 8.2, 8.3, and 8.4.

Several of these characterizations make use of strings formed by the ancestors (and sometimes also their siblings) of nodes in trees. We therefore introduce the following notions. Let t be a tree and v be a node in t. By $\text{ch-str}^t(v)$ we denote the string formed by the children of v, that is, $\text{lab}^t(v1)\cdots\text{lab}^t(vn)$ if v has n children. By $\text{anc-str}^t(v)$ we denote the *ancestor-string* of v in t, that is, the string formed by the labels on the path from the root to v, that is,

$$\text{lab}^t(\varepsilon)\text{lab}^t(i_1)\text{lab}^t(i_1i_2)\cdots\text{lab}^t(i_1i_2\cdots i_k)$$

where $v = i_1 i_2 \cdots i_k$. By $\text{l-sib-str}^t(v)$ we denote the *left-sibling-string* of v in t, formed by the labels of the left siblings of v, that is, $\text{lab}^t(u1) \cdots \text{lab}^t(uk)$ where $v = uk$. By $\text{anc-sib-str}^t(v)$ we denote the *ancestor-sibling-string* of v in t, that is, the string

$$\text{l-sib-str}^t(\varepsilon)\#\text{l-sib-str}^t(i_1)\# \cdots \#\text{l-sib-str}^t(i_1 i_2 \cdots i_k)$$

formed by concatenating the left-sibling strings of all ancestors starting from the root. By $\text{r-sib-str}^t(v)$ we denote the *right-sibling-string* of v in t, formed by the labels of the right siblings of v, that is, $\text{lab}^t(uk) \cdots \text{lab}^t(un)$ where $v = uk$ and u has n children. By $\text{anc-all-sib-str}^t(v)$ we denote the *ancestor-all-sibling-string* of v in t, that is, the string

$$\text{lab}^t(\varepsilon)\$\text{lab}^t(\varepsilon)\#\text{l-sib-str}^t(i_1)\$\text{r-sib-str}^t(i_1)\# \cdots$$
$$\cdots \#\text{l-sib-str}^t(i_1 i_2 \cdots i_k)\$\text{r-sib-str}^t(i_1 i_2 \cdots i_k)\#$$

formed by concatenating the left-sibling strings and right-sibling strings of all ancestors starting from the root. We assume that the symbols \$ and # are not in Σ. For readability, we will use the term "spine" to refer to the ancestor-all-sibling-string of a node. We also denote $\text{anc-all-sib-str}^t(v)$ by $\text{spine}^t(v)$.

The just defined notions are illustrated in Figure 8.4. When the tree t is clear from the context, we usually omit the superscript t.

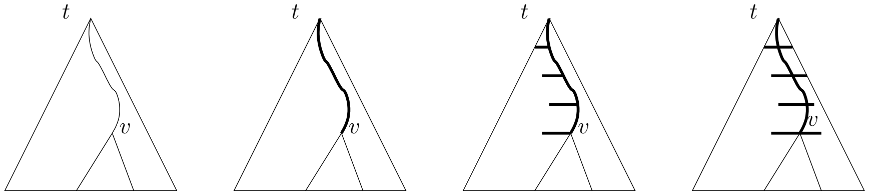


Figure 8.4: From left to right: a tree t, the ancestor-string of v, the ancestor-sibling-string of v, and the spine of v in t.

Moreover, we denote by $t_1[u \leftarrow t_2]$ the tree obtained from a tree t_1 by replacing the subtree rooted at node u of t_1 by t_2. By $\text{subtree}^t(u)$ we denote the subtree of t rooted at u.

Definition 8.12. We say that an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ *has ancestor-based types* if there is a function $f : \Sigma^* \to \Delta$ such that, for each tree $t \in L(D)$,

- t has exactly one witness t', and
- t' results from t by assigning to each node v the type $f(\text{anc-str}^t(v))$.

We say that D *has ancestor-sibling-based types* (respectively, *spine-based types*) if the above holds with $\text{anc-str}^t(v)$ replaced by $\text{anc-sib-str}^t(v)$ (respectively, $\text{spine}^t(v)$). $\diamond$

Definition 8.13. A tree language T is *closed under ancestor-guarded subtree exchange* if the following holds. Whenever for two trees $t_1, t_2 \in T$ with nodes u_1 and u_2, respectively, $\text{anc-str}^{t_1}(u_1) = \text{anc-str}^{t_2}(u_2)$ then $t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)] \in T$. We say that T is *closed under ancestor-sibling-guarded subtree exchange* (respectively, *closed under spine-guarded subtree exchange*) if the above holds with anc-str replaced by anc-sib-str (respectively, spine). $\diamond$

These definitions are illustrated in Figures 8.5, 8.6, and 8.7.

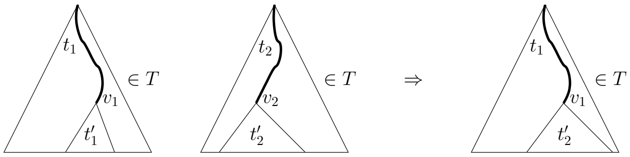


Figure 8.5: Ancestor-guarded subtree exchange.

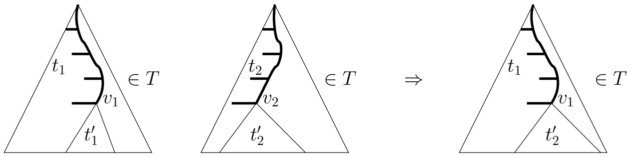


Figure 8.6: Ancestor-sibling-guarded subtree exchange.

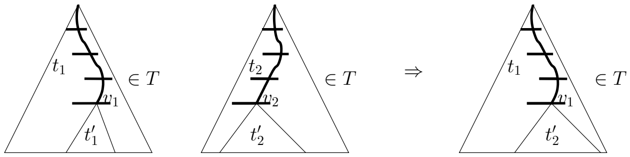


Figure 8.7: Spine-guarded subtree exchange.

Definition 8.14. An *ancestor-based schema* S is a pair (Σ, R), where R is a set of rules of the form $r \to s$, where r and s are regular expressions over Σ. A tree t satisfies S if for every node v there is some $r \to s$ in R such that $\text{anc-str}^t(v) \in L(r)$ and $\text{ch-str}^t(v) \in L(s)$. We say that S is an *ancestor-sibling-based schema* (respectively, *spine-based schema*) if the above holds with anc-str replaced by anc-sib-str (respectively, spine). $\diamond$

Definition 8.15. Let T be a set of trees. We say that T *can be characterized by ancestor-based patterns* if there is a regular string language L over $\Sigma \uplus \{\#, \$\}$ such that, for every tree t, we have that $t \in T$ if and only if $P_{\text{anc}}(t) \subseteq L$, where $P_{\text{anc}}(t) = \{\text{anc-str}(v)\#\text{ch-str}(v) \mid v \in t\}$. We say that T *can be characterized by ancestor-sibling-based patterns* (respectively, *can be characterized by spine-based patterns*) if the above holds with anc-str and P_{anc} replaced by anc-sib-str and $P_{\text{anc-sib}}$ (respectively, spine and P_{spine}). $\diamond$

8.2 The Equivalence Theorem for Schemas with EDC

As already mentioned in the introduction, XML Schema does not capture all tree languages that can be described by extended DTDs (that is, the regular tree languages). In particular, the EDC (Element Declarations Consistent) and the UPA (Unique Particle Attribution) constraint must be fulfilled. In Definition 8.3, EDC was formalized in the form of single-type EDTDs. The present section gives several equivalent characterizations of the resulting class of tree languages. Together, these characterizations provide a clear view of the effect of the EDC constraint on the expressiveness of XSDs and typing algorithms.

Theorem 8.16. *For a homogeneous regular tree language T the following conditions are equivalent.*

(a) T is definable by a single-type EDTD.

(b) T is definable by an EDTD with ancestor-based types.

(c) T is closed under ancestor-guarded subtree exchange.

(d) T is definable by an ancestor-based schema.

(e) T can be characterized by ancestor-based patterns.

Proof. We prove (a)⇒(b)⇒(c)⇒(a) and (a)⇒(d)⇒(e)⇒(b).

Let c denote the unique root label of the trees in T.

(a) ⇒ (b): Let T be defined by the single-type EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$. Then define f inductively as follows: $f(\mu(s_d)) = s_d$. Further, for any string $w \cdot a \cdot b$ with $w \in \Sigma^*$ and $a, b \in \Sigma$, $f(w \cdot a \cdot b) = b^j$ where b^j occurs in $d(a^i)$ and $f(w \cdot a) = a^i$. As $d(a^i)$ is single-type, f is well-defined and induces a unique typing. Thus, the requirements of Definition 8.12 are satisfied.

(b) ⇒ (c): Let T be defined by an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ with ancestor-based types. Let t_1, t_2 be in T and let u_1 and u_2 be nodes in t_1 and t_2, respectively, with $\text{anc-str}^{t_1}(u_1) = \text{anc-str}^{t_2}(u_2)$. Let t_1' and t_2' be the unique witnesses for t_1 and t_2, respectively. As the label of u_1 in t_1' and the label of u_2 in t_2' are determined by $\text{anc-str}^{t_1}(u_1) = \text{anc-str}^{t_2}(u_2)$, they are the same. Hence, by replacing the subtree rooted at u_1 in t_1' with the subtree rooted at u_2 in t_2' we get a tree $t' \in L(d)$. Therefore, $\mu(t') = t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)]$ is in T, as required.

(c) ⇒ (a): Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD defining a tree language closed under ancestor-guarded subtree exchange. Our aim is to construct a single-type EDTD E such that $L(E) = L(D)$.

As explained in the beginning of this section, we assume without loss of generality that D only contains useful types, that is, each type occurs in the witness of some tree in $L(D)$. For each type of D, choose a fixed tree, which is the subtree rooted at some node of this type in a tree in $L(D)$.

We will make use of the following general property of EDTDs:

(†) If t_1, t_2 are trees in $L(D)$ with witnesses t_1', t_2', respectively, such that v_1 in t_1 and v_2 in t_2 have the same type in t_1' and t_2', respectively, then the tree obtained from t_1 by replacing the subtree of v_1 with the subtree of v_2 in t_2 is in $L(D)$.

This property should not be confused with the subtree-exchange properties defined above which do not concern types at all.

For a string $w \in \Sigma^*$ and $a \in \Sigma$ let types(wa) be the set of all types a^i, for which there is a tree t with witness tree $t' \in L(d)$ and a node v in t such that anc-str$^t(v) = wa$ and the type of v in t' is a^i. For each $a \in \Sigma$, let $\tau(D, a)$ be the set of all nonempty sets types(wa), with $w \in \Sigma^*$. Clearly, each $\tau(D, a)$ is finite.

We next define $E = (\Sigma, \Delta_E, e, s_d, \mu_E)$. Its set of types is $\Delta_E := \bigcup_{a\in\Sigma} \tau(D, a)$. Note that $s_d \in \Delta_E$. For every $\tau \in \tau(D, a)$, set $\mu_E(\tau) = a$. In e, the right-hand side of the rule for each types(wa) is the disjunction of all $d(a^i)$ for $a^i \in$ types(wa), with each b^j in $d(a^i)$ replaced by types(wab). It should be noted that by (†), the definition of the rules of e does not depend on the actual choice of wa.

Clearly, E is single-type and $L(D) \subseteq L(E)$. Thus it only remains to show $L(E) \subseteq L(D)$.

To this end, let $g \in L(E)$ and let g' be a witness. We call a set S of nodes of g *well-formed* if (1) for each node $v \in S$ all its ancestors are in S and (2) if a child u of a node v is in S then all children of v are in S. The singleton set S_ε containing the root is well-formed.

We say that a tree t_2 *agrees* with a tree t_1 on an ancestor-closed set S_1 of nodes of t_1, if S_1 can be mapped to a well-formed S_2 by a mapping m which respects the child-relationship, the order of siblings and the labels of nodes.

As all trees in $L(D)$ and $L(E)$ have the same root label, there exists a tree $t_1 \in L(D)$ which agrees with g on S_ε. To complete the proof of "(c) ⇒ (a)" it is sufficient to prove the following.

Claim 8.17. *If there exists a tree $t_1 \in L(D)$ which agrees with g on a well-formed set $S \subsetneq Nodes(t)$ then there exists $t_2 \in L(D)$ which agrees with g on a well-formed set which is a strict superset of S.*

For the proof of this claim, let $wa =$ anc-str$^g(v)$, for some node $v \in S$ whose children are not in S. Let t_1 be as stated and let t_1' be its witness. Let a^i be the type of the node $m(v)$ corresponding to v in t_1'.

By construction of E the right-hand side of the rule for types(wa) is a disjunction over the (adapted) right-hand sides of rules of D. Let a^j be such that the children of v are typed in g' according to a disjunct derived from the rule for a^j. Thus, in particular, $a^j \in$ types(wa). Thus, there is a tree $t_3 \in L(D)$ with a node u such that anc-str$^{t_3}(u) = wa$ and the type of u is a^j in the witness t_3' for t_3.

Let, for each child v_1 of v in g, a type $f(v_1)$ be chosen such that ch-str(v) matches $d(a^j)$ with these types. Let t_4 be obtained from t_3 by (1) removing everything below u, (2) adding the children of v below u, and (3) adding for each child v_1 the fixed subtree chosen for $f(v_1)$. Clearly, by (†), $t_4 \in L(D)$. Furthermore, by the ancestor-closed subtree exchange property, the tree t_2 resulting from t_1 by replacing the subtree rooted at $m(v)$ by the subtree of t_4 rooted at u is in $L(D)$, too. This completes the proof of the claim and thus of "(c) ⇒ (a)".

(a) ⇒ (d): Let T be defined by a single type EDTD $D = (\Sigma, \Delta, d, c^0, \mu)$ with $\cdot, \bot \notin \Delta$. Let A be a DFA over Σ with state set $Q = \Delta \cup \{\cdot, \bot\}$, initial state $\cdot$ and transition function $\delta : Q \times \Sigma \to Q$. Let $\delta(a^i, b)$ equal the unique b^j occurring in

$d(a^i)$ if such a symbol exists, otherwise $\perp$. Furthermore, $\delta(\cdot, c) = c^0$. Note that the single-type property ensures that A is deterministic and well-defined.

Let $S = (\Sigma, R)$ be the ancestor-based schema with rules of the form $r_{a,i} \rightarrow \mu(d(a^i))$, where $r_{a,i}$ is a regular expression describing the set $\{w \mid \delta^*(\cdot, w) = a^i\}$ of strings which bring A into state a^i. Of course, the languages $L(r_{a,1}), \ldots, L(r_{a,k_a})$ are all disjoint where $\{a^1, \ldots, a^{k_a}\}$ are the symbols mapped to a by μ. Note that we also denote by μ the homomorphic extension of μ to regular expressions $d(a^i)$.

It remains to show that S defines the same set of trees as D. Let t be in $L(D)$ and t' be a witness. It is easily shown by induction that, for each node v of t', $\text{lab}^{t'}(v) = \delta^*(\cdot, \text{anc-str}^t(v))$. Hence, for each node v labeled with a^i, the rule of S responsible for v is $r_{a,i} \rightarrow \mu(d(a^i))$ and can therefore be applied. The proof of the opposite inclusion is similar.

(d) $\Rightarrow$ (e): Let T be defined by the ancestor-based schema $S = (\Sigma, R)$. Then T can be characterized by the set $L = \{u\#v \mid u \in L(r), v \in L(s), r \rightarrow s \in R\}$. By definition, for every tree $t \in T$ it holds that $P_{\text{anc}}(t) \subseteq L$. For the other direction, let t be a tree which is not in T. Hence, there is a node w in t such that either there is no rule $r \rightarrow s$ in R with $\text{anc-str}(w) \in L(r)$ or for every such triple $\text{ch-str}(w) \notin L(s)$. This implies that $\text{anc-str}(w)\#\text{ch-str}(w) \notin L$. Therefore, a tree t is in T if and only if $P_{\text{anc}}(t) \subseteq L$ and we are done.

(e) $\Rightarrow$ (b): Let T be characterized by ancestor-based patterns using the language L. Let $A = (\Sigma, Q, \delta, s, F)$ be a DFA for L. Define $D = (\Sigma, \Delta, d, s_d, \mu)$ as follows. Δ is the set of all pairs (a, q), where $a \in \Sigma$ and $q \in Q$ and $\mu((a, q)) = a$. We let $d((a, q))$ be a regular expression describing all strings $(b_1, q_1) \cdots (b_n, q_n)$, for which A accepts $\#b_1 \cdots b_n$ when started from state q and $\delta(q, b_i) = q_i$, for every $i \leq n$. The start symbol s_d is (c, q') where $\delta(s, c) = q'$. By construction, D is single-type and therefore also has ancestor-based types. It is easy to see that $L(D)$ defines T. Indeed, when $t \in T$, let t' be obtained from t by relabeling every inner node v labeled a by (a, q) where $q = \delta^*(s, \text{anc-str}^t(v))$ then $t' \in L(D)$ and $t = \mu(t'))$. Conversely, let $t' \in L(D)$. Then, for every node u of $t = \mu(t')$, $\text{anc-str}^t(u)\#\text{ch-str}^t(u) \in P_{\text{anc}}(t)$ by construction. □

As an immediate consequence of Theorem 8.16, the language we considered in Example 8.11 is not definable by a single-type EDTD. Note that the counterexample can be constructed in exactly the same manner. On the other hand, the language defined by the single-type EDTD in Example 8.4 is not definable by a DTD, so single-type EDTDs are strictly more expressive than DTDs. As a matter of fact, it can be decided whether a given EDTD is equivalent to a single-type EDTD. For instance, the non single-type EDTD $a \rightarrow b^1 b^2$, $b^1 \rightarrow c$, $b^2 \rightarrow c$ is clearly equivalent to the DTD (and, hence, single-type EDTD) consisting of the rules $a \rightarrow bb$ and $b \rightarrow c$. The complexity of this problem is considered in Section 8.5.

The importance of the characterization of single-type EDTDs by a subtree-exchange property stems from the fact that inexpressibility results can be formally proved rather than vaguely stated. For instance, a shortcoming recently attributed to XSDs is their inability to express certain co-constraints [CMV04]. An example of such a co-constraint is the following: a *store*-element can only have a *dvd*-element with *discount* child if it also has a *dvd*-element without a *discount* child. Using the ancestor-guarded

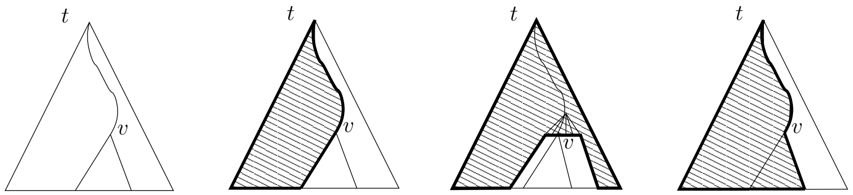


Figure 8.8: From left to right: a tree t, the preceding of v, the pruned envelope of v, and the preceding-subtree of v in t.

subtree exchange property, it is very easy to prove that this co-constraint cannot be expressed with XSDs. Indeed, the counterexample is constructed from t_1 in Example 8.11 by replacing its first subtree by the first subtree of t_2.

8.3 The Equivalence Theorem for 1-Pass Preorder Typeable Schemas

As mentioned before, the expressive power of EDTDs (and Relax NG) corresponds to the well-understood and very robust class of regular tree languages. However, this expressive power comes at a price. Although it can be determined in linear time whether a tree satisfies a given EDTD, the way to do that is sometimes at odds with the way one would like to process XML documents. More concretely, it requires to process documents in a bottom-up fashion where the type(s) of an element is only determined after reading its content. In the context of streaming XML data or for SAX-based processing, that is, when processing an XML document as a SAX-stream of opening and closing tags, it is more desirable to determine the type of an element at the time its opening tag is met. If an EDTD fulfills this requirement we say it is *1-pass preorder typeable* (1PPT). Note that not every EDTD admits 1PPT. Consider the example $a \to b^1 + b^2$, $b^1 \to c$, $b^2 \to d$ and the document `<a><b><d/></b></a>`. The type of `b` depends on the label of its child. It is hence impossible to assign a type to `b` when its opening tag `<b>` is met, that is, without looking at its child. An alternative formulation of 1PPT is that the type of an element cannot depend on anything occurring in document order after the opening tag of that element. Hence, we require that a type is uniquely determined by the preceding of an element (Figure 8.8). On top of one-pass preorder typeability, this notion therefore also enforces the attribution of a unique type to every element. The latter is, for instance, not the case for Relax NG which allows ambiguous typing as in the grammar $a \to b^1 + b^2$, $b^1 \to c$, and $b^2 \to c$, where b can both be assigned type b^1 and b^2 in the tree $a(b(c))$.

We formalize the notion of 1PPT in terms of preceding-based types in analogy to the ancestor-based types of Definition 8.12. The *preceding* of a node v in t is the tree resulting from t by removing everything below v, all right siblings of v's ancestors and of v, and their respective subtrees (see Figure 8.8). In other words, the preceding of v in t is the subtree of t consisting of all nodes that are before v in document order, and v itself. We denote the preceding of v by $\text{preceding}^t(v)$.

Definition 8.18. We say that an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ *is 1-pass preorder typeable* (1PPT) or *has preceding-based types* if there is a function $f : \mathcal{T}_\Sigma \to \Delta$ such that, for each tree $t \in L(D)$,

- there is exactly one witness t', and
- t' results from t by assigning to each node v the type $f(\text{preceding}^t(v))$. ◇

Theorem 8.16 characterizes single-type EDTDs precisely as the class of EDTDs with ancestor-based types. Therefore, every single-type EDTD admits 1PPT. The converse, however, is not true. Consider for example the following EDTD which is not single-type: $a \to b^1 b^2$, $b^1 \to c$, $b^2 \to d$. Nevertheless, the EDTD admits 1PPT. Indeed, it is easy to see that the EDTD only defines the singleton `<a><b><c/></b><b><d/></b></a>`. The rule for a says that the first b-child needs to be typed b^1 and the second b-child needs to be typed b^2. For each of the b's in the document, it can be easily determined whether it is the first or second child of a by investigating its preceding (see Figure 8.8). Hence, the notion of single-type EDTDs allows for efficient unique typing, but does not capture all of 1PPT EDTDs.

We are now ready to prove the following theorem.

Theorem 8.19. *For a homogeneous regular tree language T the following conditions are equivalent.*

(a) T is definable by a 1-pass preorder typeable EDTD.

(b) T is definable by a restrained competition EDTD.

(c) T is definable by an EDTD with ancestor-sibling-based types.

(d) T is closed under ancestor-sibling-guarded subtree exchange.

(e) T is definable by an ancestor-sibling-based schema.

(f) T can be characterized by ancestor-sibling-based patterns.

Proof. We show (a) ⇔ (c) and (c) ⇒ (d) ⇒ (b) ⇒ (e) ⇒ (f) ⇒ (c). Of these, (c) ⇒ (a) holds by definition, and (c) ⇒ (d), (e)⇒ (f), and (f) ⇒ (c) are straightforward generalizations of the proofs of (b) ⇒ (c), (d) ⇒ (e), and (e) ⇒ (b) in Theorem 8.16.

(b) ⇒ (e): Let T be defined by a restrained competition EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$. We are going to construct a DFA A which determines the type of a node v, after reading its ancestor-sibling-string. From this DFA, we will then obtain an ancestor-sibling-based schema.

For each symbol a^i in Δ, let $A_{a,i} = (Q_{a,i}, \Delta, \delta_{a,i}, s_{a,i}, F_{a,i})$ be a minimal DFA for $L(d(a^i))$. We require that the sets $Q_{a,i}$ are pairwise disjoint. Because it is minimal, each $A_{a,i}$ has at most one state $q^\perp$ from which no accepting state is reachable and it has no unreachable states. From the restrained competition property it immediately follows that, for each state q of $A_{a,i}$, if $\delta(q, b^j) = q_1$, $\delta(q, b^k) = q_2$, $q_1 \neq q_2$ and $j \neq k$ then q_1 or q_2 must be $q^\perp$.

The desired DFA $A = (Q_A, \Sigma, \delta_A, \{s_A\}, F_A)$ is constructed as follows. The set Q_A consists of all pairs (q, b), where $q \in Q_{a,i}$, for some a^i, and $b \in \Delta \cup \{\#\}$. Intuitively,

q is the current state of an automaton $A_{a,i}$ and b is the last type that has been identified. If $s_d = a^\ell$, the initial state s_A of A is $(s_{a,\ell}, \#)$. The transition function δ_A is defined as follows. For each $q \in Q_{a,i}$, $c \in \Delta \cup \{\#\}$ and $b \in \Sigma$ we let $\delta_A((q,c),b) = (\delta_{a,i}(q,b^j), b^j)$, for the unique j with $\delta_{a,i}(q,b^j) \neq q^\perp$, if such a j exists. Otherwise, $\delta_A((q,c),b) = (q^\perp, \#)$. Furthermore, we let $\delta_A((q,b^j),\#) = (s_{b,j}, \#)$. The set F_A can be chosen arbitrarily, as we do not make use of final states.

From the definition, it is obvious that, for each node v of a tree in T,

$$\delta_A^*(s_A, \text{anc-sib-str}(v)) = (q, a^i),$$

for some q, where a^i is the unique type of v.

Now we are ready to define the ancestor-sibling-based schema $S = (\Sigma, R)$. For each state (q, a^i) of A, let L_{q,a^i} denote $\{w \mid \delta_A^*(s_A, w) = (q, a^i), \text{ for some } q\}$. Then R consists of the rules $r_{q,a^i} \to \mu(d(a^i))$, where r_{q,a^i} is a regular expression for L_{q,a^i}. Note, that the languages $L(r_{q,a^i})$ are pairwise disjoint by construction.

It remains to show that S and D describe the same tree language.

To this end, let first $t \in L(D)$ and let v be a node of t. Let (q, a^i) be the state of A after reading anc-sib-str(v). Thus, in the unique labeling of t with respect to D, v has type a^i. Hence, ch-str(v) is in $\mu(d(a^i))$ and $r \to s$ is fulfilled at v.

For the converse direction, let $t \in L(S)$ and let v be a node of t. Let $r \to s$ be the unique rule for which anc-sib-str(v) matches r. By construction, $r \to s$ corresponds to a type a^i for which $\delta_A^*(s_A, \text{anc-sib-str}(v)) = (q, a^i)$. In this way, a unique labeling of t by types is induced and it is straightforward that this labeling is valid with respect to D.

(a) ⇒ (c): We show even a bit more than required: each EDTD with preceding-based types *already has* ancestor-sibling-based types.

Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD which has preceding-based types. Towards a contradiction, we assume that D has types which are *not* ancestor-sibling-based. Clearly, because D has preceding-based types, the types of each $t \in L(D)$ are uniquely determined, thus, only the second requirement of Definition 8.12 can fail. Hence, there are trees $t_1, t_2 \in L(D)$ with nodes v_1 in t_1 and v_2 in t_2 such that anc-sib-str$^{t_1}(v_1) =$ anc-sib-str$^{t_2}(v_2)$ but v_1 has a different label in t_1' than v_2 in t_2', where t_1' and t_2' are the unique witnesses for t_1 and t_2, respectively. We call t_1, t_2, v_1, v_2 a *counterexample.* Let t_1, t_2, v_1, v_2 be a counterexample for which the length of anc-sib-str$^{t_1}(v_1)$ is minimal.

Let U be the set of nodes which are left siblings of ancestors of v_1 let U_2 be the corresponding set for v_2. As anc-sib-str$^{t_1}(v_1) =$ anc-sib-str$^{t_2}(v_2)$, there is a natural bijection f from U_1 to U_2. Clearly, for each $v \in U_1$, v and $f(v)$ have the same label.

Let s be the tree resulting from t_1 by replacing each node $v \in U_1$ and its subtree by $f(v)$ and its subtree. As the counterexample was chosen minimally, for each $v \in U_1$, the label of v in t_1' is the same as the label of $f(v)$ in t_2'. Let s' be the tree resulting from s by labeling each subtree of a node $v \in f(U_1)$ as in t_2' and all other nodes as in t_1'.

It is easy to see that $s' \in L(d)$. As preceding$^s(v_1) =$ preceding$^{t_2}(v_2)$, and as we assume preceding-based types, v_1 must have the same label in s' as v_2 in t_2'. As it also has the same label in t_1' as in s' it follows that the labels in t_1' and t_2' are the same which leads to the desired contradiction.

(d) ⇒ (b): The proof is similar to but a bit more involved than the corresponding proof “(c) ⇒ (a)” in Theorem 8.16.

Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD defining a tree language closed under ancestor-sibling-guarded subtree exchange. We will construct a restrained competition EDTD $E = (\Sigma, \Delta_E, e, s_d, \mu_E)$ such that $L(E) = L(D)$. Again, we assume without loss of generality that D only contains useful types.

For a string $w \in (\Sigma \cup \{\#\})^*$ and $a \in \Sigma$ let types(wa) be the set of all types a^i, for which there is a tree t with witness tree $t' \in L(d)$ and a node v in t such that anc-sib-str$^t(v) = wa$ and the type of v in t' is a^i. For each $a \in \Sigma$, let $\tau(D, a)$ be the set of all nonempty sets types(wa), with $w \in (\Sigma \cup \{\#\})^*$. Again, each $\tau(D, a)$ is finite. The set of types of E is $\Delta_E := \bigcup_{a\in\Sigma} \tau(D, a)$ and, for each $\tau \in \tau(D, a)$, $\mu_E(\tau) = a$.

To define e, let $C \in \Delta_E$ and let $C = \{a^1, \ldots, a^\ell\}$ = types(wa) for a string wa. Then define L_C as the following regular language over Δ_E. It consists of all Δ_E-strings $x = x_1 \cdots x_n$ for which there is an $i \leq \ell$ and a string $x' \in L(d(a^i))$, such that $\mu(x') = \mu_E(x)$ and the j-th position of x is types$(wa\#\mu_E(x_1 \cdots x_j))$. Note that L_C does not depend on the choice of wa.

Intuitively, L_C is the union of all $d(a^i)$ where every jth Σ-symbol in a string $y_1 \cdots y_n$ is assigned the set of types types$(wa\#y_1 \cdots y_j)$. It should be clear that L_C is indeed restrained competition.

We next show that L_C is regular. We define an the NFA M_C accepting L_C of size exponential in the size of D. To this end, let for each type a^i, $A_{a,i} = (\Delta, Q_{a,i}, \delta_{a,i}, s_{a,i}, F_{a,i})$ be an NFA for $L(d(a^i))$. Without loss of generality, we assume that the sets $Q_{a,i}$ are pairwise disjoint and that from every state in $Q_{a,i}$, a final state is reachable.

Define $M_C = (\Delta_E, Q_C, \delta_C, s_C, F_C)$ as follows:

- $Q_C = 2^{Q_{a,1}} \times \cdots \times 2^{Q_{a,\ell}}$;
- $s_C = (\{s_{a,1}\}, \ldots, \{s_{a,\ell}\})$;
- $F_C = \{(P_1, \ldots, P_\ell) \in Q_C \mid \exists i, P_i \cap F_{a,i} \neq \emptyset\}$;
- In order to define $\delta_{a,M}$, let $\overline{P} = (P_1, \ldots, P_\ell)$ be a state of M_C. Then, each P_i contains precisely the states in which each $A_{a,i}$ is after reading the input so far.

  For a state q of $A_{a,i}$ and a Σ-symbol b, let types$_{a,i}(q, b)$ consist of those types b^j for which $\delta_{a,i}(q, b^j) \neq \emptyset$. For a set P of states of $A_{a,i}$, define

  $$\text{types}_{a,i}(P, b) = \bigcup_{q\in P} \text{types}_{a,i}(q, b).$$

  Finally, for $\overline{P}$ as above, define types$_{a,i}(\overline{P}, b) = \bigcup_{j=1}^{\ell}$ types$_{a,i}(P_j, b)$. Notice that, when starting from the state $\overline{P}$, for each $b \in \Sigma$, M_C can only make a transition when reading the Δ_E-symbol types$_{a,i}(\overline{P}, b)$. Therefore, $\delta_C(\overline{P}, \text{types}_{a,i}(\overline{P}, b)) = (P'_1, \ldots, P'_\ell)$ where $P'_i = \bigcup_j \delta_{a,i}(q, b^j)$. For all other $C' \in \Delta_E$ with $C' \neq$ types$_{a,i}(\overline{P}, b)$, set $\delta_C(\overline{P}, C') = \emptyset$.

Note that $L_C = L(M_C)$. Indeed, M_C simulates every $A_{a,i}$ in parallel while computing types$(wa\#y_1 \cdots y_j)$ for every jth symbol in the Σ-string $y_1 \cdots y_n$ from left to right.

Let t be a tree in $L(D)$ witnessed by t'. It is not hard to show by proceeding from the root to the leaves that t' can be transformed to a tree t'' witnessing that $t \in L(E)$. The crucial point is, that the type of each node v in t' is an element of its type in t''.

Thus, it only remains to show $L(E) \subseteq L(D)$. This proof is completely analogous to the corresponding proof in Theorem 8.16. Only the notions depending on ancestors now depend on the corresponding notions for ancestors and their siblings. □

As an immediate consequence, the language we considered in Example 8.11 is not definable by an EDTD admitting 1PPT. Note that the counterexample can be constructed in exactly the same manner.

Theorem 8.19 shows that, in the context of EDTDs, having preceding-based types implies having ancestor-sibling-based types. From the proof it further follows that for each such language a very simple and efficient typing algorithm exists. It is basically a deterministic pushdown automaton with a stack the height of which is bounded by the depth of the document. For each opening tag it pushes one symbol, for each closing tag it pops one. Hence, it only needs a constant number of steps per input symbol. In particular, it works in linear time in the size of the document. It should be noted that such automata have been studied in [SV02] and [KS03] in the context of streaming XML documents. The subclass of the context-free languages accepted by such automata has recently been studied in [AM04]. Thus, just like for single-type EDTDs, there is an efficient one-pass validation and typing algorithm.

Unique Particle Attribution Rule

The most well-known XML Schema constraint is perhaps the Unique Particle Attribution (UPA) rule. In [vdV02], it is mentioned that EDC and UPA are interrelated, in the sense that when a schema satisfies one constraint it almost always also satisfies the other. Although this might be true on most practical examples, in general it is definitely not the case. As we now show, the constraints are incomparable: they are related only in the weak sense that each of them alone implies 1PPT.

An EDTD satisfies the UPA constraint when, for every regular expression r over the type alphabet Δ, the expression $\mu(r)$, obtained from r by replacing every type τ by the element $\mu(\tau)$, is one-unambiguous (see Definition 8.9). The expression $a^1(a^2+b^1)$, for instance, is not EDC but satisfies UPA. For the other counterexample, consider the expression $r = (a^1 + b^1)^* a^1(a^1 + b^1)$ which clearly satisfies EDC. When matching a string against this expression, we always know that we need to type a and b by a^1 and b^1, respectively. However, the expression $\mu(r) = (a + b)^* a(a + b)$ is *not* one-unambiguous. Indeed, $a_1a_2a_3$ and a_2a_3 are both in $L((a_1 + b_1)^* a_2(a_3 + b_2))$. In [BKW98] it is even shown that $\mu(r)$ can not be defined by *any* one-unambiguous regular expression. So, none of the EDC or UPA constraints implies the other.

The definition of UPA and restrained competition regular expressions are related in the following way. When matching a string against a restrained competition regular expression the type of the next element only depends on the part of the string already seen. For a one-unambiguous regular expression over the type alphabet as defined in the previous paragraph, the symbol in the regular expression that matches the next input element only depends on the part of the string already seen. As the matched symbol in the regular expression is actually the type of that symbol, it is immediate that every such one-unambiguous regular expression is restrained competition and, therefore, UPA implies 1PPT.

Example 8.20. Suppose that $r = a^1?b^1(b^1 + c^1)^*a^2c^1$. Then, $\mu(r) = a?b(b+c)^*ac$ and $\overline{\mu(r)} = a_1?b_1(b_2+c_1)^*a_2c_2$. Clearly, $\mu(r)$ is one-unambiguous, which means that when we match, for example, *bbcbac* against $\mu(r)$, the symbol against which the a must be matched (a_2 in $\overline{\mu(r)}$), is uniquely determined without looking ahead. But then, the symbol in r that corresponds to a^2 is also uniquely determined, and this symbol has only one type. So, we also know what type must be assigned to a without looking ahead to c. It is easy to generalize this example to show that any EDTD satisfying UPA is also restrained competition and therefore implies 1PPT. ◇

Although the XML Schema specification allows typing in multiple passes (Section 5.2 in [TBMM04], note on multiple assessment episodes), the previous discussion shows that already the EDC or UPA alone allow for one-pass typing (as they imply 1PPT). Nevertheless, neither EDC nor UPA captures the class of all 1PPT schemas.

There has been quite some debate in the XML community about the restriction to 1-unambiguous regular expressions (see, for example, page 98 of [vdV02] and [Man01, SM03]) as it does not serve its purpose: even for general regular expressions simple validation algorithms exist that are as efficient as those for one-unambiguous regular expressions. One reason to maintain this restriction is to ensure compatibility with SGML parsers, the predecessor of XML. The results of this chapter show that, on the other hand, by using restrained competition EDTDs instead, a larger expressive power can be achieved without (essential) loss in efficiency. For both classes, validation and typing is possible in linear time, allowed schemas can still be recognized in NLOGSPACE and an allowed schema can be constructed in exponential time, if one exists (see [BKW98] and Section 8.5).

On the negative side, both 1-unambiguous expressions and restrained competition expressions lack a comprehensive syntactical counterpart. Whether such an equivalent syntactical restriction exists remains open. It would also be interesting to find syntactic restrictions which imply an efficient construction of an equivalent restrained competition EDTD.

8.4 The Equivalence Theorem for Top-Down Typeable Schemas

We formalize the notion of top-down typeability in terms of pruned-envelope-based types in analogy to the preceding-based types of Definition 8.18. The *pruned envelope* of a node v in t is the tree resulting from t by removing everything below v and below its siblings (see Figure 8.8). In other words, the pruned envelope of v in t is the subtree of t consisting of all nodes that are *not* descendants of v's siblings or of v itself. We denote the pruned envelope of v by p-envelope$^t(v)$.

Definition 8.21. We say that an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ *is top-down typeable* (TDT) or *has pruned-envelope-based types* if there is a function $f : \mathcal{T}_\Sigma \times \text{Nodes} \to \Delta$ such that, for each tree $t \in L(D)$,

- there is exactly one witness t', and
- t' results from t by assigning to each node v the type $f(\text{p-envelope}^t(v), v)$. $\diamond$

We are now ready to prove the following theorem. It is a natural extension of Theorem 8.19 to top-down typeable schemas.

Theorem 8.22. *For a homogeneous regular tree language T the following conditions are equivalent.*

(a) T is definable by a top-down typeable EDTD.

(b) T is definable by a top-down deterministic EDTD.

(c) T is definable by an EDTD with spine-based types.

(d) T is closed under spine-guarded subtree exchange.

(e) T is definable by a spine-based schema.

(f) T can be characterized by spine-based patterns.

Proof. We show (a) ⇔ (c), (e) ⇒ (f) ⇒ (b) ⇒ (e), and (b) ⇒ (c) ⇒ (d) ⇒ (b). Of these, (c) ⇒ (a) holds by definition, and (e) ⇒ (f) and (c) ⇒ (d) are straightforward generalizations of the proofs of (d) ⇒ (e) and (b) ⇒ (c) of Theorem 8.16.

(a) ⇒ (c): We show even a bit more than required: each EDTD with pruned-envelope-based types *already has* spine-based types. The proof is analogous to but slightly different than the proof of (a) ⇒ (c) of Theorem 8.19.

Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD which has pruned-envelope-based types. Towards a contradiction, we assume that D has types which are *not* spine-based. Clearly, because D has pruned-envelope-based types, the types of each $t \in L(D)$ are uniquely determined, thus, only the second requirement of Definition 8.12 can fail. Hence, there are trees $t_1, t_2 \in L(D)$ with nodes v_1 in t_1 and v_2 in t_2 such that $\text{spine}^{t_1}(v_1) = \text{spine}^{t_2}(v_2)$ but v_1 has a different label in t'_1 than v_2 in t'_2, where t'_1 and t'_2 are the unique witnesses for t_1 and t_2, respectively. We call t_1, t_2, v_1, v_2 a *counterexample.* Let t_1, t_2, v_1, v_2 be a counterexample for which the length of $\text{anc-str}^{t_1}(v_1)$ is minimal (that is, the *depth* of v_1 in t_1 is minimal).

Let U_1 be the set of nodes which are left or right siblings of ancestors of v_1 (not of v_1 itself) and let U_2 be the corresponding set for v_2. As $\text{spine}^{t_1}(v_1) = \text{spine}^{t_2}(v_2)$, there is a natural bijection f from U_1 to U_2. Clearly, for each $v \in U_1$, v and $f(v)$ have the same label.

Let s be the tree resulting from t_1 by replacing each node $v \in U_1$ and its subtree by $f(v)$ and its subtree. As the depth of node v_1 was chosen minimally, for each $v \in U_1$, the label of v in t'_1 is the same as the label of $f(v)$ in t'_2. Let s' be the tree resulting from s by labeling each subtree of a node $v \in f(U_1)$ as in t'_2 and all other nodes as in t'_1.

It is easy to see that $s' \in L(d)$, as it can be obtained from t'_1 and t'_2 by label-guarded subtree exchange. As $\text{p-envelope}^s(v_1) = \text{p-envelope}^{t_2}(v_2)$, and as we assume pruned-envelope-based types, v_1 must have the same label in s' as v_2 in t'_2. As v_1 also has the same label in t_1' as in s' it follows that the labels in t_1' and t_2' are the same which leads to the desired contradiction.

(f) ⇒ (b): Let T be characterized by spine-based patterns using the regular language L. Let $A = (\Sigma, Q, \delta, \{s\}, F)$ be a DFA for L.

We define the EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ as follows. We define $\Delta \subseteq (\Sigma \times Q)$ and d inductively as follows. The start symbol s_d of d is in Δ and is defined as the unique (c, q') for which $\delta^*(s, c\$c\#) = \{q'\}$. We set $\mu((c, q')) = c$. Furthermore, when $(a, q) \in \Delta$, we let $d((a, q))$ be a regular expression describing all strings $(b_1, q_1) \cdots (b_n, q_n)$, such that

- A accepts the string $\#b_1 \cdots b_n$ when started from state q; and,
- $\delta^*(q, b_1 \cdots b_i \$ b_i \cdots b_n \#) = \{q_i\}$, for every $i \leq n$.

By construction, D is top-down deterministic. Indeed, for every b_i, there is a unique $\{q_i\}$ with $\delta^*(q, b_1 \cdots b_i \$ b_i \cdots b_n \#) = \{q_i\}$ since A is a DFA. We show that $L(D) = T$. Indeed, when $t \in T$, let t' be obtained from t by relabeling every inner node v labeled a by (a, q) where $q = \delta^*(s, \text{spine}^t(v))$. Then we have that $t' \in L(d)$ and $t = \mu(t')$. Conversely, let $t' \in L(d)$. Then, for every node u of $t = \mu(t')$, $\text{spine}^t(u)\#\text{ch-str}^t(u) \in P_{\text{spine}}(t)$ by construction.

(b) ⇒ (e): Let T be defined by a top-down deterministic EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$. We are going to construct a NFA A which determines the type of a node v, after reading spine(v). From this NFA, we will then obtain a spine-based schema.

For each symbol a^i in Δ, let $L_{a,i}$ be the language $\{wb^j\$b^jv \mid b^j \in \Delta, wbv \in L(d(a^i))\}$. As $L(d(a^i))$ is regular, $L_{a,i}$ is also regular. Let $A_{a,i} = (Q_{a,i}, \Delta \uplus \{\$\}, \delta_{a,i}, s_{a,i}, F_{a,i})$ be a DFA for $L_{a,i}$. We require that the sets $Q_{a,i}$ are pairwise disjoint. From the unambiguously typed property it immediately follows that, for each pair of strings $w_1b^{j_1}\$b^{j_1}v_1, w_2b^{j_2}\$b^{j_2}v_2 \in L_{a,i}$ with $\mu(w_1) = \mu(w_2)$, $\mu(b^{j_1}) = \mu(b^{j_2})$, and $\mu(v_1) = \mu(v_2)$, we have that $j_1 = j_2$.

The desired NFA $A = (Q_A, \Sigma, \delta_A, \{s_A\}, F_A)$ is constructed as follows. Intuitively, it simulates the automata $A_{a,i}$ one after another, while reading Σ-symbols. When it is in an initial state of $A_{a,i}$ and reads a string $wb\$bv$, it guesses types w_1b^j such that $\mu(w_1b^j) = wb$ and it simulates $A_{a,i}$ on w_1b^j. When reading the special character \$, it remembers the last type b^j. It then continues to simulate $A_{a,i}$ while reading v and guessing types v_1 such that $\mu(v_1) = v$. When A does not reach a final state of $A_{a,i}$ after guessing $w_1b^jv_1$, it rejects. According to the unambiguously typeable property of $L(d(a^i))$, there exists at most one way to guess the types such that a final state of $A_{a,i}$ is reached. Finally, when A is in a final state of $A_{a,i}$, has remembered the type b^j, and reads the character #, it continues in $s_{b,j}$ and starts simulating $A_{b,j}$.

Formally, the set Q_A consists of all pairs (q, b_1) and triples (q, b_1, b_2), where $q \in Q_{a,i}$, for some a^i, and $b_1, b_2 \in \Delta \uplus \{\#\}$. Intuitively, q is the current state of an automaton $A_{a,i}$, b_1 is the last type that has been identified, and b_2 is the type that has been identified for the label following the special marker \$. If $s_d = x^l$, the initial state s_A of A is $(s_{x,l}, \#)$. The transition function δ_A is defined as follows. For each $q \in Q_{a,i}$, $c^k \in \Delta$ and $b \in \Sigma$ we let $\delta_A((q, c^k), b) = \{(p, b^j) \mid p \in \delta_{a,i}(q, b^j) \text{ for some } j\}$ and $\delta_A((q, c^k), \$) = \{(p, \$, c^k) \mid p \in \delta_{a,i}(q, \$)\}$. For each $q \in Q_{a,i}$, $c^k, e^\ell \in \Delta$ and $b \in \Sigma$, we let $\delta_A((q, \$, c^k), c) = \{(p, c^k, c^k) \mid p \in \delta_{a,i}(q, c^k)\}$ and $\delta_A((q, e^\ell, c^k), b) = \{(p, b^j, c^k) \mid p \in \delta_{a,i}(q, b^j) \text{ for some } j\}$. Furthermore, we let $\delta_A((q, e^\ell, c^k), \#) = \{(s_{c,k}, \#)\}$ if

$q \in F_{a,i}$ and $\delta_A((q, e^\ell, c^k), \#) = \emptyset$, otherwise. The set F_A can be chosen arbitrarily, as we do not make use of final states.

From the definition of A and the fact that, for each pair of strings $w_1 b^{j_1}\$b^{j_1}v_1$, $w_2 b^{j_2}\$b^{j_2}v_2 \in L_{a,i}$ with $\mu(w_1) = \mu(w_2)$, $\mu(b^{j_1}) = \mu(b^{j_2})$, and $\mu(v_1) = \mu(v_2)$, we have that $j_1 = j_2$, it now follows that, for every string $wb\$bv$, there exists at most one b^{j_1} such that $(s_{b,j_1}, \#) \in \delta_A^*((s_{a,i}, \#), wb\$bv\#)$. From this observation and the above definition of A, we obtain that, for each node v of a tree in T,

$$\delta_A^*(s_A, \text{spine}^t(v)) = \{(s_{a,i}, \#)\},$$

where a^i is the unique type of v.

Now we are ready to define the spine-based schema $S = (\Sigma, R)$. For each state $(s_{a,i}, \#)$ of A, let $L_{s_{a,i}}$ denote $\{w \mid \delta_A^*(s_A, w) = \{(s_{a,i}, \#)\}\}$. Then R consists of the rules $r_{a,i} \to s$, where $r_{a,i}$ is a regular expression defining $L_{s_{a,i}}$ and s is $\mu(d(a^i))$. Notice that the languages $L(r_{a,i})$ are pairwise disjoint by construction.

It remains to show that S and D describe the same tree language.

To this end, let first $t \in L(D)$ and let v be a node of t. Let (q, a^i) be the state of A after reading anc-sib-str(v). Thus, in the unique labeling of t with respect to D, v has type a^i. Hence, ch-str(v) is in $\mu(d(a^i))$ and $r \to s$ is fulfilled at v.

For the converse direction, let $t \in L(S)$ and let v be a node of t. Let $r \to s$ be the unique rule for which anc-sib-str(v) matches r. By construction, $r \to s$ corresponds to a type a^i for which $\delta_A^*(s_A, \text{anc-sib-str}(v)) = (q, a^i)$. In this way, a unique labeling of t by types is induced and it is straightforward that this labeling is valid with respect to D.

(b) ⇒ (c): Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be a top-down deterministic EDTD with $\mu(s_d) = c$. Then define f inductively as follows: $f(\mu(c\$c\#)) = s_d$. Further, for any string $w \cdot w_1 a\$aw_2\#w_3 b\$bw_4\#$ with $w \in (\Sigma \cup \{\#, \$\})^*$, $w_1, w_2, w_3, w_4 \in \Sigma^*$ and $a, b \in \Sigma$, $f(w \cdot w_1 a\$aw_2\#w_3 b\$bw_4\#) = b^j$ where $f(w \cdot w_1 a\$aw_2\#) = a^i$ and there exists a string $w_3' b^j w_4'$ in $d(a^i)$ with $\mu(w_3' b^j w_4') = w_3 b w_4$. As $L(d(a^i))$ is unambiguously typed, f is well-defined and induces a unique typing. Thus, the requirements of Definition 8.12 are satisfied.

(d) ⇒ (b): The proof is similar to but more involved than the corresponding proof "(d) ⇒ (b)" in Theorem 8.19.

Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD defining a tree language closed under spine-guarded subtree exchange. We will construct a top-down deterministic EDTD $E = (\Sigma, \Delta_E, e, s_d, \mu_E)$ such that $L(E) = L(D)$. Again, we assume without loss of generality that D is *reduced*.

For a string $w \in (\Sigma \uplus \{\#, \$\})^*$, $w_1, v_1 \in \Sigma^*$, and $a \in \Sigma$ let types($ww_1 a\$av_1\#$) be the set of all types a^i, for which there is a tree t with witness tree $t' \in L(d)$ and a node v in t such that $\text{spine}^t(v) = ww_1 a\$av_1\#$ and the type of v in t' is a^i. For each $a \in \Sigma$, let $\tau(D, a)$ be the set of all nonempty sets types($ww_1 a\$av_1\#$), with $w \in (\Sigma \uplus \{\#, \$\})^*$ and $w_1, v_1 \in \Sigma^*$. Again, each $\tau(D, a)$ is finite. The set of types of E is $\Delta_E := \bigcup_{a \in \Sigma} \tau(D, a)$ and, for each $\tau \in \tau(D, a)$, $\mu_E(\tau) = a$.

To define e, let $C \in \Delta_E$ and let $C = \{a^1, \ldots, a^\ell\} = \text{types}(ww_1 a\$av_1\#)$ for a string $ww_1 a\$av_1\#$. Then define L_C as the following regular language over Δ_E. It consists of all Δ_E-strings $x = x_1 \cdots x_n$ for which there is an $i \leq \ell$ and a string $x' \in L(d(a^i))$, such that $\mu(x') = \mu_E(x)$ and the j-th position of x is

$$\text{types}(ww_1a\$av_1\#\mu_E(x_1\cdots x_j)\$\mu_E(x_j\cdots x_n)\#).$$

Note that L_C does not depend on the choice of $ww_1a\$av_1$.

Intuitively, L_C is the union of all $d(a^i)$ where every jth Σ-symbol in a string $y_1\cdots y_n$ is assigned the set of types types$(ww_1a\$av_1\#y_1\cdots y_j\$y_j\cdots y_n\#)$. It should be clear that, by definition, L_C is indeed unambiguously typed.

We next show that L_C is regular by showing that there exists a loop-free 2-way alternating finite automaton (2AFA$^{\text{lf}}$) M_C accepting L_C (recall the definition of a 2AFA$^{\text{lf}}$ from Section 3.4). To this end, let, for each type $a^i \in C$, $A_{a,i} = (Q_{a,i}, \Delta, \delta_{a,i}, \{s_{a,i}\}, F_{a,i})$ be a NFA for $L(d(a^i))$. Without loss of generaliy, we assume that the sets $Q_{a,i}$ are pairwise disjoint and that, from every state in $Q_{a,i}$, a final state is reachable.

We now define M_C such that $L(M_C) = \{WBV \mid B = \{b^k \mid w_2b^kv_2 \in \bigcup_{i=1}^{\ell} L(d(a^i))$ and $\mu(w_2) = \mu_E(W), \mu(b^k) = \mu_E(B), \mu(v_2) = \mu_E(V)\}\}$. Note that $L_C = L(M_C)$. Indeed, M_C accepts precisely those strings WBV with $W, V \in \Delta_E^*$ and $B \in \Delta_E$ for which $B = \text{types}(ww_1a\$av_1\#y_1\cdots y_{j-1}b\$by_{j+1}\cdots y_n)$, $\mu_E(W) = y_1\cdots y_{j-1}$ and $\mu_E(V) = y_{j+1}\cdots y_n$.

We now explain the operation of M_C. Intuitively, M_C starts in a universal state and its computation works in three phases: when reading a string $WBV \in \Delta_E^*$ with $\mu(B) = b$,

(i) M_C computes every state p of every automaton $A_{a,i}$ for which there is a string $w_{a,i} \in \Delta$ with $\mu(w_{a,i}) = \mu_E(W)$ and $p \in \delta_{a,i}^*(s_{a,i}, w_{a,i})$;

(ii) M_C nondeterministically determines when it reads the symbol B; and,

(iii) M_C then verifies, that

  (A) for every $b^k \in B$, there exists a state $p \in Q_{a,i}$ computed in step (i) and a string $v_{a,i}$ with $\mu(v_{a,i}) = \mu_E(V)$ and $\delta_{a,i}^*(p, v_{a,i}) \cap F_{a,i} \neq \emptyset$; and,

  (B) for every $b^k \notin B$, for every state $p \in Q_{a,i}$ computed in step (i), and for every string $v_{a,i}$ with $\mu(v_{a,i}) = \mu_E(V)$, we have that $\delta_{a,i}^*(p, v_{a,i}) \cap F_{a,i} = \emptyset$.

Note that, as the initial state of M_C is universal, the test for the symbol B in phase (ii) and (iii) will actually be performed for *every* symbol in the input. By definition of $L(M_C)$, the type B is correct if and only if both tests (A) and (B) are successful. Indeed, a type b^k must be included in B if test (A) is successful and a type b^k must not be included in B if test (B) is successful.

We describe the operation of $M_C = (Q_{M_C}, \Delta_E \uplus \{\triangleright, \triangleleft\}, \delta_{M_C}, I_{M_C}, F_{M_C}, r_{M_C}, U_{M_C})$ more formally. Intuitively, M_C reads its input once from left to right.

- $I_{M_C} = \{(\{s_{a,1}\}, \ldots, \{s_{a,\ell}\})\} \subseteq U_{M_C}$. The computation of M_C starts in a universal state, which is an ℓ-tuple consisting of the start states of $A_{a,1}, \ldots, A_{a,\ell}$.

- $\delta_{M_C}\big((\{s_{a,1}\}, \ldots, \{s_{a,\ell}\}), \triangleright\big) = \big\{\big((\{s_{a,1}\}, \ldots, \{s_{a,\ell}\}), \rightarrow\ \big)\big\}$. When reading the left delimiter symbol of the input string, we simply move to the right.

- For every state $(P_{a,1},\dots,P_{a,\ell}) \subseteq 2^{Q_{a,1}} \times \cdots \times 2^{Q_{a,\ell}}$, we define

$$\begin{aligned}&\delta_{M_C}\big((P_{a,1},\dots,P_{a,\ell}),C'\big) = \\ &\qquad \big\{\big((P'_{a,1},\dots,P'_{a,\ell}),\rightarrow\big) \mid \forall 1 \leq i \leq \ell : P'_{a,i} = \bigcup_{p\in P_{a,i}} \bigcup_{c\in C'} \delta_{a,i}(p,c)\big\} \\ &\qquad\qquad\qquad\qquad \cup \big\{\big((P_{a,1},\dots,P_{a,\ell},\text{check}),-\big)\big\}.\end{aligned}$$

  Here, every state $(P'_{a,1},\dots,P'_{a,\ell})$ and $(P_{a,1},\dots,P_{a,\ell},\text{check})$ is universal. The transitions to states in $2^{Q_{a,1}} \times \cdots \times 2^{Q_{a,\ell}}$ simulate the behavior of all the $A_{a,i}$'s in phase (i). The transition to the state $(P_{a,1},\dots,P_{a,\ell},\text{check})$ is phase (ii) of the operation of M_C.

- For every state $(P_{a,1},\dots,P_{a,\ell}) \subseteq 2^{Q_{a,1}} \times \cdots \times 2^{Q_{a,\ell}} \times \{\text{check}\}$, we define

$$\begin{aligned}\delta_{M_C}\big((P_{a,1},\dots,P_{a,\ell},\text{check}),B\big) = &\bigcup_{b^k\in B} \big\{\big((P_{a,1},\dots,P_{a,\ell},\text{accept},b^k),-\big)\big\} \\ &\cup \bigcup_{b^k\notin B} \big\{\big((P_{a,1},\dots,P_{a,\ell},\text{reject},b^k),-\big)\big\}.\end{aligned}$$

  Here, the state $(P_{a,1},\dots,P_{a,\ell},\text{accept},b^k)$ is existential (we have to check whether a certain string leading to a final state *exists*) and the state $(P_{a,1},\dots,P_{a,\ell},\text{reject}, b^k)$ is a universal state (we have to test that *no string* with certain restrictions leads to a final state).

- For every state $(P_{a,1},\dots,P_{a,\ell},\text{accept},b^k) \in 2^{Q_{a,1}} \times \cdots \times 2^{Q_{a,\ell}} \times \{\text{accept}\} \times \Delta$, we define

$$\begin{aligned}&\delta_{M_C}\big((P_{a,1},\dots,P_{a,\ell},\text{accept},b^k),B\big) = \\ &\qquad\qquad \bigcup_{1\leq i\leq \ell} \bigcup_{p\in P_{a,i}} \big\{\big((p',\text{accept}),\rightarrow\big) \mid p' \in \delta_{a,i}(p,b^k)\big\}.\end{aligned}$$

  Here, every state (p',accept) is existential. This is the start of phase (iii)(A) of the operation of M_C. We continue to simulate the automata $A_{a,i}$, starting from every state p' computed before. The goal is to reach a final state of $A_{a,i}$.

- For every state $(P_{a,1},\dots,P_{a,\ell},\text{reject},b^k) \in 2^{Q_{a,1}} \times \cdots \times 2^{Q_{a,\ell}} \times \{\text{reject}\} \times \Delta$, we define

$$\begin{aligned}&\delta_{M_C}\big((P_{a,1},\dots,P_{a,\ell},\text{reject},b^k),B\big) = \\ &\qquad\qquad \bigcup_{1\leq i\leq \ell} \bigcup_{p\in P_{a,i}} \big\{\big((p',\text{reject}),\rightarrow\big) \mid p' \in \delta_{a,i}(p,b^k)\big\}.\end{aligned}$$

  Here, every state (p',reject) is universal. This is the start of phase (iii)(B) of the operation of M_C. We continue to simulate the automata $A_{a,i}$, starting from every state p' computed before. The goal is to verify that no final state of $A_{a,i}$ can be reached through some $v_{a,i}$ with $\mu(v_{a,i}) = \mu(V)$.

- For every $p \in \bigcup_{1\leq i\leq \ell} Q_{a,i}$, we define

  $$\delta_{M_C}\big((p,\text{accept}),C'\big) = \bigcup_{c\in C'} \{\big((p',\text{accept}),\rightarrow\big) \mid p' \in \delta_{a,i}(p,c), p \in Q_{a,i}\}.$$

  Recall that we assumed that the $Q_{a,i}$ were pairwise disjoint. Here, every state (p',accept) is existential. We are in phase (iii)(A) of the operation of M_C.

- For every $p \in \bigcup_{1\leq i\leq \ell} Q_{a,i}$, we define

  $$\delta_{M_C}\big((p,\text{reject}),C'\big) = \bigcup_{c\in C'} \{\big((p',\text{reject}),\rightarrow\big) \mid p' \in \delta_{a,i}(p,c)\}.$$

  Here, every state (p',accept) is universal. We are in phase (iii)(B) of the operation of M_C.

Finally, for every $i = 1,\ldots,\ell$, M_C goes into a rejecting state whenever it reaches (p,reject) for $p \in F_{a,i}$ or (p,accept) for $p \in Q_{a,i} - F_{a,i}$ on the symbol $\triangleleft$. For all other states, M_C enters an accepting state when reading $\triangleleft$.

It remains to show that $L(D) = L(E)$. To this end, let t be a tree in $L(D)$ witnessed by t'. It is not hard to show by proceeding from the root to the leaves that t' can be transformed to a tree t'' witnessing that $t \in L(E)$. The crucial point is, that the type of each node v in t' is an element of its type in t''.

Thus, it only remains to show $L(E) \subseteq L(D)$. This proof is completely analogous to the corresponding proof in Theorem 8.16. Only the notions depending on ancestors now depend on the corresponding notions for ancestors and their left and right siblings. □

We note that Cristau, Löding, and Thomas [CLT05] have provided a characterization for languages definable by their top-down deterministic tree automata which is equivalent to (f) (on homogeneous regular tree languages). As a consequence, our top-down deterministic EDTDs and their top-down deterministic tree automata are equally expressive on homogeneous languages.

As an immediate consequence of this theorem, the language we considered in Example 8.11 is not definable by an EDTD admitting TDT. Note that the counterexample can be constructed in exactly the same manner.

8.5 Static Analysis and Optimization

In this section, we consider various decision problems that are important for any automated treatment of schemas. In particular, we consider the following problems:

RECOGNITION: Given an EDTD, check whether it is of a restricted type, that is, a DTD, a single-type EDTD or a restrained competition EDTD.

SIMPLIFICATION: Given an EDTD, check whether it has an equivalent EDTD of a restricted type, that is, an equivalent DTD, single-type EDTD or restrained competition EDTD.

Further important problems, such as INCLUSION and MINIMIZATION, are treated in Chapters 9 and 10.

Note the difference between RECOGNITION and SIMPLIFICATION. The former checks whether a given EDTD *is* of a specific form, while the latter checks whether the tree language defined by the given, possibly unrestricted, EDTD can be defined by a constrained EDTD. For instance, the non single-type EDTD $a \to b^1 b^2$, $b^1 \to c$, $b^2 \to c$ is clearly equivalent to the DTD consisting of the rules $a \to bb$ and $b \to c$.

The proofs in this section make use of the tree automata for unranked trees which we defined in Section 2.2.

8.5.1 Recognition of EDTDs

We first consider the RECOGNITION problem. As the definition of a DTD and single-type EDTD is syntactical in nature, it can be immediately verified by an inspection of the rules whether an EDTD is in fact a DTD or a single-type EDTD. The case of restrained competition and top-down deterministic EDTDs is considered in the following Theorem.

Theorem 8.23. *It is decidable in* NLOGSPACE *for an EDTD D whether it is restrained competition or top-down deterministic.*

Proof. We need to check that every regular expression occurring in a rule of D restrains competition, or is unambiguously typed, respectively. We present a nondeterministic logspace algorithm which accepts a regular expression r if it does *not* have the required property. As NLOGSPACE is closed under complement, the theorem follows.

Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be an EDTD. For the restrained competition restriction, the algorithm checks whether, for the automaton of some regular expression there are two states q_1, q_2 which can be reached by two strings $w_1 a^i, w_2 a^j$ for which $\mu(w_1 a^i) = \mu(w_2 a^j)$, and $i \neq j$, and from which an accepting path exists. For the top-down deterministic restriction, the algorithm checks whether an accepting path exists from q_1, respectively, q_2 by reading strings v_1, respectively, v_2 for which $\mu(v_1) = \mu(v_2)$.

Let $N_r = (\Delta, Q, \delta, q_0, F)$ be an NFA equivalent to r. We explain the algorithm in detail for the top-down deterministic restriction. The algorithm for the restrained competition property is analogous. The algorithm works as follows.

1. it first guesses two states (q_1, q_2) of N_r;
2. it verifies that there are strings $w_1 a^i, w_2 a^j$ with $\mu(w_1 a^i) = \mu(w_2 a^j)$ and $i \neq j$ such that $q_1 \in \delta^*(q_0, w_1 a^i)$ and $q_2 \in \delta^*(q_0, w_2 a^j)$;
3. it verifies that there are strings v_1, v_2 with $\mu(v_1) = \mu(v_2)$ such that $\delta^*(q_1, v_1) \cap F \neq \emptyset$ and $\delta^*(q_2, v_2) \cap F \neq \emptyset$;
4. it accepts if all these verifications work out.

Furthermore, all steps can be done in logarithmic space, as the NFA A does not have to be computed in advance and the verifications in the second and third step can be done by remembering a pair of states of A, guessing the strings one symbol at a time and testing, for each pair of guessed symbols x, y, whether $\mu(x) = \mu(y)$. □

8.5.2 Simplification of EDTDs

Next, we study the complexity of the SIMPLIFICATION problem for the target schema types DTD, single-type EDTD, restrained competition EDTD, and top-down deterministic EDTD, respectively. Unfortunately, this test is hard for EXPTIME. For the former three schema types, the test is also in EXPTIME and our algorithm also constructs a corresponding equivalent simpler schema when it exists. We have to leave the precise complexity upper bound for top-down deterministic EDTDs open.

Theorem 8.24.

1. *Each of deciding whether an EDTD has an equivalent DTD, single-type EDTD, restrained competition EDTD, or top-down deterministic EDTD is* EXPTIME-*hard.*

2. *Each of deciding whether an EDTD has an equivalent DTD, single-type EDTD, or restrained competition EDTD is in* EXPTIME.

Proof. We start with the lower bounds. In all four cases, the lower bound is obtained by a reduction from the universality problem for non-deterministic tree automata (Proposition 3.9). Let NTA(RE) denote the class of NTAs where the regular languages encoding the transition function are represented by regular expressions. The hardness result even holds for NTA(RE) where automata only have one final state and where all accepted trees have the same root symbol (say a).

Therefore, let $A = (Q, \Sigma, \delta, F)$ be an NTA(RE) over alphabet $\Sigma = \{a, b\}$ with one final state $F = \{q_F\}$. We can assume without loss of generality that A accepts at least one tree. We can construct in LOGSPACE an equivalent EDTD $D = (\Sigma, \Delta, d, a^{q_F}, \mu)$ as follows: $\Delta = \{b^q \mid b \in \Sigma, q \in Q\}$, $\mu(b^q) = b$ for every $b \in \Sigma$, and d consists of the rules $d(b^q) = r_{b,q}$ where $r_{b,q}$ is the regular expression obtained from $\delta(b, q)$ by replacing every occurrence of a state p by $(a^p + b^p)$. As every $t \in L(d)$ induces an accepting run of A on $\mu(t)$, it is immediate that A and D are equivalent.

From D, we now construct an EDTD D' such that

(i) if $L(A) = \mathcal{T}_\Sigma$ then $L(D')$ is defined by a DTD; and,

(ii) if $L(A) \neq \mathcal{T}_\Sigma$ then $L(D')$ is not defined by a restrained competition EDTD.

Of course (i) and (ii) together imply the statement of the theorem.

Let $D_{\text{all}} = (\Sigma, \{a^1, b^1\}, d_{\text{all}}, a^1, \mu_{\text{all}})$ be the EDTD with rules $a^1 \to (a^1 + b^1)^*$, $b^1 \to (a^1 + b^1)^*$, $\mu_{\text{all}}(a^1) = a$, and $\mu_{\text{all}}(b^1) = b$. Hence, $L(D_{\text{all}}) = \mathcal{T}_\Sigma$.

Intuitively, D' accepts all trees of the form $r(t_1 \cdots t_n)$ such that, there exists a j for which $t_j \in L(D)$ and, for all $i \neq j, 1 \leq i \leq n$, $t_i \in L(D_{\text{all}})$. Formally, we define $D' = (\Sigma \uplus \{r\}, \Delta \uplus \{a^1, b^1, r\}, d', \mu')$, where μ' is defined in the obvious manner and d' is defined as follows:

- $d'(r) = (a^1)^* a^{q_F} (a^1)^*$;

- for every $c \in \{a^1, b^1\}$, $d'(c) = d_{\text{all}}(c)$; and,

- for every $c \in \Delta$, $d'(c) = d(c)$.

We show (i) and (ii):

(i): First note that when $L(A) = \mathcal{T}_\Sigma$, then $L(D) = L(D_{\text{all}})$. Hence, $L(D') = \{r(t_1 \cdots t_n) \mid t_1, \ldots, t_n \in \mathcal{T}_\Sigma\}$. The latter can clearly be defined by a DTD.

(ii): Let $L(A) \neq \mathcal{T}_\Sigma$ and let t_{in} and t_{out} be two trees such that $t_{\text{in}} \in L(A)$ and $t_{\text{out}} \notin L(A)$. Towards a contradiction, assume that $L(D')$ is definable by a top-down typeable EDTD. Hence, $L(D')$ is closed under spine-guarded subtree exchange (Theorem 8.22). Let $t_{\text{left}} := r(t_{\text{in}}t_{\text{out}})$ and let $t_{\text{right}} := r(t_{\text{out}}t_{\text{in}})$. By definition of D', we have that $t_{\text{left}}, t_{\text{right}} \in L(D')$. Let t be the tree $t_{\text{left}}[1 \leftarrow t_{\text{out}}] = r(t_{\text{out}}t_{\text{out}})$. Notice that, as $r\$r\#a\$aa\# = \text{spine}^{t_{\text{left}}}(1) = \text{spine}^{t_{\text{right}}}(1)$, t can be obtained from t_{left} and t_{right} by spine-guarded subtree exchange. However, t is not in $L(D')$, which is a contradiction. Hence, (ii) follows.

The exponential time upper bounds for the single-type and restrained competition cases can be obtained by performing the constructions in the proofs (c) ⇒ (a) and (d) ⇒ (b) in Theorems 8.16 and 8.19, respectively. Both the construction of the EDTD and checking equivalence with the original one can be done in exponential time. For DTDs a similar construction is in polynomial time, but the equivalence check still needs exponential time.

- In the case of single-type EDTDs we proceed as follows. Let $D = (\Sigma, \Delta_D, d, s_d, \mu_D)$ be a given EDTD. We assume D is reduced. We first construct the EDTD(NFA) $E = (\Sigma, \Delta_E, e, s_d, \mu_E)$ as described in the proof of Theorem 8.16 (c) ⇒ (a). We argue that this can be done in exponential time. First, we need to compute $\Delta_E \subseteq 2^{\Delta_D}$. To this end, we enumerate all sets types(w). Let $s_d = c^0$. Initially, set Anc-strings $:= \{c\}$, ATypes$(c) := \{c^0\}$ and $R := \{\{c^0\}\}$.

  Repeat the following until Anc-strings becomes empty:

  1. Remove a string wa from Anc-strings.
  2. For every $b \in \Sigma$, let ATypes(wab) contain all b^i for which there exists an a^j in ATypes(wa) and a string in $d(a^j)$ containing b^i. If ATypes(wab) is not empty and not already in R, then add it to R and add wab to Anc-strings.

  Since we add every set only once to R, the algorithm runs in time exponential in the size of D. Moreover, we have that ATypes$(w) =$ types(w) for every w, and that $R = \Delta_E$. Now we know Δ_E, the rules of e can be directly computed.

  It follows from the proof of Theorem 8.16 (c) ⇒ (a) that D is equivalent to a single-type EDTD if and only if D is in fact equivalent to E. Further, E then is the corresponding single-type EDTD. The construction of E can be done in exponential time and E might be of exponential size in D. Then it has to be checked whether D and E are equivalent. Fortunately, as always $L(D) \subseteq L(E)$, we only have to check whether $L(E) - L(D)$ is empty. This involves the complementation of the tree automaton for D, resulting in a tree automaton of possibly exponential size, and in the test whether the automata for $L(E)$ and the complement of $L(D)$ have a non-empty intersection. The latter is polynomial in the size of the automata. Hence, we altogether get an exponential time algorithm.

- Testing whether an EDTD has an equivalent restrained competition EDTD can be done along the same lines, this time based on the proof of Theorem 8.19 (d) $\Rightarrow$ (b). To compute types(w) for ancestor-sibling-strings w, we just need to let b in step (2) above range over $\Sigma \cup (\{\#\} \cdot \Sigma)$. A type b^ℓ is then added to ATypes(wb) if w is of the form $w'a\#x_1 \cdots x_k b$ where $x_1 \cdots x_k$ does not contain a separator # and

  1. there is an a^i in ATypes($w'a$) and
  2. there are $x^{i_j} \in$ ATypes($w'a\#x_1 \cdots x_j$),
  3. such that, $x_1^{i_1} \cdots x_k^{i_k} b^\ell$ is a prefix of a string in $d(a^i)$.

- Finally, we describe how it can be tested whether a given EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ has an equivalent DTD. As usual, we can assume that D is reduced. Let, for each $a^i \in \Delta$, $r_{a,i}$ be the regular expression obtained from $d(a^i)$ by replacing every symbol b^j by b. We define a DTD (Σ, d_1, s_d) simply by taking the rules $a \to \bigcup_i r_{a,i}$, for every $a \in \Sigma$. It remains to show that D has an equivalent DTD if and only if $L(D) = L(d_1)$.

  Analogously as in Theorem 8.16((c)$\Rightarrow$(a)), we have that $L(D) \subseteq L(d_1)$. Towards a contradiction, suppose that D has an equivalent DTD and that $t \in L(d_1) - L(D)$. According to Lemma 2.10 in [PV00] (see Section 8.1.2), $L(D)$ is closed under label-guarded subtree exchange. As $t \notin L(D)$ there exists a node u in t such that subtree$^t(u) \notin L((D, a^i))$ for any $a^i \in \Delta$, but for every child $u_1, \ldots, u_n$ of u, we have that subtree$^t(u_j) \in L((D, b_j^{i_j}))$ for some $b_j^{i_j} \in \Delta$. Note that u and u_j are labeled with a and b, respectively. First, we note that u can never be a leaf node. Indeed, if there is no $a^i \in \Delta$ such that $\varepsilon \in L(r_{a,i})$, then ε is also not in $\bigcup_i L(r_{a,i})$, which is the content model of a in d_1.

  If u is not a leaf node, we can do the following. By definition of d_1, for every $b_j^{i_j}$, there exists an a^k such that $b_j^{i_j}$ occurs in $d(a^k)$. Thus, as D is reduced, for every u_j there exists a tree $t_j \in L(D)$ with a $v \in$ Nodes(t) such that lab$^{t_j}(v) = b_j$, the parent of v is labeled a, and subtree$^{t_j}(v) =$ subtree$^t(u)$. But this means that t can be constructed from $t_1, \ldots, t_n$ by label-guarded subtree exchange, which is a contradiction as $t \notin L(D)$. □

8.6 Subtree-Based Schemas

From what was presented so far an obvious question arises. What happens if we soften the requirement that the type of an element has to be determined when its *opening* tag is visited? What if instead it has to be computed when the *closing* tag is seen? It turns out that every regular tree language has an EDTD which allows such 1-pass *postorder* typing. Furthermore, the EDTDs used for this purpose can be defined as straightforward extensions of restrained competition EDTDs.

Definition 8.25. An EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ is *extended restrained competition* if, for every regular expression r occurring in a rule the following holds: whenever there are two strings wa^iv and wa^jv' in $L(r)$ with $a^i \neq a^j$ and $\mu(a^i) = \mu(a^j)$, then $L((D, a^i)) \cap L((D, a^j))$ is empty. ◇

For a tree t and a node v, the *preceding-subtree* of v in t is the tree resulting from t by removing all right siblings of v and its ancestors together with the respective subtrees (see Figure 8.4). We denote the preceding-subtree of v by preceding-subtree$^t(v)$. The notion of preceding-subtree is illustrated in Figure 8.8.

Definition 8.26. We say that an EDTD $D = (\Sigma, \Sigma', d, \mu)$ *has preceding-subtree-based types* if there is a function f which maps tree-node pairs to Σ' such that, for each tree $t \in L(D)$,

- t has exactly one witness t', and
- t' results from t by assigning to each node v the type $f(\text{preceding-subtree}^t(v), v)$.

◇

Stated in terms of XML documents, the type of an element depends on the prefix of the document which ends with the closing tag of the element.

The following result shows that all regular tree languages admit 1-pass postorder typing.

Theorem 8.27. *For a homogeneous tree language T the following are equivalent:*

(a) T is definable by an extended restrained competition EDTD;

(b) T is definable by an EDTD with preceding-subtree-based types; and,

(c) T is regular.

Proof. The directions (a) ⇒ (c) and (b) ⇒ (c) are trivial. The proof of the opposite directions uses the fact that regular languages can be validated by deterministic bottom-up automata.

(c) ⇒ (a) and (c) ⇒ (b): Let T be the tree language defined by a bottom-up deterministic tree automaton $B = (Q, \Sigma, \delta, F)$. We can assume that transition functions are represented by regular expressions. We construct an EDTD $D = (\Sigma, \Delta, d, s_d, \mu)$ such that $L(D) = L(B)$ exactly as in the proof of Theorem 8.24. In particular, $\Delta = \{a^q \mid a \in \Sigma, q \in Q\}$. It is immediate that a tree $t \in L(D, a^q)$ if and only if $\delta^*(t) = q$, where $\text{lab}^t(v) = a$ for the root v of t. Here, δ^* is the canonical extension of δ to trees. As B is deterministic, $L((D, a^q)) \cap L((D, a^{q'})) = \emptyset$ for all $a \in \Sigma$ and $q \neq q' \in Q$. Hence, D is extended restrained competition. By observing that there is only one accepting run for every tree and defining $f(\text{preceding-subtree}^t(u), u) = \delta^*(\text{subtree}^t(u))$, it follows that D has preceding-subtree-based types. □

In the EDTD used in the proof the type of each element actually only depends on its subtree. This should be compared with the previous characterizations where the type depended on the upper context.

Remark 8.28. Although there is an extended restrained competition for every regular tree language, not every EDTD itself is extended restrained competition. The EDTD D defined by the rules

$$r \to (a^1 + a^2) \qquad a^1 \to b + c + \varepsilon \qquad a^2 \to c + d + \varepsilon,$$

is *not* extended restrained competition, as $\{\varepsilon, c\} \subseteq L((D, a^1)) \cap L((D, a^2))$. $\diamond$

We conclude by noting that extended restrained competition is a tractable notion.

Theorem 8.29. *It is decidable in* PTIME *for an EDTD D whether it is extended restrained competition.*

Proof. Let $D = (\Sigma, \Sigma', d, s_d, \mu)$ be an EDTD. Let E be the set $\{(a^i, a^j) \mid L((D, a^i)) \cap L((D, a^j)) \neq \emptyset\}$. This set can be computed in polynomial time by checking whether the non-deterministic tree automata for $L((D, a^i))$ and $L((D, a^j))$ have a non-empty intersection (Proposition 3.18).

It suffices to show that the following is in PTIME: testing whether, for a single regular expression r, there are two strings wa^iv and wa^jv' in $L(r)$ with $a^i \neq a^j$, $\mu(a^i) = \mu(a^j)$ and $L((D, a^i)) \cap L((D, a^j))$ is empty. Let $N_r = (\Delta, Q, \delta, q_0, F)$ be an NFA equivalent to r.

The algorithm makes use of two sets:

- the set of reachable states $R := \{q \in Q \mid \exists w \in \Delta^*, \delta^*(q, w) \in F\}$; and,
- the set of pairs of states that can be reached by the same string, $S := \{(q_1, q_2) \in Q \times Q \mid \exists w \in \Delta^*, \{q_1, q_2\} \subseteq \delta^*(q_0, w)\}$.

Note that R and S can be computed in linear and quadratic time, respectively, by the usual reachability algorithm. Then, r is extended restrained competition if and only if there are no $q_1, q_2 \in S$ and a, i, j with $i \neq j$, $\delta(q_1, a^i) \cap R \neq \emptyset$, $\delta(q_2, a^j) \cap R \neq \emptyset$, and $(a^i, a^j) \in E$. The latter test is in PTIME. $\square$

9

XML Schemas and Chain Regular Expressions

The presence of a schema accompanying an XML document has many advantages: it allows for automatic validation, and optimization of translation, storage and processing of XML data. Furthermore, for typechecking or type inference of XML transformations [HP03, MN05a, MSV03, PV00], schema information is even crucial. The following standard optimization problems for schemas are among the basic building blocks for many of the algorithms for the above mentioned problems:

- INCLUSION: Given two schemas D and D', is every XML document in D also defined by D'?
- EQUIVALENCE: Given two schemas D and D', do D and D' define the same set of XML documents?
- INTERSECTION NON-EMPTINESS: Given schemas $D_1, \ldots, D_n$, do they define a common XML document?

It is therefore important to establish the exact complexity of these problems.

The purpose of the present chapter is to investigate the complexity of the above mentioned problems for extended context free grammars, single-type EDTDs, restrained competition EDTDs, and top-down deterministic EDTDs. As argued in Chapter 8, the latter are abstractions of DTDs, XML schema with the EDC-constraint, 1-pass preorder typeable schemas, and top-down typeable schemas, respectively. As EDTDs have a close correspondence to unranked tree automata, and as grammars and tree automata have already been studied in depth for many decades, it is not surprising that the complexity of the above mentioned decision problems is already known. Indeed, in the case of DTDs, the problems reduce to their counterparts for

Factor	Abbr.
a	a
a^*	a^*
a^+	a^+
$a?$	$a?$
w^*	w^*
w^+	w^+
$w?$	$w?$

Factor	Abbr.
$(a_1 + \cdots + a_n)$	$(+a)$
$(a_1 + \cdots + a_n)^*$	$(+a)^*$
$(a_1 + \cdots + a_n)^+$	$(+a)^+$
$(a_1 + \cdots + a_n)?$	$(+a)?$
$(a_1^* + \cdots + a_n^*)$	$(+a^*)$
$(a_1^+ + \cdots + a_n^+)$	$(+a^+)$

Factor	Abbr.
$(w_1 + \cdots + w_n)$	$(+w)$
$(w_1 + \cdots + w_n)^*$	$(+w)^*$
$(w_1 + \cdots + w_n)^+$	$(+w)^+$
$(w_1 + \cdots + w_n)?$	$(+w)?$
$(w_1^* + \cdots + w_n^*)$	$(+w^*)$
$(w_1^+ + \cdots + w_n^+)$	$(+w^+)$

Table 9.1: Possible factors in chain regular expressions and how they are denoted ($a, a_i \in \Sigma$, $w, w_i \in \Sigma^+$).

regular expressions: all three problems are PSPACE-complete (Proposition 3.9). For tree automata, they are well-known to be EXPTIME-complete (Proposition 3.9).

Unfortunately, these complexity results result do not tell us much about the hardness of optimization of actual XML schemas:

1. As we explained in Chapter 8, XML Schema Definitions are not powerful enough to express the entire class of regular unranked tree languages, because of the Element Declarations Consistent rule (which is abstracted by the single-type restruction on EDTDs). Even the more expressive restrained competition and top-down deterministic EDTDs still do not have the expressive power of tree automata over unranked trees. Hence, the mentioned decision problems can be easier on these restricted EDTDs than on unranked tree automata in general.

2. Practical DTDs and XML Schema Definitions are usually much more simpler than the regular expressions and tree automata needed for the classical PSPACE- and EXPTIME-hardness proofs. Actually, a study by Bex, Neven, and Van den Bussche [BNV04] confirms that more than ninety percent of the regular expressions occurring in practical DTDs and XSDs are CHAin Regular Expressions (CHAREs), that is, expressions $e_1 \cdots e_n$, where every e_i is a factor of the form $(w_1 + \cdots + w_m)$ — possibly extended with Kleene-star, plus or question mark — and each w_i is a string. We define this class of regular expressions more precisely in Section 9.1.1.

In the present chapter, we therefore revisit the complexity of INCLUSION, EQUIVALENCE, and INTERSECTION NON-EMPTINESS for DTDs and restricted EDTDs with CHAREs. Clearly, complexity lower bounds for INCLUSION, EQUIVALENCE, or INTERSECTION NON-EMPTINESS for a class of regular expressions $\mathcal{R}$ imply lower bounds for the corresponding decision problems for DTDs, single-type EDTDs, restrained competition EDTDs and top-down deterministic EDTDs with right-hand sides in $\mathcal{R}$. Interestingly, we show that, for INCLUSION and EQUIVALENCE, the complexity upper bounds for the string case *also* carry over to DTDs and single-type EDTDs. For restrained competition and top-down deterministic EDTDs, we show that the complexity upper bounds carry over from $\mu(\mathcal{R})$-expressions to EDTDs with right hand sides in $\mathcal{R}$. For intersection, the latter still holds for DTDs, but not for single-type, restrained competition, or top-down deterministic EDTDs. So, in many cases, it suffices to restrict attention to the complexity of CHAREs to derive complexity bounds for XML schema languages.

Before we give an overview of our complexity results regarding CHAREs, we briefly discuss the determinism constraint: the XML specification requires DTD content models to be *deterministic* because of compatibility with SGML (see Section 8.1.2 and also Section 3.2.1 of [BPSM$^+$04]). In XML Schema, this determinism constraint is referred to as the *Unique Particle Attribution* constraint (see Section 8.3 and also Section 3.8.6 of [SMT05]). Brüggemann-Klein and Wood [BKW98] formalized the regular expressions adhering to this constraint as the *one-unambiguous* regular expressions. As such expressions can be translated in polynomial time to an equivalent deterministic finite state machine, it immediately follows that INCLUSION and EQUIVALENCE for such regular expressions, and hence also for practical DTDs, are in PTIME. In contrast, we show in Theorem 9.23 that INTERSECTION NON-EMPTINESS of one-unambiguous regular expressions remains PSPACE-hard (even when every symbol can occur at most three times). Nevertheless, we think it is important to also study the complexity of CHAREs without the determinism constraint as there has been quite some debate in the XML community about the restriction to one-unambiguous regular expressions, as we noted in Section 8.3. Another reason to study CHAREs without the determinism constraint is that they are included in navigational queries expressed by caterpillar expressions [BKW00], $\mathcal{X}_{\text{reg}}^{\text{CPath}}$ and $\mathcal{X}_{\text{reg}}$ [Mar04], or regular path queries [CGGLV03]. Hence, lower bounds for optimization problems for CHAREs imply lower bound for optimization problems for navigational queries. Hence, it is relevant to study the broader class of possibly non-deterministic but simple and practical regular expressions.

Our results on the complexity of CHAREs are summarized in Table 9.2. We denote by RE($\mathcal{S}$) the set of all CHAREs. Recall that the three decision problems are PSPACE-complete for the class of all regular expressions (Proposition 3.9). We briefly discuss our results:

- We show that INCLUSION is already CONP-complete for several, seemingly very innocent expressions: when every factor is of the form *(i)* a or a^*, *(ii)* a or $a?$, *(iii)* a or $(a_1^+ + \cdots + a_n^+)$, *(iv)* a or w^+ and *(v)* a^+ or $(a_1 + \cdots + a_n)$ with $a, a_1, \ldots, a_n$ arbitrary alphabet symbols and w an arbitrary string with at least one symbol. Even worse, when factors of the form $(a_1 + \cdots + a_n)^*$ or $(a_1 + \cdots + a_n)^+$ are also allowed, we already obtain the maximum complexity: PSPACE. When such factors are disallowed the complexity remains CONP. The inclusion problem is in PTIME when we allow (general) regular expressions where the number of occurrences of the same symbol is bounded by some constant k (a fragment we denote with RE$^{\leq k}$). As the running time is n^k, k should of course be small to be feasible. Fortunately, this seems to be the case quite often. Recent investigation has pointed out that in practice, for ninety-nine percent of the regular expressions occurring in DTDs or XML Schema Definitions, k is equal to one [BNST05].

- The precise complexity of EQUIVALENCE largely remains open. Of course, it is never harder than inclusion, but we conjecture that it is tractable for a large fragment of RE($\mathcal{S}$). We only prove a PTIME upper bound for expressions where each factor is a or a^*, or a or $a?$. Even for these restricted fragments the proof is non-trivial. Basically, we show that two expressions are equivalent if and only if they have the same *sequence normal form*, modulo one rewrite rule. Interestingly, the sequence normal form specifies factors much in the same way as XML Schema does. For every symbol, an explicit upper and lower bound is specified. For instance, $aa^*bbc?c?$ becomes $a[1,*]b[2,2]c[0,2]$.

- INTERSECTION NON-EMPTINESS is coNP-complete when each factor is either of the form *(i)* a or a^*, *(ii)* a or $a?$, *(iii)* a or $(a_1^+ + \cdots + a_n^+)$, *(iv)* a or $(a_1 + \cdots + a_n)^+$ or of the form *(v)* a^+ or $(a_1 + \cdots + a_n)$. As we can see, the complexity of INTERSECTION NON-EMPTINESS is not always the same as for INCLUSION. There are even cases where inclusion is harder and others where intersection is harder. In case *(iv)*, for example, INCLUSION is PSPACE-complete, whereas INTERSECTION NON-EMPTINESS problem is coNP-complete. Indeed, INTERSECTION NON-EMPTINESS remains in coNP even if we allow all kinds of factors except $(w_1 + \cdots + w_n)^*$ or $(w_1 + \cdots + w_n)^+$. On the other hand, INTERSECTION NON-EMPTINESS is PSPACE-hard for RE$^{\leq 3}$ and for deterministic (or *one-unambiguous*) regular expressions [BKW98], whereas their inclusion problem is in PTIME. The only tractable fragment we obtain is when each factor is restricted to a or a^+, which is the class of RE$^+$-expressions that we used in Chapter 6.

RE-fragment	Inclusion	Equivalence	Intersection
a, a^+	in PTIME (DFA!)	in PTIME	in PTIME (9.25)
a, a^*	coNP (9.10)	in PTIME (9.17)	NP (9.18)
$a, a?$	coNP (9.10)	in PTIME (9.17)	NP (9.18)
$a, (+a^+)$	coNP (9.10)	in coNP	NP (9.18)
$a^+, (+a)$	coNP (9.10)	in coNP	NP (9.18)
a, w^+	coNP (9.10)	in coNP	in NP (9.18)
$\mathcal{S} - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\}$	coNP (9.10)	in coNP	NP (9.18)
$a, (+a)^*$	PSPACE (9.10)	in PSPACE	NP (9.18)
$a, (+a)^+$	PSPACE (9.10)	in PSPACE	NP (9.18)
$\mathcal{S} - \{(+w)^*, (+w)^+\}$	PSPACE (9.10)	in PSPACE	NP (9.18)
$a, (+w)^*$	PSPACE (9.10)	in PSPACE	PSPACE ([Bal02])
$a, (+w)^+$	PSPACE (9.10)	in PSPACE	PSPACE ([Bal02])
$\mathcal{S}$	PSPACE (9.10)	in PSPACE	PSPACE ([Bal02])
RE$^{\leq k}$ ($k \geq 3$)	in PTIME (9.11)	in PTIME	PSPACE (9.23)
one-unambiguous	in PTIME	in PTIME	PSPACE (9.23)

Table 9.2: Summary of our results. Unless specified otherwise, all complexities are completeness results. The theorem numbers are given in brackets.

Related Work. The complexities of EQUIVALENCE, INCLUSION and INTERSECTION NON-EMPTINESS for general regular expressions and several fragments were studied in [HRS76, Koz77, SM73]. From these, the most related result is the CONP-completeness of EQUIVALENCE and INCLUSION of bounded languages [HRS76]. A language L is *bounded* if there are strings $v_1, \ldots, v_n$ such that $L \subseteq v_1^* \cdots v_n^*$. It should be noted that the latter class is much more general than, for instance, our class RE(w^*). More recently, INCLUSION for two fragments of chain regular expressions have been shown to be tractable: INCLUSION for RE($a?, (+a)^*$) [ABJ98] and RE(a, Σ, Σ^*) [MS99a, MS04]. This last result should be contrasted with the PSPACE-completeness of INCLUSION for RE($a, (+a), (+a)^*$), or even RE($a, (+a)^*$). Further, Bala investigated INTERSECTION NON-EMPTINESS for regular expressions of limited star height [Bal02]. He showed that it is PSPACE-complete to decide whether INTERSECTION NON-EMPTINESS for RE($(+w)^*$) expressions contains a non-empty string. Bala's proof can easily be adjusted to obtain PSPACE-hardness of INTERSECTION NON-EMPTINESS for RE($a, (+w)^*$) or RE($a, (+w)^+$) expressions.

9.1 Preliminaries

9.1.1 Chain Regular Expressions

In many of our proofs, we make use of how a string can be matched against a regular expression. We formalize this by the notion of a *match*. A *match* m between a string $w = a_1 \cdots a_n$ and a regular expression r is a (partial) mapping from pairs (i, j), $1 \leq i \leq j + 1 \leq n$, of positions of w to sets of subexpressions of r. This mapping is consistent with the semantics of regular expressions, that is,

(1) if $\varepsilon \in m(i, j)$, then $i = j + 1$;

(2) if $a \in m(i, j)$, for $a \in \Sigma$, then $i = j$ and $a_i = a$;

(3) if $(r_1 + r_2) \in m(i, j)$, then $r_1 \in m(i, j)$ or $r_1 \in m(i, j)$;

(4) if $r_1 r_2 \in m(i, j)$, then there is a k such that $r_1 \in m(i, k)$ and $r_2 \in m(k + 1, j)$;

(5) if $r^* \in m(i, j)$, then there are $k_1, \ldots, k_t$ such that $r \in m(i, k_1)$, $r \in m(k_t + 1, j)$ and $r \in m(k_\ell + 1, k_{\ell+1})$, for all ℓ, $1 \leq \ell < t$.

Furthermore, m is minimal with these properties. That is, if m' fulfills (1)–(5) and $m'(i, j) \subseteq m(i, j)$ for each i, j, then $m' = m$. We say that m *matches* a substring $a_i \cdots a_j$ of w onto a subexpression r' of r when $r' \in m(i, j)$. Often, we leave m implicit whenever this cannot give rise to confusion. We then say that $a_i \cdots a_j$ *matches* r'.

We consider simple regular expressions occurring in practice in DTDs and XML Schemas [Cho02], which we call *CHAin Regular Expessions (CHAREs)*. These regular expressions are defined as follows.

Definition 9.1. A *base symbol* is a regular expression s, s^*, s^+, or $s?$, where s is a non-empty string; a *factor* is of the form e, e^*, e^+, or $e?$ where e is a disjunction of base symbols of the same kind. That is, e is of the form $(s_1 + \cdots + s_n)$, $(s_1^* + \cdots + s_n^*)$, $(s_1^+ + \cdots + s_n^+)$, or $(s_1? + \cdots + s_n?)$, where $n \geq 0$ and $s_1, \ldots, s_n$ are non-empty strings. A *chain regular expression (CHARE)* is $\emptyset$, ε, or a sequence of factors. ◇

The regular expressions $((abc)^* + b^*)(a + b)?(ab)^+(ac + b)^*$ is a chain regular expression. The expression $(a + b) + (a^*b^*)$, however, is not.

We introduce a uniform syntax to denote subclasses of chain regular expressions by specifying the allowed factors. We distinguish whether the string s of a base symbol consists of a single symbol (denoted by a) or a string (denoted by w) and whether it is extended by $*$, $+$, or ?. Furthermore, we distinguish between factors with one disjunct or with arbitrarily many disjuncts: the latter is denoted by $(+\cdots)$. Finally, factors can again be extended by $*$, $+$, or ?. A list of possible factors, together with their abbreviated notation, is displayed in Table 9.1. This table only shows factors which give rise to chain regular expressions with different expressive power.

We denote subclasses of chain regular expressions by RE(X), where X is a list of the allowed factors. For example, we write RE($(+a)^*, w?$) for the set of regular expressions $e_1 \cdots e_n$ where every e_i is either *(i)* $(a_1 + \cdots + a_m)^*$ for some $a_1, \ldots, a_m \in \Sigma$ and $m \geq 1$, or *(ii)* $w?$ for some $w \in \Sigma^+$.

If $A = \{a_1, \ldots, a_n\}$ is a set of symbols, we often denote $(a_1 + \ldots + a_n)$ simply by A. We denote the class of all chain regular expressions by RE($\mathcal{S}$).

9.1.2 Decision Problems

The following three problems are fundamental to this chapter. Let $\mathcal{R}$ be a class of regular expressions.

- INCLUSION for $\mathcal{R}$: Given two expressions $r, r' \in \mathcal{R}$, is $L(r) \subseteq L(r')$?
- EQUIVALENCE for $\mathcal{R}$: Given two expressions $r, r' \in \mathcal{R}$, is $L(r) = L(r')$?.
- INTERSECTION NON-EMPTINESS for $\mathcal{R}$: Given an arbitrary number of expressions $r_1, \ldots, r_n \in \mathcal{R}$, is $\bigcap_{i=1}^{n} L(r_i) \neq \emptyset$?

In the sequel, we abuse notation and denote $\bigcap_{i=1}^{n} L(r_i) \neq \emptyset$ simply by $\bigcap_{i=1}^{n} r_i \neq \emptyset$.

9.1.3 Automata on Compressed Strings

We introduce some more notions that are frequently used in our proofs. We use the abbreviations NFA and DFA for non-deterministic and deterministic finite automata, respectively. Such automata are 5-tuples $(Q, \Sigma, \delta, I, F)$, where Q is the state set, Σ is the input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, I is the set of initial states and F is the set of final states. Furthermore, a DFA is an NFA with the property that I is a singleton, and $\delta(q, a)$ is a singleton for every $q \in Q$ and $a \in \Sigma$.

A *compressed string* is a finite sequence of pairs (w, i), where $w \in \Sigma^*$ is a string and $i > 0$ is a natural number. The pair (w, i) stands for the string w^i. The size of a pair (w, i) is $\lceil \log i \rceil$, plus the size of w. The size of a compressed string $v = (w_1, i_i) \cdots (w_n, i_n)$ is the sum of the sizes of $(w_1, i_1), \ldots, (w_n, i_n)$. By string($v$), we denote the *decompressed* string corresponding to v, which is the string $w_1^{i_1} \cdots w_n^{i_n}$. Notice that the size of string(v) can be exponentially larger than the size of v.

The following lemma shows that we can decide in polynomial time whether a compressed string a useful tool to obtain complexity upper bounds in our proofs.

Lemma 9.2. *Given a compressed string v and an NFA A, we can test whether string$(v) \in L(A)$ in time polynomial in the size of v and A.*

Proof. The idea is based on a proof by Meyer and Stockmeyer [SM73], which shows that the equivalence problem for regular expressions over a unary alphabet is coNP-complete. Let $v = (w_1, i_1) \cdots (w_n, i_n)$ be a compressed string and let $A = (Q_A, \Sigma, \delta_A, I_A, F_A)$ be an NFA with $Q_A = \{1, \ldots, k\}$. We basically compute the set $R \subseteq Q_A$ of states which are reachable from a state in I_A by reading string(v) from left to right. When we encounter a pair (w, i), we compute $(M_w)^i$, where M_w is the transition matrix of A on w. The latter can be done by $\mathcal{O}(\log_2 i)$ matrix multiplications. Using $(M_w)^i$, we then replace the current set R by the set of states which are reachable from a state in R, by reading w^i. If, in the end, $R \cap F_A$ is non-empty, we accept.

We now describe this formally. We denote the canonical extension of δ to strings in Σ^* by δ^*. Initially, $R = I_A$. We read v from left to right and repeatedly apply the following rule. When reading (w_j, i_j), we distinguish two cases:

- $i_j = 1$: We replace R with $R' = \{q' \mid q' \in \delta_A^*(q, w_i), q \in R\}$. The latter can be done in time polynomial in the size of v and A in the straightforward manner.
- $i_j > 1$: Let M_w be the $k \times k$ matrix such that for all $\ell, m \in Q_A$, $M_w(\ell, m) = 1$ if $m \in \delta_A^*(\ell, w)$ and $M_w(\ell, m) = 0$, otherwise. Then, we replace R by R' consisting of those q' for which $q \in R$ and $M_w^{i_j}(q, q') = 1$. Notice that M_w can be computed in time polynomial in the size of v and A. Furthermore, by applying the method of successive squaring [Sed83], $M_w^{i_j}$ can be computed by $\mathcal{O}(\log_2(i_j))$ multiplications of $k \times k$-matrices.

Finally, we accept when $R \cap F_A$ is non-empty. □

9.2 Decision Problems for DTDs and XML Schemas

As explained in the introduction, an important motivation for this study comes from reasoning about XML schemas. In this section, we describe how the basic decision problems for such schemas, namely whether two schemas describe the same set of documents or whether one describes a subset of the other, basically reduce to the equivalence and inclusion problem for regular expressions. We also address the problem whether a set of schemas define a common XML document. In the case of DTDs, the latter problem again reduces to the corresponding problem for regular expressions; for XML Schema Definitions (XSDs) it does not.

Remark 9.3. Murata et al. observed that there is a very simple deterministic algorithm to check validity of a tree t with respect to a EDTD$^{\text{st}}$ or EDTD$^{\text{rc}}$ D [MLMK05]. The extension of this algorithm to EDTD$^{\text{td}}$s is straightforward, as we argue here. It proceeds top-down and assigns to every node with some symbol a a type a^i. To the root the start symbol of d is assigned; then, for every interior node u with type a^i, it is checked whether the children of u match $\mu(d(a^i))$; if not, the tree is rejected; otherwise, as D is top-down deterministic, to each child a unique type can be assigned. The tree is accepted, if this process terminates at the leaves without any rejection. ◇

Recall the definition of a reduced DTD from page 14. Recall that an EDTD $(\Sigma, \Delta, d, s_d, \mu)$ is reduced if d is reduced. According to Corollary 3.17, we have that reducing an EDTD(RE) is in PTIME. Unless mentioned otherwise, we assume in the remainder of this chapter that all DTDs and EDTDs are reduced.

We consider the same decision problems for XML schemas as for regular expressions. Let $\mathcal{M}$ be a subclass of the class of DTDs, EDTDs, EDTD$^{\mathrm{st}}$s, EDTD$^{\mathrm{rc}}$s, or EDTD$^{\mathrm{td}}$s.

- INCLUSION for $\mathcal{M}$: Given two schemas $d, d' \in \mathcal{M}$, is $L(d) \subseteq L(d')$?
- EQUIVALENCE for $\mathcal{M}$: Given two schemas $d, d' \in \mathcal{M}$, is $L(d) = L(d')$?
- INTERSECTION NON-EMPTINESS for $\mathcal{M}$: Given the schemas $d_1, \ldots, d_n \in \mathcal{M}$, is $\bigcap_{i=1}^{n} L(d_i) \neq \emptyset$?

9.2.1 Inclusion and Equivalence of XML Schema Languages

As already mentioned, testing equivalence and inclusion of XML schema languages is related to testing equivalence and inclusion of regular expressions. It is immediate that complexity lower bounds for regular expressions imply lower bounds for XML schema languages. A consequence is that testing equivalence and inclusion of XML schemas is PSPACE-hard, which suggests looking for simpler regular expressions.

Interestingly, in the case of the practically important DTDs and single-type EDTDs, it turns out that the complexities of the equivalence and inclusion problem on strings also imply *upper bounds* for the corresponding problems on XML trees.

For a class $\mathcal{R}$ of regular expressions, we denote by DTD($\mathcal{R}$), EDTD($\mathcal{R}$), EDTD$^{\mathrm{st}}$($\mathcal{R}$), EDTD$^{\mathrm{rc}}$($\mathcal{R}$) and EDTD$^{\mathrm{td}}$($\mathcal{R}$), the class of DTDs, EDTDs, EDTD$^{\mathrm{st}}$, EDTD$^{\mathrm{rc}}$s, and EDTD$^{\mathrm{td}}$s with regular expressions in $\mathcal{R}$.

We call a complexity class $\mathcal{C}$ *closed under positive reductions* if the following holds for every $O \in \mathcal{C}$. Let L' be accepted by a deterministic polynomial-time Turing machine M with oracle O (denoted $L' = L(M^O)$). Let M further have the property that $L(M^A) \subseteq L(M^B)$ whenever $A \subseteq B$. Then L' is also in $\mathcal{C}$. For a more precise definition of this notion we refer the reader to [HO02]. For our purposes, it is sufficient that important complexity classes like PTIME, NP, CONP, and PSPACE have this property, and that every such class contains PTIME.

We now show that deciding INCLUSION or EQUIVALENCE for DTDs, EDTD$^{\mathrm{st}}$s, EDTD$^{\mathrm{rc}}$s or EDTD$^{\mathrm{td}}$s is essentially not harder than deciding INCLUSION or EQUIVALENCE for the regular expressions that they use.

Theorem 9.4. *Let $\mathcal{R}$ be a class of regular expressions and $\mathcal{C}$ be a complexity class which is closed under positive reductions. Then the following are equivalent:*

(a) INCLUSION *for $\mathcal{R}$ expressions is in $\mathcal{C}$.*

(b) INCLUSION *for DTD($\mathcal{R}$) is in $\mathcal{C}$.*

(c) INCLUSION *for EDTDst($\mathcal{R}$) is in $\mathcal{C}$.*

Moreover, INCLUSION *for EDTD$^{rc}(\mathcal{R})$ or EDTD$^{td}(\mathcal{R})$ is in $\mathcal{C}$ if* INCLUSION *for $\mu(\mathcal{R})$ expressions is in $\mathcal{C}$.*

The corresponding statements hold for EQUIVALENCE*.*

Proof. The implications (c) ⇒ (b) ⇒ (a) are immediate.

We prove that (a) implies (c). Let $D_1 = (\Sigma, \Delta_1, d_1, s_{d_1}, \mu_1)$ and $D_2 = (\Sigma, \Delta_2, d_2, s_{d_2}, \mu_2)$ be two reduced EDTD$^{\text{td}}(\mathcal{R})$s. We define a correspondence relation $R \subseteq \Delta_1 \times \Delta_2$ as follows:

(1) $(s_{d_1}, s_{d_2}) \in R$; and,

(2) if $(a^i, a^j) \in R$, $w_1 b^k v_1 \in L(d_1(a^i))$, $w_2 b^\ell v_2 \in L(d_2(a^j))$, then $(b^k, b^\ell) \in R$.

We need the following observation further in the proof. The observation follows immediately from the single-type deterministic property of D_1 and D_2 and the fact that D_1 and D_2 are reduced.

Observation 9.5. A pair (a^i, a^j) is in R if and only if there is a tree t with a node u labeled a, such that:

(1) the algorithm of Remark 9.3 assigns type a^i to u with respect to D_1; and

(2) the algorithm of Remark 9.3 assigns type a^j to u with respect to D_2. ◇

Notice that we do not require that t matches D_1 or D_2 overall.

We show that the relation R can be computed in polynomial time in a top-down manner. To this end, let $A_{a^i} = (Q_{a^i}, \Delta_1, \delta_{a^i}, I_{a^i}, F_{a^i})$ and $A_{a^j} = (Q_{a^j}, \Delta_2, \delta_{a^j}, I_{a^j}, F_{a^j})$ be the Glushkov-automata of $d_1(a^i)$ and $d_2(a^j)$, respectively (see [Glu61, BKW98]). We now give an NLOGSPACE decision procedure that tests, given $(a^i, a^j) \in R$ and b^k, b^ℓ, whether there are $w_1 b^k v_1 \in L(d_1(a^i))$ and $w_2 b^\ell v_2 \in L(d_2(a^j))$ with $\mu_1(w_1) = \mu_2(w_2)$ and $\mu_1(v_1) = \mu_2(v_2)$. The algorithm guesses the strings $w_1 b^k v_1$ and $w_2 b^\ell v_2$ one symbol at a time, while simulating A_{a^i} on $w_1 b^k v_1$ and A_{a^j} on $w_2 b^\ell v_2$. For both strings, we only remember the last symbol we guessed. We also only remember one state per automaton. Initially, this is the start state of A_{a^i} and the start state of A_{a^j}. Every time, after guessing a symbol x_1 of $w_1 b^k v_1$ and a symbol x_2 of $w_2 b^\ell v_2$, we overwrite the current states $q_1 \in Q_{a^i}$ and $q_2 \in Q_{a^j}$ by a state in $\delta_{a^i}(q_1, x_1)$ and $\delta_{a^j}(q_2, x_2)$, respectively. The algorithm nondeterministically sets a flag on a moment when $x_1 = b^k$ and $x_2 = b^\ell$. The algorithm is successful when, from the moment that we stop guessing, we are in final states of A_{a^i} and A_{a^j} and the flag is set. As we only remember a flag, two states and two alphabet symbols at the same time, we only use logarithmic space.

We now show how testing INCLUSION and EQUIVALENCE for EDTD$^{\text{st}}$s reduces to testing INCLUSION and EQUIVALENCE for the regular expressions. This follows from Claim 9.6 below and the closure property of $\mathcal{C}$. The statement for EQUIVALENCE follows likewise.

Claim 9.6. *With the notation as above, $L(D_1)$ is included in (equivalent with) $L(D_2)$ if and only if, for every $a^i \in \Sigma'_1$ and $a^j \in \Sigma'_2$ with $(a^i, a^j) \in R$, the regular expression $\mu_1(d_1(a^i))$ is included in (equivalent with) $\mu_2(d_2(a^j))$.*

Proof. INCLUSION for $\text{EDTD}^{\text{rc}}(\mathcal{R})$ or $\text{EDTD}^{\text{td}}(\mathcal{R})$ is in $\mathcal{C}$ if INCLUSION for $\mu(\mathcal{R})$ expressions is in $\mathcal{C}$. We give a proof for INCLUSION. EQUIVALENCE then immediately follows. We can assume without loss of generality that $\mu_1(s_{d_1}) = \mu_2(s_{d_2})$.
($\Rightarrow$) We prove this direction by contraposition. Suppose) that there is a pair $(a^i, a^j) \in R$ for which the regular expression $\mu_1(d_1(a^i))$ is not included in $\mu_2(d_2(a^j))$. We then need to show that $L(D_1)$ is not included in $L(D_2)$.

Thereto, let w be a counterexample Σ-string in $L(\mu_1(d_1(a^i))) - L(\mu_2(d_2(a^j)))$. From Observation 9.5, we now know that there exists a tree $t \in L(D_1)$, with a node $u \in \text{Nodes}(t)$, such that the following holds:

- the type assigned to u with respect to D_1 is a^i;
- the type assigned to u with respect to D_2 is a^j; and
- the concatenation of the labels of u's children is w.

Obviously, t is not in $L(D_2)$ as u's children do not match $\mu_2(d_2(a^j))$. So, $L(D_1)$ is not included in $L(D_2)$.
($\Leftarrow$) We prove this direction by contraposition. Suppose that $L(D_1)$ is not included in $L(D_2)$, so there is a tree t matching D_1 but not D_2. We need to show that there is a $a^i \in \Delta_1$ and $a^j \in \Delta_2$, with $(a^i, a^j) \in R$, such that $\mu_1(d_1(a^i))$ is not included in $\mu_2(d_2(a^j))$.

As $t \notin L(D_2)$, there is a node to which the algorithm in Remark 9.3 assigns a type a^j of D_2, but for which the concatenation of the labels of the children do not match the regular expression $\mu_2(d_2(a^j))$. Let $u \in \text{Nodes}(t)$ be such a node such that no other node on the path in t from the root to u has this property. Let a^i be the type that the algorithm in Remark 9.3 assigns to u with respect to D_1. So, by Observation 9.5, we have that $(a^i, a^j) \in R$. Let w be the concatenation of the labels of u's children. But as $w \in L(\mu_1(d_1(a^i)))$, and w is not in $L(\mu_2(d_2(a^j)))$, we have that $\mu_1(d_1(a^i))$ is not included in $\mu_2(d_2(a^j))$. □

Showing that INCLUSION for $\text{EDTD}^{\text{td}}(\mathcal{R})$ is in $\mathcal{C}$ if INCLUSION for $\mu(\mathcal{R})$ expressions is in $\mathcal{C}$ is analogous to the above proof. The only difference lies in the definition and computation of R. Here,

(1) $(s_{d_1}, s_{d_2}) \in R$; and,

(2) if $(a^i, a^j) \in R$, $w_1 b^k v_1 \in L(d_1(a^i))$, $w_2 b^\ell v_2 \in L(d_2(a^j))$ and $\mu_1(w_1) = \mu_2(w_2)$ then $(b^k, b^\ell) \in R$.

To compute R, the algorithm now additionally has to check, after guessing a symbol x_1 of $w_1 b^k v_1$ and a symbol x_2 of $w_2 b^\ell v_2$, whether $\mu(x_1) = \mu(x_2)$. The corresponding result for $\text{EDTD}^{\text{rc}}(\mathcal{R})$ also immediately follows.

This concludes the proof of Theorem 9.4 □

9.2.2 Intersection of DTDs and XML Schemas

We show in this section that the complexity of the INTERSECTION NON-EMPTINESS problem for regular expressions is an upper bound for the corresponding problem on

DTDs. In Theorem 9.8, we show how the INTERSECTION NON-EMPTINESS problem for DTDs reduces to testing INTERSECTION NON-EMPTINESS for the regular expressions that are used in the DTDs.

Unfortunately, a similar property probably does not hold for the case for single-type, restrained competition, or top-down deterministic EDTDs, as we show in Theorem 9.9. Theorem 9.9 shows that there is a class of EDTD$^{\text{st}}$s for which the intersection non-emptiness problem is EXPTIME-hard. As the intersection non-emptiness problem for regular expressions is PSPACE-complete, the intersection non-emptiness problem for single-type or restrained competition EDTDs cannot be reduced to the corresponding problem for regular expressions, unless EXPTIME = PSPACE.

We start by showing that the intersection non-emptiness problem for DTDs can be reduced to the corresponding problem for regular expressions. Thereto, let $\mathcal{R}$ be a class of regular expressions. The *generalized intersection non-emptiness* problem for $\mathcal{R}$ is to determine, given an arbitrary number of expressions $r_1, \ldots, r_n \in \mathcal{R}$ and a set $S \subseteq \Sigma$, whether $\bigcap_{i=1}^n r_i \cap S^* \neq \emptyset$.

We first show that the intersection non-emptiness problem and the generalized intersection non-emptiness problem are equally complex.

Lemma 9.7. *For a $\mathcal{R}$ a class of regular expressions and $\mathcal{C}$ a complexity class closed under positive reductions. The following are equivalent:*

(a) The intersection non-emptiness problem for $\mathcal{R}$ is in $\mathcal{C}$.

(b) The generalized intersection non-emptiness problem for $\mathcal{R}$ is in $\mathcal{C}$.

Proof. We only prove the direction from (a) to (b). Let $r_1, \ldots, r_n$ be regular expressions in $\mathcal{R}$ and let $S \subseteq \Sigma$. Let r_i' be obtained from r_i by replacing every occurrence of a symbol in $\Sigma - S$ by the regular expression $\emptyset$. Then, $\bigcap_{i=1}^n r_i' \neq \emptyset$ if and only if $\bigcap_{i=1}^n r_i \cap S^* \neq \emptyset$. □

We are now ready to show the theorem for DTDs.

Theorem 9.8. *Let $\mathcal{R}$ be a class of regular expressions and let $\mathcal{C}$ be a complexity class which is closed under positive reductions. Then the following are equivalent:*

(a) The INTERSECTION NON-EMPTINESS *problem for $\mathcal{R}$ expressions is in $\mathcal{C}$.*

(b) The INTERSECTION NON-EMPTINESS *problem for DTD($\mathcal{R}$) is in $\mathcal{C}$.*

Proof. We only prove the direction from (a) to (b), as the reverse direction is trivial. Thereto, let $d_1, \ldots, d_n$ be in DTD($\mathcal{R}$). We assume without loss of generality that $d_1, \ldots, d_n$ all have the same start symbol. We compute the set of symbols S_i with the following property:

$$a \in S_i \text{ if and only if there is a tree of depth at most } i \\ \text{with root labeled } a \text{ in } L((d_1, a)) \cap \cdots \cap L((d_n, a)). \qquad (*)$$

Initially, we set $S_1 = \{a \mid \varepsilon \in L(d_j(a)) \text{ for all } j \in \{1, \ldots, n\}\}$. For every $i > 1$, S_i is the set S_{i-1} extended with all symbols a for which $S_{i-1}^* \cap d_1(a) \cap \cdots \cap d_n(a) \neq \emptyset$. Clearly, $S_{k+1} = S_k$ for $|\Sigma| = k$. It can be shown by a straightforward induction on i

that $(*)$ holds. So, there is a tree satisfying all DTDs if and only if the start symbol belongs to S_k.

It remains to argue that S_k can be computed in $\mathcal{C}$. Clearly, for any set of regular expressions, it can be checked in PTIME whether their intersection accepts ε. So, S_1 can be computed in PTIME and hence in $\mathcal{C}$. From Lemma 9.7, it follows that S_i can be computed from S_{i-1} in $\mathcal{C}$. As only k sets need to be computed, the overall algorithm is in $\mathcal{C}$. This concludes the proof of Theorem 9.8. □

The following theorem shows that single-type and restrained competition EDTDs probably cannot be included in Theorem 9.8. Indeed, by Theorem 9.18, the intersection non-emptiness problem for $\text{RE}((+a), w?, (+a)?, (+a)^*)$ expressions is in NP and by Theorem 9.9, the intersection non-emptiness problem is already EXPTIME-complete for single-type EDTDs with $\text{RE}((+a), w?, (+a)?, (+a)^*)$ expressions. The proof of Theorem 9.9 is similar to the proof that intersection non-emptiness of deterministic top-down tree automata is EXPTIME-complete [Sei94]. However, single-type EDTDs and the latter automata are incomparable. Indeed, the tree language consisting of the trees $\{a(bc), a(cb)\}$ is not definable by a top-down deterministic tree automaton, while it is by the EDTD consisting of the rules $a^1 \to b^1c^1 + c^1b^1$, $b^1 \to \varepsilon$, $c^1 \to \varepsilon$. Conversely, the tree language $\{a(b(c)b(d))\}$ is not definable by a single-type EDTD, but is definable by a top-down deterministic tree automaton.

Theorem 9.9. *The* INTERSECTION NON-EMPTINESS *problem for* $\mathit{EDTD}^{st}((+a), w?, (+a)?, (+a)^*)$ *is* EXPTIME-*hard.*

Proof. We use a reduction from TWO-PLAYER CORRIDOR TILING, which is EXPTIME-hard (Theorem 3.23). Let $D = (T, H, V, \bar{b}, \bar{t}, n)$ be a tiling system with $T = \{t_1, \dots, t_k\}$. We construct several single-type EDTDs such that their intersection is non-empty if and only if player CONSTRUCTOR has a winning strategy.

As Σ we take $T \cup \{\#\}$. We define d_0 to be a single-type EDTD defining all possible strategy trees. Every path in such a tree will encode a tiling. The root is labeled with #. Inner nodes are labeled with tiles. Nodes occurring on an even depth are placed by player CONSTRUCTOR and have either no children or have every tile in T as a child representing every possible answer of SPOILER. Nodes occurring on an odd depth are placed by player SPOILER and have either no children or precisely one child representing the choice of CONSTRUCTOR. The start symbol of d_0 is #. The EDTD d_0 uses the alphabet $\Sigma' = \{\#, \texttt{error}, t_1^1, \dots, t_k^1, t_1^2, \dots, t_k^2\}$. The rules are as follows:

- $\# \to (t_1^1 + \cdots + t_k^1)$;
- for every $t \in T$, $t^1 \to (t_1^2 \cdots t_k^2)?$; and
- for every $t \in T$, $t^2 \to (t_1^1 + \cdots + t_k^1 + \texttt{error})?$.

Here, a tile with label t_i is assigned the type t_i^1 (respectively t_i^2) if it corresponds to a move of player CONSTRUCTOR (respectively SPOILER). We use the special symbol `error` to mark that player SPOILER has placed a wrong tile. Notice that $\bar{b}$ and $\bar{t}$ are not present in the tree.

All other single-type EDTDs will check the correct shape of the tree and the horizontal and vertical constraints.

First, we make the following observation. Let $M = (Q, \Sigma, \delta, \{0\}, F)$ be a DFA with state set $Q = \{0, \dots, m\}$, with the property that there is an $a \in \Sigma$ such that every string in $L(M)$ starts with a. Then, a single-type EDTD d_M can be constructed in LOGSPACE defining the trees over Σ for which every path from the root to a leaf is accepted by M. Indeed, we let a^0 be the start symbol. Then, for every $i, j \in Q$ and $b \in \Sigma$ for which $\delta(i, b) = \{j\}$, d_M contains the rule $b^i \to (t_1^j + \cdots + t_k^j + \#^j)^*$ if $i \in F$, and $b^i \to (t_1^j + \cdots + t_k^j + \#^j)^+$, otherwise.

Let $M_1, \dots, M_\ell$ be a sequence of DFAs that check the following properties:

- Every string starts with #, has no other occurrences of #, and either *(i)* ends with `error` or *(ii)* the length of every string is one modulo n. This can be checked by one automaton.
- All horizontal constraints are satisfied, or player SPOILER places the first tile which violates the horizontal constraints. This tile is followed by the special symbol `error`. This can be checked by one automaton.
- For every position $i = 1, \dots, n$, all vertical constraints on tiles on a position i (mod n) are satisfied, or player SPOILER places a the first tile which violates the vertical constraindt. This tile is followed by the special symbol `error`. This can be checked by n automata.
- For every position $i = 1, \dots, n$, the ith tile of $\bar{b}$ and the ith tile of the first row should satisfy the vertical constraints. This can be checked by one automaton.
- For every position $i = 1, \dots, n$, the ith tile of the last row and the ith tile of $\bar{t}$ should satisfy the vertical constraints. This can be checked by one automaton.

Clearly, $d_0 \cap d_{M_1} \cap \cdots \cap d_{M_\ell}$ is non-empty if and only if player CONSTRUCTOR has a winning strategy. □

9.3 Inclusion

We now turn to the complexity of INCLUSION, EQUIVALENCE, and INTERSECTION NON-EMPTINESS for the chain regular expressions themselves. We start our investigation with the inclusion problem. As mentioned before, it is PSPACE-complete for general regular expressions. The following tractable cases have been identified in the literature:

- INCLUSION for $RE(a?, (+a)^*)$ can be solved in linear time. Whether $p \subseteq p_1 + \cdots + p_k$ for $p, p_1, \dots, p_k$ from $\mathrm{RE}(a?, (+a)^*)$ can be checked in quadratic time [ABJ98].
- In [MS99a] it is stated that INCLUSION for $\mathrm{RE}(a, \Sigma, \Sigma^*)$ is in PTIME. A proof can be found in [MS04].

Some of the fragments we defined are so small that one expects their containment problem to be tractable. Therefore, it comes as a surprise that even for $\mathrm{RE}(a, a?)$ and $\mathrm{RE}(a, a^*)$ the inclusion problem is already coNP-complete. Even worse, for

RE$(a, (+a)^*)$ and RE$(a, (+a)^+)$ we already obtain the maximum complexity: PSPACE-completeness. The PSPACE-hardness of inclusion of RE$(a, (+a)^*)$ expressions should be contrasted with the PTIME inclusion for RE(a, Σ, Σ^*) obtained in [MS99a], where disjunctions can only range over the complete alphabet.

Our results, together with corresponding upper bounds are summarized in Theorem 9.10, which we prove in a series of lemmas (Lemma 9.12–9.15). Let RE$(\mathcal{S} - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\})$ denote the fragment of RE$(\mathcal{S})$ where no factors of the form $(a_1 + \cdots + a_n)^*$, $(w_1 + \cdots + w_n)^*$, $(a_1 + \cdots + a_n)^+$ or $(w_1 + \cdots + w_n)^+$ are allowed for $n \geq 2$.

Theorem 9.10. *(a)* INCLUSION *is co*NP*-hard for*

(1) RE(a, a^*),
(2) RE$(a, a?)$,
(3) RE$(a, (+a^+))$,
(4) RE(a, w^+),
(5) RE$(a^+, (+a))$;

(b) INCLUSION *is* PSPACE*-hard for*

(1) RE$(a, (+a)^+)$; and
(2) RE$(a, (+a)^*)$;

(c) INCLUSION *is in co*NP *for RE$(\mathcal{S} - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\})$; and*

(d) INCLUSION *is in* PSPACE *for RE$(\mathcal{S})$.*

Theorem 9.10 does not leave much room for tractable cases. Of course, INCLUSION is in PTIME for any class of regular expressions for which expressions can be transformed into DFAs in polynomial time. An easy example of such a class is RE(a, a^+).

Another example has probably more importance in practice. Often, the same symbol occurs only a few times in a regular expression of a DTD. As a matter of fact, if we impose a fixed bound k on the number of such occurrences, then the containment problem becomes tractable. For every k, let RE$^{\leq k}$ denote the class of all regular expressions where every symbol can occur at most k times.

Theorem 9.11. INCLUSION *for RE$^{\leq k}$ is in* PTIME.

Proof. Let r be an RE$^{\leq k}$ expression. Let $A_r = (Q, \Sigma, \delta, I, F)$ be the Glushkov automaton for r [Glu61] (see also [BKW98]). As the states of this automaton are basically the positions in r, after reading a symbol there are always at most k possible states in which the automaton might be. Therefore, determinizing A only leads to a DFA of size $|A|^k$. As k is fixed, inclusion of such automata is in PTIME. □

It should be noted though that the upper bound for the running time is $\mathcal{O}(n^k)$, therefore k should be very small to be really useful. Fortunately, this seems to be the case in many practical scenarios. Indeed, recent investigation has pointed out that in practice, for ninety-nine percent of the regular expressions DTDs or XML Schemas, k is equal to one [BNST05].

In the rest of this section, we prove Theorem 9.10.

Lemma 9.12 (Theorem 9.10(a)). INCLUSION *is co*NP*-hard for*

(1) $RE(a, a^*)$,

(2) $RE(a, a?)$,

(3) $RE(a, (+a^+))$,

(4) $RE(a, w^+)$; *and*

(5) $RE(a^+, (+a))$.

Proof. We show that for all five cases, there is a LOGSPACE reduction from VALIDITY of propositional 3DNF formulas. The VALIDITY problem asks, given a propositional formula Φ in 3DNF with variables $\{x_1, \ldots, x_n\}$, whether Φ is true under all truth assignments for $\{x_1, \ldots, x_n\}$. The VALIDITY problem for 3DNF formulas is known to be coNP-complete [GJ79]. We note that, for the cases (1–3), we even show that INCLUSION is already coNP-hard when the expressions use a fixed-size alphabet.

Our proof technique is inspired by a proof of Miklau and Suciu, showing that the inclusion problem for XPath expressions with predicates, wildcard, and the axes "child" and "descendant" is coNP-hard [MS04]. We present a robust generalization and use it to show coNP-hardness of all five fragments.

We now proceed with the proof. Thereto, let $\Phi = C_1 \vee \cdots \vee C_k$ be a propositional formula in 3DNF using variables $\{x_1, \ldots, x_n\}$. In the five cases, we construct regular expressions R_1, R_2 such that

$$L(R_1) \subseteq L(R_2) \text{ if and only if } \Phi \text{ is valid.}$$

More specifically, we encode truth assignments for Φ by strings. The basic idea is to construct R_1 and R_2 such that $L(R_1)$ contains all string representations of truth assignments and a string w matches R_2 if and only if w represents an assignment which makes Φ true.

We discuss the building blocks of expressions R_1 and R_2. Let U be a regular expression describing exactly one string u. In this proof, U is either of the form a^n or of the form $\#a^i\$\cdots\$a^i\#$ (n occurrences of a^i separated by \$) for some integer i. We construct a regular expression W such that the strings of $L(W)$ can be interpreted as truth assignments. More precisely, for each truth assignment A, there is a string $w_A \in L(W)$ and for each string $w \in L(W)$ there is a corresponding truth assignment A_w. Then we set

$$\begin{array}{rcl} R_1 & = & U^k W U^k, \\ R_2 & = & N F_1 \cdots F_k N, \end{array}$$

where N and F_i, $i = 1, \ldots, k$, are regular expressions for which the following properties hold:

(i) $u^i \in L(N)$ for every $i = 1, \ldots, k$.

(ii) If A_w makes C_i true then $w \in L(F_i)$. If $w_A \in L(F_i)$ then A makes C_i true.

(iii) $u \in L(F_i)$ for every $i = 1, \ldots, k$.

(iv) If $u^k w u^k \in L(R_1) \cap L(R_2)$, then w matches some F_i.

We first show the following claim:

Claim 9.13. *If there are expressions U, W, N, $F_1, \ldots, F_k$ satisfying (i)–(iv) above, then*

$$L(R_1) \subseteq L(R_2) \textit{ if and only if } \Phi \textit{ is valid.}$$

Proof. Suppose that there are expressions U, W, N, $F_1, \ldots, F_k$ satisfying (i)–(iv). We prove that $L(R_1) \subseteq L(R_2)$ if and only if Φ is valid.

($\Rightarrow$) Assume that $L(R_1) \subseteq L(R_2)$. Let A be an arbitrary truth assignment for Φ and let $v = u^k w_A u^k$. As $v \in L(R_1)$ and $L(R_1) \subseteq L(R_2)$, we also have that $v \in L(R_2)$. By (iv), it follows that w_A matches some F_i. Hence, by (ii), C_i is made true by A. Consequently, Φ is also made true by A. As A is an arbitrary truth assignment, we have that Φ is a valid propositional formula.

($\Leftarrow$) Suppose that Φ is valid. Let v be an arbitrary string in $L(R_1)$. By definition of R_1, v is of the form $v = u^k w u^k$, where u is the unique string in $L(U)$. Consider the truth assigment A_w corresponding to w. As Φ is valid, there is a clause C_i of Φ that becomes true under A_w. Due to (ii), we have that $w \in L(F_i)$. Furthermore, as u matches F_j for every $j = 1, \ldots, k$ (by (iii)), and as $L(N)$ contains u^ℓ for every $\ell = 1, \ldots, k$ (by (i)), we have that $v = u^k w u^k \in L(R_2)$. As v is an arbitrary string in $L(R_1)$, we have that $L(R_1) \subseteq L(R_2)$. This completes the proof of the claim. □

It remains to construct regular expressions U, W, N, $F_1, \ldots, F_k$ with the required properties. In all five cases, we construct these expressions starting from five basic regular expressions r^{true}, r^{false}, $r^{\text{true,false}}$, r^{all}, and α, which must adhere to the inclusion structure graphically represented in Figure 9.1 and formally defined by the properties (INC1)–(INC5) below. Intuitively, the two dots in Figure 9.1 are strings z^{true} and z^{false}, which represent the truth values *true* and *false*, respectively. The expressions r^{true} and r^{false} are used to match z^{true} and z^{false} in R_2, respectively. The expression $r^{\text{true,false}}$ is used to generate the truth values *true* and *false*. It will be used in R_1, allowing R_1 to generate all truth assignments which must then be matched in R_2. Finally, α and r^{all} are used to ensure that condition (iv) above holds. That is, they ensure that when $L(R_1) \subseteq L(R_2)$, every string generated by W must match some F_i.

In all cases, the expressions r^{true}, r^{false}, $r^{\text{true,false}}$, r^{all}, and α have the properties (INC1)–(INC5):

$$\alpha \in L(r^{\text{false}}) \cap L(r^{\text{true}}) \tag{INC1}$$

$$L(r^{\text{true,false}}) \subseteq L(r^{\text{false}}) \cup L(r^{\text{true}}) \tag{INC2}$$

$$L(r^{\text{true,false}}) \cup \{\alpha\} \subseteq L(r^{\text{all}}) \tag{INC3}$$

$$z^{\text{true}} \in L(r^{\text{true,false}}) - L(r^{\text{false}}) \tag{INC4}$$

$$z^{\text{false}} \in L(r^{\text{true,false}}) - L(r^{\text{true}}) \tag{INC5}$$

For the first three fragments, we now define the needed expressions. We deal with the other two fragments later. Note that the alphabet size is fixed (at most four) in the reduction for these fragments.

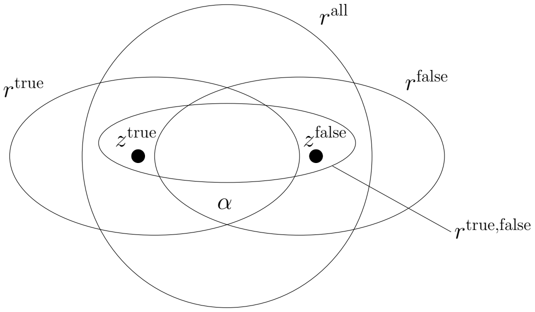


Figure 9.1: Inclusion structure of regular expressions used in coNP-hardness of INCLUSION.

(1) For RE(a, a^*):

- $\alpha = a$;
- $r^{\text{true}} = aa^*b^*a^*$;
- $r^{\text{false}} = b^*a^*$;
- $r^{\text{true,false}} = r^{\text{all}} = a^*b^*a^*$;
- $z^{\text{true}} = ab$;
- $z^{\text{false}} = ba$;
- $U = \#\alpha\$\alpha\$ \cdots \$\alpha\#$ (n occurrences of α);
- $W = \#r^{\text{true,false}}\$ \cdots \$r^{\text{true,false}}\#$ (n occurrences of $r^{\text{true,false}}$); and
- $N = (\#^*a^*\$^*a^*\$^* \cdots \$^*a^*\#^*)^k$ (n occurrences of a^* in each of the k copies);

(2) For RE$(a, a?)$:

- $\alpha = a$;
- $r^{\text{true}} = aa?$;
- $r^{\text{false}} = a?$;
- $r^{\text{true,false}} = r^{\text{all}} = a?a?$;
- $z^{\text{true}} = aa$;
- $z^{\text{false}} = \varepsilon$;
- $U = \#\alpha\$\alpha\$ \cdots \$\alpha\#$ (n occurrences of α);
- $W = \#r^{\text{true,false}}\$ \cdots \$r^{\text{true,false}}\#$ (n occurrences of $r^{\text{true,false}}$); and
- $N = (\#?a?\$?a?\$? \cdots \$?a?\#?)^k$ (n occurrences of $a?$ in each of the k copies)

(3) For RE(a, w^+):

- $\alpha = aaaa$;
- $r^{\text{true}} = a^+(aa)^+$;

- $r^{\text{false}} = (aa)^+$;
- $r^{\text{true,false}} = aa^+$;
- $r^{\text{all}} = a^+$;
- $z^{\text{true}} = aaa$;
- $z^{\text{false}} = aa$;
- $U = \#\alpha\$\alpha\$\cdots\$\alpha\#$ (n occurrences of α);
- $W = \#r^{\text{true,false}}\$\cdots\$r^{\text{true,false}}\#$ (n occurrences of $r^{\text{true,false}}$); and
- $N = (\#aaaa\$aaaa\$\cdots\$aaaa\#)^+$ (n occurrences of $aaaa$)

It is straighforward to verify that the conditions (INC1)–(INC5) are fulfilled for each of the fragments.

With $w = \#w_1\$\cdots\$w_n\# \in L(W)$ we associate a truth assignment A_w as follows:

$$A_w(x_j) := \begin{cases} \text{true}, & \text{if } w_j \in L(r^{\text{true}}); \\ \text{false}, & \text{otherwise.} \end{cases}$$

Let $z^{\text{false}} \in L(r^{\text{true,false}}) - L(r^{\text{true}})$ and $z^{\text{true}} \in L(r^{\text{true,false}}) - L(r^{\text{false}})$. They exist by conditions (INC4) and (INC5). For a truth assignment A, let

$$w_A = \#w_1\$\cdots\$w_n\#,$$

where, for each $j = 1, \ldots, n$, $w_j = z^{\text{true}}$ if $A(x_j) = \text{true}$ and $w_j = z^{\text{false}}$, otherwise.

For each $i = 1, \ldots, k$, we set

$$F_i = \#e_1\$\cdots\$e_n\#,$$

where for each $j = 1, \ldots, n$,

$$e_j := \begin{cases} r^{\text{false}}, & \text{if } x_j \text{ occurs negated in } C_i, \\ r^{\text{true}}, & \text{if } x_j \text{ does not occur negated in } C_i, \text{ and} \\ r^{\text{all}}, & \text{otherwise.} \end{cases}$$

It remains to show that, for each of the fragments, conditions (i)–(iv) hold:

(i) Trivial.

(ii) Let $w = \#w_1\$\cdots\$w_n\# \in W$ be a string for which A_w makes C_i true and let $F_i = \#e_1\$\cdots\$e_n\#$ be as defined above. We need to show that $w \in L(F_i)$. Thereto, let $j \leq n$ be an arbitrary positive integer. We need to consider three cases:

1. If x_j does not occur in C_i then $e_j = r^{\text{all}}$, by definition of e_j. Hence, as $w_j \in L(r^{\text{true,false}})$, and by condition (INC3), we have that $w_j \in L(e_j)$.
2. If x_j occurs positively, then $e_j = r^{\text{true}}$, by definition of e_j. As A_w makes C_i true, we know that $A_w(x_j) = \text{true}$. By definition of A_w, we know that $w_j \in L(r^{\text{true}}) = L(e_j)$.

3. If x_j occurs negatively, then $e_j = r^{\text{false}}$, by definition of e_j. As A_w makes C_i true, we know that $A_w(x_j) = \text{false}$. As $w_j \in r^{\text{true,false}}$ and $w_j \notin L(r^{\text{true}})$ by definition of $A_w(x_j)$, condition (INC2) gives that $w_j \in L(r^{\text{false}}) = L(e_j)$.

As $j \leq n$ is an arbitrarily chosen positive integer, we have that for each $j = 1, \ldots, n, w_j \in L(e_j)$. Consequently, we also have that $w \in L(F_i)$.

We show the other statement by contraposition. Thereto, let A be a truth assignment such that C_i is a clause not fulfilled by A. We need to show that $w_A \notin L(F_i)$. We need to consider two cases:

1. Suppose there exists an x_j which occurs positively in C_i and $A(x_j)$ is false. By definition of F_i, the e_j component of F_i is r^{true} and, by definition of w_A, the w_j component of w_A is $z^{\text{false}} \notin L(r^{\text{true}})$. Hence, $w_A \notin L(F_i)$.
2. Otherwise, there exists an x_j which occurs negatively in C_i and $A(x_j)$ is true. By definition of F_i, the e_j component of F_i is r^{false} and, by definition of w_A, the w_j component of w_A is $z^{\text{true}} \notin L(r^{\text{false}})$. Hence, $w_A \notin L(F_i)$.

(iii) This follows immediately from conditions (INC1), (INC3), and the definition of F_i.

(iv) Suppose that $u^k w u^k \in L(R_1) \cap L(R_2)$. We need to show that w matches some F_i. Observe that the strings u, w, and every string in every $L(F_i)$ is of the form $\#y\#$ where y is a non-empty string over the alphabet $\{a, b, \$\}$. Also, every string in $L(N)$ is of the form $\#y_1\#\#y_2\# \cdots \#y_\ell\#$, where $y_1, \ldots, y_\ell$ are non-empty strings over $\{a, \$\}$. Hence, as $u^k w u^k \in L(R_2)$ and as none of the strings $y, y_1, \ldots, y_\ell$ contain the symbol "#", we have that w either matches some F_i or w matches a sub-expression of N.

We now distinguish between fragments (1–2) and fragment (3). Let m be a match between $u^k w u^k$ and R_2.

- In fragments (1–2) we have that $\ell \leq k$. Towards a contradiction, assume that m matches a superstring of $u^k w$ to the left occurrence of N in R_2. Note that $u^k w$ is a string of the form $\#y'_1\#\#y'_2\# \cdots \#y'_{k+1}\#$, where $y'_1, \ldots, y'_{k+1}$ are non-empty strings over $\{a, b, \$\}$. But as $\ell \leq k$, no superstring of $u^k w$ can match N, which is a contradiction. Analogously, no superstring of wu^k matches the right occurrence of the expression N in R_2. So, m must match w onto some F_i.
- In fragment (3), we have that $\ell \geq 1$. Again, towards a contradiction, assume that m matches a superstring of $u^k w$ onto the left occurrence of the expresion N in R_2. Observe that every string that matches $F_1 \cdots F_k N$ is of the form $\#y''_1\#\#y''_2\# \cdots \#y''_{\ell'}\#$, where $\ell' > k$. As u^k is not of this form, m cannot match u^k onto $F_1 \cdots F_k N$, which is a contradiction. Analogously, m cannot match a superstring of wu^k onto the right occurrence of the expression N in R_2. So, m must match w onto some F_i.

This proves Lemma 9.12 for fragments (1)–(3).

We still need to deal with the fragments (4) and (5). The main difference with the fragments (1)–(3) is that we will no longer use an alphabet with fixed size. Instead, we use the symbols b_j and c_j, for $j = 1, \ldots, n$. Instead of the basic regular expressions r^{true}, r^{false}, $r^{\text{true,false}}$, and r^{all}, we will now have expressions r_j^{true}, r_j^{false}, $r_j^{\text{true,false}}$, and r_j^{all} for every $j = 1, \ldots, n$. We will require that these expressions have the same properties (INC1)–(INC5), but only between the expressions r_j^{true}, r_j^{false}, $r_j^{\text{true,false}}$, and r_j^{all} with the same index j, and α.

The needed regular expressions are then defined as follows:

(4) For RE$(a, (+a^+))$:

- $\alpha = a$;
- $r_j^{\text{true}} = (a^+ + b_j^+)$;
- $r_j^{\text{false}} = (a^+ + c_j^+)$;
- $r_j^{\text{true,false}} = (b_j^+ + c_j^+)$;
- $r_j^{\text{all}} = (a^+ + b_j^+ + c_j^+)$;
- $z_j^{\text{true}} = b_j$;
- $z_j^{\text{false}} = c_j$;
- $U = \alpha^n$;
- $W = r_1^{\text{true,false}} \cdots r_n^{\text{true,false}}$; and
- $N = a^+$

(5) For RE$(a^+, (+a))$:

- $\alpha = a$;
- $r_j^{\text{true}} = (a + b_j)$;
- $r_j^{\text{false}} = (a + c_j)$;
- $r_j^{\text{true,false}} = (b_j + c_j)$;
- $r_j^{\text{all}} = (a + b_j + c_j)$;
- $z_j^{\text{true}} = b_j$;
- $z_j^{\text{false}} = c_j$;
- $U = \alpha^n$;
- $W = r_1^{\text{true,false}} \cdots r_n^{\text{true,false}}$; and
- $N = a^+$

With $w = w_1 \cdots w_n \in L(W)$, where for every $j = 1, \ldots, n$, $w_j \in r_j^{\text{true,false}}$, we associate a truth assignment A_w as follows:

$$A_w(x_j) := \begin{cases} \text{true}, & \text{if } w_j \in L(r^{\text{true}}); \\ \text{false}, & \text{otherwise.} \end{cases}$$

Let $z_j^{\text{false}} \in L(r_j^{\text{true,false}}) - L(r_j^{\text{true}})$ and $z_j^{\text{true}} \in L(r_j^{\text{true,false}}) - L(r_j^{\text{false}})$. They exist by conditions (INC4) and (INC5). For a truth assignment A, let

$$w_A = w_1 \cdots w_n,$$

where, for each $j = 1, \ldots, n$, $w_j = z_j^{\text{true}}$ if $A(x_j) = \text{true}$ and $w_j = z_j^{\text{false}}$, otherwise.

For each $i = 1, \ldots, k$, we set

$$F_i = e_1 \cdots e_n,$$

where for each $j = 1, \ldots, n$,

$$e_j := \begin{cases} r_j^{\text{false}}, & \text{if } x_j \text{ occurs negated in } C_i, \\ r_j^{\text{true}}, & \text{if } x_j \text{ does not occur negated in } C_i, \text{ and} \\ r_j^{\text{all}}, & \text{otherwise.} \end{cases}$$

We show that, for fragments (4) and (5), conditions (i)–(iv) hold:

(i) Trivial.

(ii) This can be shown analogously as for the fragments (1)–(3).

(iii) This follows immediately from conditions (INC1), (INC3), and the definition of F_i.

(iv) Suppose that $u^k w u^k \in L(R_1) \cap L(R_2)$. We need to show that w matches some F_i. To this end, let m be a match between $u^k w u^k$ and R_2. For every $j = 1, \ldots, n$, let Σ_j denote the set $\{b_j, c_j\}$. Observe that the string w is of the form $y_1 \cdots y_n$, where, for every $j = 1, \ldots, n$, y_j is a string in Σ_j^+. Moreover, no strings in $L(N)$ contain symbols from Σ_j for any $j = 1, \ldots, n$. Hence, m cannot match any symbol of the string w onto N. Consequently, m matches the entire string w onto a subexpression of $F_1 \cdots F_k$ in R_2.

Further, observe that every string in every F_i, $i = 1, \ldots, k$, is of the form $y'_1 \ldots y'_n$, where each y'_j is a string in $(\Sigma_j \cup \{a\})^+$. As m can only match symbols in Σ_j onto subexpressions with symbols in Σ_j, m matches w onto some F_i.

This concludes the proof of Lemma 9.12. □

We prove Theorem 9.10(b) by a reduction from Corridor Tiling (Theorem 3.23).

Lemma 9.14 (Theorem 9.10(b)). inclusion *is* pspace*-hard for*

(1) RE$(a, (+a)^+)$; and

(2) RE$(a, (+a)^*)$.

Proof. We first show that inclusion is pspace-hard for RE$(a, (+a)^+)$ and we consider the case of RE$(a, (+a)^*)$ later. In both cases, we use a reduction from the Corridor Tiling problem, which is known to be pspace-complete (Theorem 3.23).

To this end, let $D = (T, H, V, \bar{b}, \bar{t}, n)$ be a tiling system. Without loss of generality, we assume that $n \geq 2$. We construct two regular expressions R_1 and R_2 such that

$$R_1 \subseteq R_2 \text{ if and only if there exists no correct corridor tiling for } D.$$

Let $\Sigma_i = \{t_i \mid t \in T\}$, which is the alphabet we will use to tile the i-th column. Set $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$. For ease of exposition, we denote $\Sigma \cup \{\$\}$ by $\Sigma_\$$ and $\Sigma \cup \{\#, \$\}$ by $\Sigma_{\#,\$}$. We encode candidates for a correct tiling by a string in which the rows are separated by the symbol \$, that is, by strings of the form

$$\$\bar{b}\$\Sigma^+\$\Sigma^+\$\cdots\$\Sigma^+\$\bar{t}\$. \qquad (\dagger)$$

The following regular expressions detect strings of this form which do not encode a correct tiling:

- $\Sigma_\$^+ t_i t'_j \Sigma_\$^+$, for every $t, t' \in T$, where $i = 1, \ldots, n-1$ and $j \neq i+1$. These expressions detect consecutive symbols that are not from consecutive column sets;

- $\Sigma_\$^+ \$ t_i \Sigma_\$^+$ for every $i \neq 1$ and $t_i \in \Sigma_i$, and $\Sigma_\$^+ t_i \$ \Sigma_\$^+$ for every $i \neq n$ and $t_i \in \Sigma_i$. These expressions detect rows that do not start or end with a correct symbol. Together with the previous expressions, these expressions detect all candidates with at least one row not in $\Sigma_1 \cdots \Sigma_n$.

- $\Sigma_\$^+ t_i t'_{i+1} \Sigma_\$^+$, for every $(t, t') \notin H$, and $i = 1, \ldots, n-1$. These expressions detect all violations of horizontal constraints.

- $\Sigma_\$^+ t_i \Sigma^+ \$ \Sigma^+ t'_i \Sigma_\$^+$, for every $(t, t') \notin \Sigma$ and for every $i = 1, \ldots, n$. These expressions detect all violations of vertical constraints.

Let $e_1, \ldots, e_k$ be an enumeration of the above expressions. Notice that $k = \mathcal{O}(|D|^4)$. It is straightforward that a string w in ($\dagger$) does not match $\bigcup_{i=1}^{k} e_i$ if and only if w encodes a correct tiling.

Let $e = e_1 \cdots e_k$. Because of leading and trailing $\Sigma_\$^+$ expressions, $L(e) \subseteq L(e_i)$, for every $i = 1, \ldots, k$. We are now ready to define R_1 and R_2:

$$\begin{array}{rcl} R_1 & = & \overbrace{\#e\#e\#\cdots\#e\#}^{k \text{ times } e}\,\$\bar{b}\$\Sigma_\$^+\$\bar{t}\$\,\overbrace{\#e\#e\#\cdots\#e\#}^{k \text{ times } e};\ \text{ and,} \\ R_2 & = & \Sigma^+_{\#,\$}\#e_1\#e_2\#\cdots\#e_k\#\Sigma^+_{\#,\$}. \end{array}$$

Notice that both R_1 and R_2 are in $\mathrm{RE}(a, (+a)^+)$ and can be constructed in polynomial time. It remains to show that $R_1 \subseteq R_2$ if and only if there is no correct tiling for D.

We first show the implication from left to right. Thereto, let $R_1 \subseteq R_2$. Let uwu' be an arbitrary string in $L(R_1)$ such that $u, u' \in L(\#e\#e\#\cdots\#e\#)$ and $w \in \$\bar{b}\$\Sigma_\$^+\$\bar{t}\$$. Hence, $uwu' \in L(R_2)$. Let m be a match between uwu' and R_2. Notice that uwu' contains $2k+2$ times the symbol "#". However, as uwu' starts and ends with the symbol "#", m matches the first and the last "#" of uwu' onto the $\Sigma^+_{\#,\$}$ subexpressions of R_2 ($\ddagger$). This means that m matches $k+1$ consecutive #-symbols of the remaining $2k$ #-symbols in uwu' onto the #-symbols in $\#e_1\#e_2\#\cdots\#e_k\#$. Hence, m matches w onto some e_i. So, w does not encode a correct tiling. As the sub-expression $\$\bar{b}\$\Sigma_{\$}^{+}\$\bar{t}\$$ of R_1 defines all candidate tilings, the system D has no solution.

To show the implication from right to left, assume that there is a string $uwu' \in L(R_1)$ that is not in R_2, where $u, u' \in L(\#e\#e\#\cdots\#e\#)$. Then $w \notin \bigcup_{i=1}^{k} L(e_i)$ and, hence, w encodes a correct tiling.

The PSPACE-hardness proof for RE$(a, (+a)^*)$ is completely analogous, except that every "+" (which is not a disjunction) has to be replaced by a "$*$" and that $R_2 = \#\Sigma_{\#,\$}^{*}\#e_1\#e_2\#\cdots\#e_k\#\Sigma_{\#,\$}^{*}\#$. The addition of the start and end symbol "#" is to enforce the condition (‡) above. □

Lemma 9.15 (Theorem 9.10(c))**.** INCLUSION *is in co*NP *for*

$$RE(\mathcal{S} - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\}).$$

Proof. Let r_1, r_2 be expressions in RE$(\mathcal{S} - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\})$. By translating the regular expressions to NFAs and determinizing them, it is easy to see that when $r_1 \not\subseteq r_2$, there is a counterexample string s of at most exponential size in $|r_1| + |r_2|$, such that $s \in L(r_1)$ but $s \notin L(r_2)$. Let N_{r_1,r_2} denote this size.

Because of the restricted form of r_1, it is possible to encode s as a compressed string s' of size polynomial in $|r_1| + |r_2|$. Indeed, r_1 is of the form $e_1 \cdots e_n$ where each e_i is of the form $(w_1 + \cdots + w_k)$, $(w_1 + \cdots + w_k)?$, $(w_1^* + \cdots + w_k^*)$ or $(w_1^+ + \cdots + w_k^+)$, for $k \geq 1$ and $w_1, \ldots, w_k \in \Sigma^+$. Hence, s can be written as $s_1 \cdots s_n$, where each s_i matches e_i. Unless e_i is of the form $(w_1^* + \cdots + w_k^*)$ or $(w_1^+ + \cdots + w_k^+)$, s_i is of small size, that is, smaller than or equal to $|r_1|$. However, if e_i is of the form $(w_1^* + \cdots + w_k^*)$ or $(w_1^+ + \cdots + w_k^+)$, then $s_i = w_j^\ell$, for some j, ℓ where $\ell \leq N_{r_1,r_2}$. So, the binary representation of ℓ has polynomial length in $|r_1| + |r_2|$. We therefore represent s_i by the pair (w_j, ℓ). In all other cases, $|s_i| \leq |e_i|$ and we represent s_i simply by $(s_i, 1)$. So, it suffices to guess for every e_i a pair (w_i, ℓ) where w_i occurs in e_i and $\ell \leq N_{r_1,r_2}$. According to Lemma 9.2, we can verify in polynomial time in the size of the compressed string s' that $s' \notin L(r_2)$. This concludes the proof of (c). □

Notice that, in the proof of Lemma 9.15, we actually did not make use of the restricted structure of the expression r_2. We can therefore state the following corollary:

Corollary 9.16. *Let r_1 be an RE$(\mathcal{S} - \{(+a)^*, (+w)^*, (+a)^+, (+w)^+\})$ expression and let r_2 be an arbitrary regular expression. Then, deciding whether $L(r_1) \subseteq L(r_2)$ is in co*NP.

Theorem 9.10(d) holds as the containment problem for arbitrary regular expressions is in PSPACE. This concludes the proof of Theorem 9.10.

9.4 Equivalence

In the present section, we merely initiate the research on the equivalence of chain regular expressions. Of course, upper bounds for INCLUSION imply upper bounds for EQUIVALENCE, but testing equivalence can be simpler. We show that the problem is in PTIME for RE($a, a?$) and RE(a, a^*, a^+) by showing that such expressions are equivalent if and only if they have a corresponding *sequence normal form* (defined below). We conjecture that EQUIVALENCE remains tractable for larger fragments, or even the full fragment of chain regular expressions. However, showing that EQUIVALENCE is in PTIME is already non-trivial for RE($a, a?$) and RE(a, a^*, a^+) expressions.

We now define the required normal form for RE($a, a?$) and RE(a, a^*, a^+) expressions. To this end, let $r = r_1 \cdots r_n$ be a chain regular expression with factors $r_1, \ldots, r_n$. The *sequence normal form* of r is obtained in the following way. First, we replace every factor of the form

- s by $s[1, 1]$;
- $s?$ by $s[0, 1]$;
- s^* by $s[0, *]$; and,
- s^+ by $s[1, *]$,

where s is an alphabet symbol. We call s the *base symbol* of the factor $s[i, j]$. Then, we replace successive subexpressions $s[i_1, j_1]$ and $s[i_2, j_2]$ with the same base symbol s by

- $s[i_1 + i_2, j_2 + j_2]$ when j_1 and j_2 are integers; and by
- $s[i_1 + i_2, *]$ when $j_1 = *$ or $j_2 = *$,

until no such replacements can be made anymore. For instance, the sequence normal form of $aa?aa?b^*bb?b^*$ is $a[2, 4]b[1, *]$. When r' is the sequence normal form of a chain regular expression, and $s[i, j]$ is a subexpression of r', then we call $e[i, j]$ a *factor of* r'.

Unfortunately, there are equivalent RE(a, a^*) expressions that do not share the same sequence normal form. For instance, the regular expressions

$$r_1(a, b) = a[i, *]b[0, *]a[0, *]b[1, *]a[l, *]$$

and

$$r_2(a, b) = a[i, *]b[1, *]a[0, *]b[0, *]a[l, *]$$

are equivalent but have different sequence normal forms. So, whenever an expression of the form $r_1(a, b)$ occurs, it can be replaced by $r_2(a, b)$. The *strong sequence normal form* of an expression r is the expression r' obtained by applying this rule as often as possible. It is easy to see that r' is unique.

We extend the notion of a *match* between a string and a regular expression in the obvious way to expressions in (strong) sequence normal form.

Theorem 9.17. EQUIVALENCE *is in* PTIME *for*

(1) RE($a, a?$), and

(2) RE(a, a^*, a^+).

Proof. We prove that, for both fragments, two expressions are equivalent only if they have the same strong sequence normal form.

We introduce some notions. If f is an expression of the form $e[i,j]$, we write $\text{base}(f)$ for e, $\text{upp}(f)$ for the upper bound j and $\text{low}(f)$ for the lower bound i. If $r = r_1 \cdots r_n$ is an expression in sequence normal form, we write $\max(r)$ for the maximum upper bound in r different from $*$, that is, for $\max\{\text{upp}(r_i) \mid \text{upp}(r_i) \neq *\}$. Finally, we call a substring v of a string w a *block* of w when w is of the form $a_1^{k_1} \cdots a_n^{k_n}$, where for each $i = 1, \ldots, n-1$, $a_i \neq a_{i+1}$ and v is of the form $a_i^{k_i}$ for some i.

In the following, we prove that if two expressions r and s from one of the two stated fragments are equivalent, their strong sequence normal forms r' and s' are equal. The proof is a case study which eliminates one by one all differences between the strong normal forms of the two equivalent expressions.

Therefore, let r and s be two equivalent expressions and let $r' = r_1 \cdots r_n$ and $s' = s_1 \cdots s_m$ be the strong sequence normal form of r and s, respectively. We assume that every $r_1, \ldots, r_n$ and $s_1, \ldots, s_m$ is of the form $e[i,j]$. Let $k := 1 + \max(\max(r'), \max(s'))$, that is, k is larger than any upper bound in r' and s' different from $*$.

We first show that $m = n$ and that, for every $i = 1, \ldots, n$, $\text{base}(r_i) = \text{base}(s_i)$. Thereto, let $v^{\max} = v_1^{\max} \cdots v_n^{\max}$, where, for every $i = 1, \ldots, n$,

$$v_i^{\max} = \begin{cases} \text{base}(r_i)^k & \text{if upp}(r_i) = *, \\ \text{base}(r_i)^{\text{upp}(r_i)} & \text{otherwise.} \end{cases}$$

Obviously, $v^{\max}$ is an element of $L(r)$. Hence, by our assumption that r is equivalent to s, we also have that $v^{\max} \in L(s)$. As $v^{\max}$ contains n blocks, s' must have at least n factors, so $m \geq n$. Correspondingly, we define the string $w^{\max} = w_1^{\max} \cdots w_m^{\max}$, where, for every $j = 1, \ldots, m$,

$$w_j^{\max} = \begin{cases} \text{base}(s_j)^k & \text{if upp}(s_j) = *, \\ \text{base}(s_j)^{\text{upp}(s_j)} & \text{otherwise.} \end{cases}$$

As $w^{\max}$ has to match r, we can conclude that r' has at least m factors, so $n \geq m$. Hence, we obtain that $m = n$. Furthermore, for each $i = 1, \ldots, n$, it follows immediately that $\text{base}(r_i) = \text{base}(s_i)$.

We now show that, for every $i = 1, \ldots, n$, $\text{upp}(r_i) = \text{upp}(s_i)$. If, for some i, $\text{upp}(r_i) = *$ then $v_i^{\max}$ has to be matched in s' by a factor with upper bound $*$. The analogous statement holds if $\text{upp}(s_i) = *$. Hence, $\text{upp}(r_i) = *$ if and only if $\text{upp}(s_i) = *$. Finally, we similarly get that $\text{upp}(r_i) = \text{upp}(s_i)$ for factors r_i, s_i with $\text{upp}(r_i) \neq *$ and $\text{upp}(s_i) \neq *$.

It only remains to show for each $i = 1, \ldots, n$, that we also have that $\text{low}(r_i) = \text{low}(s_i)$. By considering the string $v^{\min} = v_1^{\min} \cdots v_n^{\min}$, where each $v_i^{\min} = \text{base}(r_i)^{\text{low}(r_i)}$ and its counterpart $w^{\min} = w_1^{\min} \cdots w_n^{\min}$, where each $w_i^{\min} = \text{base}(s_i)^{\text{low}(s_i)}$, it is immediate that the sequence of non-zero lower bounds is the same in r' and s'.

For the sake of a contradiction, let us now assume that there exists an $i_{\min} \in \{1, \ldots, n\}$ with $\text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}})$ and $i_{\min}$ is minimal with this property. We consider two cases:

1) If $\text{low}(r_{i_{\min}}) > 0$, we define the string v' by replacing $v^{\max}_{i_{\min}}$ in the string $v^{\max}$ by $\text{base}(r_{i_{\min}})^{\text{low}(r_{i_{\min}})}$. As $\text{low}(s_{i_{\min}}) > \text{low}(r_{i_{\min}})$ this string can not be matched by s'. Indeed, as v' has n blocks, the only possible match for s' would match the $i_{\min}$-th block $\text{base}(r_{i_{\min}})^{\text{low}(r_{i_{\min}})}$ onto $s_{i_{\min}}$. As this would mean that r is not equivalent to s, this gives the desired contradiction.

2) In the second case, assume that $0 = \text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}})$. If $\text{low}(s_{i_{\min}}) \geq 2$, then the string resulting from $w^{\max}$ by replacing $w^{\max}_{i_{\min}}$ by the single symbol $\text{base}(r_{i_{\min}})$ matches r' but does not match s', which again contradicts that r is equivalent to s.

   The only remaining case is that $0 = \text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}}) = 1$.

   - If $i_{\min} = 1$, then the string $v^{\max}_2 \cdots v^{\max}_n$ does not match s', as the string starts with the wrong symbol. However, the string matches r', which is a contradiction.
   - If $i_{\min} = n$, then the string $v^{\max}_1 \cdots v^{\max}_{n-1}$ does not match s', as the string ends with the wrong symbol. However, the string matches r', which is a contradiction.

   Hence, we know that $1 < i_{\min} < n$.

   Let x be the string $v^{\max}_1 \cdots v^{\max}_{i_{\min}-1} v^{\max}_{i_{\min}+1} \cdots v^{\max}_n$ which matches r' and therefore also matches s'. If $\text{base}(r_{i_{\min}-1}) \neq \text{base}(r_{i_{\min}+1})$, then x has $n-1$ blocks. Recall that for every $j = 1, \ldots, n$, $\text{base}(r_j) = \text{base}(s_j)$. As $\text{base}(s_{i_{\min}}) \neq \text{base}(s_{i_{\min}+1})$ and $\text{base}(s_{i_{\min}}) \neq \text{base}(s_{i_{\min}-1})$, $s_{i_{\min}}$ can only match $v^{\max}_j$, for some $j > i_{\min} + 1$ or $j < i_{\min} - 1$. But then all the blocks before $v^{\max}_j$ or after $v^{\max}_j$ (which are at least $i_{\min}$ or $n - i_{\min} + 1$ blocks, respectively) must match $s_1 \cdots s_{i_{\min}-1}$ or $s_{i_{\min}+1} \cdots s_n$, respectively, which is impossible.

   We are left with the case where $\text{base}(r_{i_{\min}-1}) = \text{base}(r_{i_{\min}+1})$.

   (a) If r and s are from $\text{RE}(a, a?)$ then neither $s_{i_{\min}-1}$ nor $s_{i_{\min}+1}$ matches the string $v^{\max}_{i_{\min}-1} v^{\max}_{i_{\min}+1}$ as this string has length $\text{upp}(r_{i_{\min}-1}) + \text{upp}(r_{i_{\min}+1}) = \text{upp}(s_{i_{\min}-1}) + \text{upp}(s_{i_{\min}+1})$, which is more than $\max(\text{upp}(s_{i_{\min}-1}), \text{upp}(s_{i_{\min}+1}))$. As this would mean again that $s_{i_{\min}}$ can only match $v^{\max}_j$, for some $j > i_{\min}+1$ or $j < i_{\min} - 1$, we again have that s can not match x, a contradiction.

   (b) Now let r and s be from $\text{RE}(a, a^*)$. Let $j_{\min} > i_{\min}$ be minimal such that $\text{low}(r_{j_{\min}}) \neq 0$. As we already obtained that the sequence of non-zero lower bounds is the same in r' and s', such a $j_{\min}$ must exist and $\text{base}(r_{j_{\min}}) = \text{base}(r_{i_{\min}}) = \text{base}(s_{i_{\min}})$. Let b be the symbol $\text{base}(r_{j_{\min}}) = \text{base}(r_{i_{\min}})$ and let a be the symbol $\text{base}(r_{i_{\min}-1}) = \text{base}(r_{i_{\min}+1})$. Notice that $a \neq b$.

   Our goal is to get a contradiction by showing that r' is not the strong sequence normal form of r. On our way we have to deal with a couple of other possible cases.

   Assume that there is some ℓ, $i_{\min} < \ell < j_{\min}$, with $a \neq \text{base}(r_\ell) \neq b$. Let $\ell' < i_{\min}$ be maximal such that $a \neq \text{base}(r_{\ell'}) \neq b$. If there is no such ℓ', we

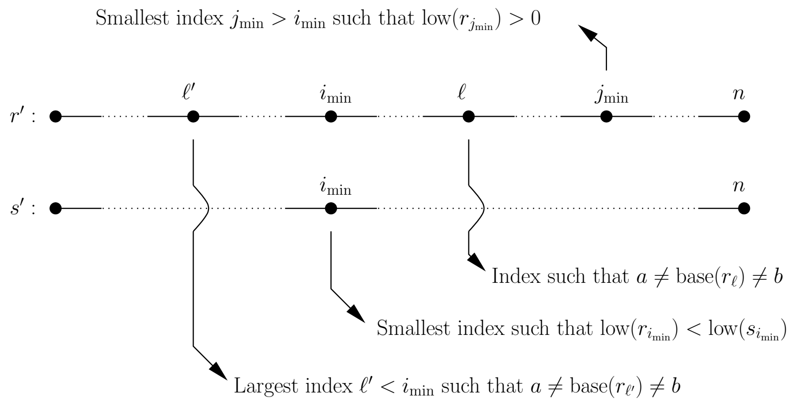

Smallest index $j_{\min} > i_{\min}$ such that $\text{low}(r_{j_{\min}}) > 0$
ℓ'
$i_{\min}$
ℓ
$j_{\min}$
n
r' :
$i_{\min}$
n
s' :
Index such that $a \neq \text{base}(r_\ell) \neq b$
Smallest index such that $\text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}})$
Largest index $\ell' < i_{\min}$ such that $a \neq \text{base}(r_{\ell'}) \neq b$

Figure 9.2: Graphical representation of several indices defined in the proof of Theorem 9.17

set $\ell' = 0$. We present the ordering of the defined indices $i_{\min}, j_{\min}, \ell'$, and ℓ in Figure 9.2 Let, for each $d = 1, \ldots, n$, x'_d be defined by

$$x'_d = \begin{cases} \text{base}(r_d)^1 & \text{if } \text{low}(r_d) = 0, \\ \text{base}(r_d)^{\text{low}(r_d)} & \text{otherwise.} \end{cases}$$

Let $x' = x'_1 \cdots x'_{\ell'} v^{\min}_{\ell'+1} \cdots v^{\min}_{\ell-1} x'_\ell \cdots x'_n$. Notice that the string $v^{\min}_{\ell'+1} \cdots v^{\min}_{\ell-1}$ is non-empty. As x' matches r, x' also matches s. Note that, in this match, $x'_{\ell'}$ has to be matched by $s_{\ell'}$, because the string $x'_1 \cdots x'_{\ell'}$ contains ℓ' blocks. Analogously, x'_ℓ has to be matched by s_ℓ because $x'_\ell \cdots x'_n$ contains $n - \ell + 1$ blocks.

As $i_{\min}$ is minimal such that $\text{low}(r_{i_{\min}}) < \text{low}(s_{i_{\min}})$ and as there are no factors f in $r_{i_{\min}} \cdots r_\ell$ with $\text{low}(f) > 0$, we know that the number of factors f in $s_{\ell'+1} \cdots s_{\ell-1}$ with $\text{base}(f) = b$ and $\text{low}(f) > 0$ is larger than the number of such factors in $r_{\ell'+1} \cdots r_{\ell-1}$. Hence, the number of b's in x' between $x'_{\ell'}$ and x'_ℓ is smaller than the sum of the numbers $\text{low}(f)$ over the factors f in $s_{\ell'+1} \cdots s_{\ell-1}$ with $\text{base}(f) = b$. This contradicts the fact that x' matches s.

We can conclude that there is no such ℓ, that is, all factors between position $i_{\min}$ and $j_{\min}$ have symbol a or b.

Let us consider next the possibility that the symbol $\text{base}(r_{j_{\min}+1})$ exists and is different from a and b, say $\text{base}(r_{j_{\min}+1}) = c$. If $\text{low}(s_{j_{\min}}) = 0$ then the string $x'_1 \cdots x'_{j_{\min}-1} x'_{j_{\min}+1} \cdots x'_n$ (consisting of $n - 1$ blocks, as c is different from a or b) matches s but not r. This contradicts that r and s are equivalent. Let ℓ' be defined as before. If $\text{low}(s_{j_{\min}}) = 1$ then the string $x'_1 \cdots x'_{\ell'} v^{\min}_{\ell'+1} \cdots v^{\min}_{j_{\min}} x'_{j_{\min}+1} \cdots x'_n$ matches r but not s (as it has too few bs between ℓ' and $j_{\min}$). An analogous reasoning also works if $j_{\min} = n$.

We still need to deal with the case where $\text{base}(r_{j_{\min}+1}) = a$. Hence, between position $i_{\min} - 1$ and $j_{\min} + 1$, r' consists of a sequence of factors f which alternate between $\text{base}(f) = a$ and $\text{base}(f) = b$, and ends with the factor

$r_{j_{\min}+1}$ for which $\text{base}(r_{j_{\min}+1}) = a$. Furthermore, all these factors, besides $r_{j_{\min}}$ and $r_{j_{\min}+1}$ have $\text{low}(f) = 0$, and $\text{low}(r_{j_{\min}}) = 1$. It follows immediately that r' is not the sequence normal form of r as it contains a subsequence of the form $a[i,*]b[0,*]a[0,*]b[1,*]a[l,*]$. □

9.5 Intersection

For arbitrary regular expressions, INTERSECTION NON-EMPTINESS is PSPACE-complete. We show that the problem is NP-hard for the same seemingly innocent fragments RE(a, a^*), RE$(a, a?)$, RE$(a, (+a^+))$ and RE$(a^+, (+a))$ already studied in Section 9.3. By RE$(\mathcal{S} - \{(+w)^*, (+w)^+\})$, we denote the fragment of RE$(\mathcal{S})$ where no factor can be of the form $(w_1 + \cdots + w_n)^*$ or $(w_1 + \cdots + w_n)^+$ with $n \geq 2$ and the length of at least one w_i larger than two; so, factors of the form $(a_1 + \cdots + a_n)^*$ and $(a_1 + \cdots + a_n)^+$ are allowed. For the latter fragment, we obtain a matching NP-upper bound. INTERSECTION NON-EMPTINESS is already PSPACE-hard for RE$(a, (+w)^*)$ and RE$(a, (+w)^+)$ expressions. This follows from a proof of Bala, who showed that it is PSPACE-hard to decide whether the intersection of an arbitrary number of RE$((+w)^*)$ expressions contains a *non-empty* string [Bal02]. The precise complexity of RE(a, w^+) remains open. These results are summarized in the following theorem:

Theorem 9.18. *(a)* INTERSECTION NON-EMPTINESS *is* NP*-hard for*

(1) RE(a, a^*);

(2) RE$(a, a?)$;

(3) RE$(a, (+a^+))$;

(4) RE$(a, (+a)^+)$; and

(5) RE$(a^+, (+a))$;

(b) INTERSECTION NON-EMPTINESS *is in* NP *for RE$(\mathcal{S} - \{(+w)^*, (+w)^+\})$;*

(c) INTERSECTION NON-EMPTINESS *is* PSPACE*-hard for*

(1) RE$(a, (+w)^*)$; and

(2) RE$(a, (+w)^*)$; and,

(d) INTERSECTION NON-EMPTINESS *is in* PSPACE *for RE$(\mathcal{S})$.*

As in Section 9.3, we split the proof of Theorem 9.18 into a series of lemmas to improve readability.

Lemma 9.19 (Theorem 9.18(a))**.** INTERSECTION NON-EMPTINESS *is* NP*-hard for*

(1) RE(a, a^*);

(2) RE$(a, a?)$;

(3) RE$(a, (+a^+))$;

(4) RE$(a, (+a)^+)$; and

(5) RE$(a^+, (+a))$.

Proof. The proof is along the same lines as the proof of Lemma 9.12. In all five cases, it is a LOGSPACE reduction from SATISFIABILITY of propositional 3CNF formulas. The SATISFIABILITY problem asks, given a propositional formula Φ in 3CNF with variables $\{x_1, \ldots, x_n\}$, whether there exists a truth assignment of $\{x_1, \ldots, x_n\}$ for which Φ is true. The SATISFIABILITY problem for 3CNF formulas is known to be NP-complete [GJ79]. We note that, for the cases (1)–(3), INTERSECTION NON-EMPTINESS is already NP hard when the expressions use a fixed-size alphabet.

Let $\Phi = C_1 \wedge \cdots \wedge C_k$ be a 3CNF formula using variables $\{x_1, \ldots, x_n\}$. Let, for each i, $C_i = L_{i,1} \vee L_{i,2} \vee L_{i,3}$ be the ith clause with three literals. Our goal is to construct regular expressions $R_1, \ldots, R_k$ and S_1, S_2 such that

$$L(R_1) \cap \cdots \cap L(R_k) \cap L(S_1) \cap L(S_2) \neq \emptyset \text{ if and only if } \Phi \text{ is satisfiable.}$$

Analogously as in the proof of Lemma 9.12, we encode truth assignments for Φ by strings. We construct S_1 and S_2 such that $L(S_1 \cap S_2)$ contains all such string representations w of truth assignments and a string w matches R_i if and only if w represents an assignment which makes C_i true.

We discuss the building blocks of these regular expressions. We make use of a regular expression U describing exactly one string u: U will either be of the form a^n or of the form $\#a^i\$\cdots\$a^i\#$ (n occurrences of a^i separated by \$) for some integer i. We define expressions W_1 and W_2 such that each string w in $L(W_1) \cap L(W_2)$ can be interpreted as a truth assignment. More precisely, for each truth assignment A there is a string $w_A \in L(W_1) \cap L(W_2)$ and for each string $w \in L(W_1) \cap L(W_2)$ there is a corresponding truth assignment A_w.

We set

$$\begin{aligned} S_1 &= U^3 W_1 U^3, \\ S_2 &= U^3 W_2 U^3, \text{ and} \\ R_i &= N F_{i,1} F_{i,2} F_{i,3} N, \end{aligned}$$

for $i = 1, \ldots, k$, where N and $F_{i,1}, \ldots, F_{i,3}$ are regular expressions for which the following properties hold:

(i') u, u^2, and $u^3 \in L(N)$.

(ii') If A makes $L_{i,j}$ true then $w_A \in L(F_{i,j}) \cap L(W_1) \cap L(W_2)$. If $w \in L(F_{i,j}) \cap L(W_1) \cap L(W_2)$ then A_w makes $L_{i,j}$ true.

(iii') $u \in L(F_{i,j})$ for every $i = 1, \ldots, k$ and $j = 1, 2, 3$.

(iv') If $u^3 w u^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$, then w matches some F_{i,j_i} in every R_i.

We claim the following:

Claim 9.20. *If there are expressions* U, W_1, W_2, N, $F_{1,1}$, $F_{1,2}$, $F_{1,3}$, $F_{2,1}, \ldots, F_{k,1}$, $F_{k,2}$, $F_{k,3}$ *satisfying (i')–(iv') above, then*

$$L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k) \neq \emptyset \textit{ if and only if } \Phi \textit{ is satisfiable.}$$

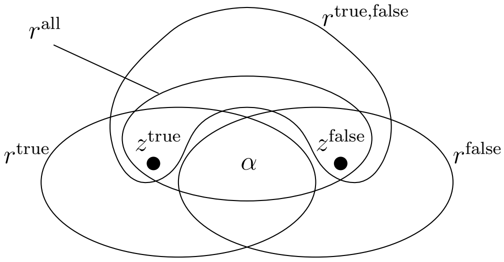


Figure 9.3: Inclusion structure of regular expressions used in NP-hardness of INTERSECTION NON-EMPTINESS.

Proof. Suppose there are expressions U, W_1, W_2, N, $F_{1,1}$, $F_{1,2}$, $F_{1,3}$, $F_{2,1}, \ldots, F_{k-1,3}$, $F_{k,1}$, $F_{k,2}$, $F_{k,3}$ satisfying (i')–(iv'). We prove that $L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k) \neq \emptyset$ if and only if Φ is satisfiable.

($\Rightarrow$) Assume that $S_1 \cap S_2 \cap \bigcap_{i=1}^{k} R_i \neq \emptyset$. Hence, there exists a string $v = u^3 w u^3$ in $S_1 \cap S_2 \cap \bigcap_{i=1}^{k} R_i$, where u is the unique string in $L(U)$. By (iv'), w matches some F_{i,j_i} in every R_i. By (ii'), A_w makes L_{i,j_i} true for every $i = 1, \ldots, k$. Hence, Φ is true under truth assignment A_w, so Φ is satisfiable.

($\Leftarrow$) Suppose now that Φ is true under some truth assignment A. Hence, for every i, some L_{i,j_i} becomes true under A and therefore $w_A \in L(F_{i,j_i})$ by (ii'). As u, u^2, and u^3 are in $L(N)$ by (i') and as u is in each $L(F_{i,j})$ by (iii'), we get that the string $u^3 w_A u^3$ is in $L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$. This completes the proof of the claim. □

It remains to construct the regular expressions with the required properties. As in the proof of Lemma 9.12, we construct these expressions starting from five basic regular expressions r^{true}, r^{false}, $r^{\text{true,false}}$, r^{all}, and α, which must adhere to the inclusion structure as shown in Figure 9.3 and formally defined by the properties (INT1)–(INT4) below. The two dots in Figure 9.3 denote the strings z^{true} and z^{false}, which represent the truth values *true* and *false*, respectively. The expressions r^{true} and r^{false} are used to match z^{true} and z^{false}, respectively, in the expressions $R_1, \ldots, R_k$. The expression $r^{\text{true,false}}$ is used to generate the truth values *true* and *false*. It will be defined as the intersection of two expressions in S_1 and S_2 and will be used to generate all truth assignments which must then be matched in $R_1, \ldots, R_k$. Finally, α and r^{all} are used to ensure that condition (iv') above holds. That is, they ensure that when $u^3 w u^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$, w matches some F_{i,j_i} in every R_i.

In all cases, these expressions have the properties INT1–INT4:

$$\alpha \in L(r^{\text{false}}) \cap L(r^{\text{true}}) \cap L(r^{\text{all}}) \tag{INT1}$$

$$z^{\text{false}} \in L(r^{\text{false}}) \cap L(r^{\text{true,false}}) \cap L(r^{\text{all}}) \tag{INT2}$$

$$z^{\text{true}} \in L(r^{\text{true}}) \cap L(r^{\text{true,false}}) \cap L(r^{\text{all}}) \tag{INT3}$$

$$L(r^{\text{false}}) \cap L(r^{\text{true}}) \cap L(r^{\text{true,false}}) = \emptyset \tag{INT4}$$

Note that $z^{\text{false}} \notin L(r^{\text{true}})$ and $z^{\text{true}} \notin L(r^{\text{false}})$ by (INT4). For the first three fragments, we now define the needed expressions. We deal with the other two fragments later. Note that the alphabet size is fixed (at most five) in the reduction for these fragments.

(1) For RE(a, a^*):

- $\alpha = a$;
- $r^{\text{true}} = aa^*b^*$;
- $r^{\text{false}} = b^*aa^*$;
- $r^{\text{true,false}} = aa^*bb^* + bb^*aa^*$, which is constructed from the intersection of $s^{\text{true,false}} = b^*aa^*b^*$ and $(s')^{\text{true,false}} = a^*bb^*a^*$;
- $r^{\text{all}} = a^*b^*a^*$;
- $z^{\text{true}} = ab$;
- $z^{\text{false}} = ba$;
- $U = \#\alpha\$\cdots\$\alpha\#$ (n occurrences of α);
- $W_1 = \#s^{\text{true,false}}\$\cdots\$s^{\text{true,false}}\#$;
- $W_2 = \#(s')^{\text{true,false}}\$\cdots\$(s')^{\text{true,false}}\#$;
- $N = (\#^*a^*\$^*a^*\$^*\cdots\$^*a^*\#^*)^3$ (n occurrences of a^* in each of the three copies).

(2) For RE$(a, a?)$:

- $\alpha = a$;
- $r^{\text{true}} = ab?$;
- $r^{\text{false}} = b?a$;
- $r^{\text{true,false}} = ab + ab$, which is constructed from the intersection of $s^{\text{true,false}} = b?ab?$ and $(s')^{\text{true,false}} = a?ba?$;
- $r^{\text{all}} = a?b?a?$;
- $z^{\text{true}} = ab$;
- $z^{\text{false}} = ba$;
- $U = \#\alpha\$\cdots\$\alpha\#$ (n occurrences of α)
- $W_1 = \#s^{\text{true,false}}\$\cdots\$s^{\text{true,false}}\#$;
- $W_2 = \#(s')^{\text{true,false}}\$\cdots\$(s')^{\text{true,false}}\#$;
- $N = (\#?a?\$?a?\$?\cdots\$?a?\#?)^3$ (n occurrences of $a?$ in each of the three copies).

(3) For RE$(a, (+a^+))$:

- $\alpha = a$;
- $r^{\text{true}} = (a + b)^+$;
- $r^{\text{false}} = (a + c)^+$;

- $r^{\text{true,false}} = (b+c)^+$;
- $r^{\text{all}} = (a+b+c)^+$;
- $z^{\text{true}} = b$;
- $z^{\text{false}} = c$;
- $U = \#\alpha\$\alpha\$\cdots\$\alpha\#$ (n occurrences of α);
- $W_1 = W_2 = \#r^{\text{true,false}}\$\cdots\$r^{\text{true,false}}\#$; and
- $N = (\# + \$ + a)^+$

It is easy to check that the conditions (INT1)–(INT4) are fulfilled for all of the fragments.

With $w = \#w_1\$\cdots\$w_n\# \in L(W_1) \cap L(W_2)$ we associate a truth assignment A_w as follows

$$A_w(x_j) := \begin{cases} \text{true}, & \text{if } w_j \in L(r^{\text{true}}), \\ \text{false}, & \text{otherwise.} \end{cases}$$

Let $z^{\text{false}} \in L(r^{\text{false}}) \cap L(r^{\text{all}}) \cap L(r^{\text{true,false}})$ and $z^{\text{true}} \in L(r^{\text{true}}) \cap L(r^{\text{all}}) \cap L(r^{\text{true,false}})$. By (INT2) and (INT3) we know that these strings exist. Notice that $z^{\text{false}} \notin L(r^{\text{true}})$ and $z^{\text{true}} \notin L(r^{\text{false}})$ by (INT4). For a truth assignment A, let

$$w_A = \#w_1\$\cdots\$w_n\#,$$

where $w_j = z^{\text{true}}$ if $A(x_j) = \text{true}$ and $w_j = z^{\text{false}}$, otherwise.

For each i, j, we set

$$F_{i,j} = \#e_1\$\cdots\$e_n\#,$$

where for each $\ell = 1, \ldots, n$,

$$e_\ell := \begin{cases} r^{\text{false}}, & \text{if } L_{i,j} = \neg x_\ell, \\ r^{\text{true}}, & \text{if } L_{i,j} = x_\ell,\text{, and} \\ r^{\text{all}}, & \text{otherwise.} \end{cases}$$

Notice that only one e_ℓ among $\{e_1, \ldots, e_n\}$ is different from r^{all}.

It remains to show that, for each of the fragments, conditions (i')–(iv') hold.

(i') Trivial.

(ii') Let A be a truth assignment such that $L_{i,j}$ is true under A. We need to show that $w_A \in L(F_{i,j}) \cap L(W_1) \cap L(W_2)$. Suppose that $L_{i,j} = x_\ell$, so, the variable x_ℓ occurs positively in clause C_i and $A(x_\ell) = \text{true}$. Let $F_{i,j} = \#e_1\$\cdots\$e_n\#$ be as defined above. By definition of $F_{i,j}$, we have that $e_\ell = r_\ell^{\text{true}}$ and for all $\ell' \neq \ell$ we have that $e_{\ell'} = r_k^{\text{all}}$. Let $w_A = \#w_1\$\cdots\$w_n\#$ be as defined above, so $w_\ell = z_\ell^{\text{true}}$ by definition of w_A. Notice that, for every $\ell' \neq \ell$, $w_{\ell'}$ is either $z_{\ell'}^{\text{false}}$ or $z_{\ell'}^{\text{true}}$. Since $z_\ell^{\text{true}} \in L(r_\ell^{\text{true}}) \cap L(r_\ell^{\text{all}})$ and $\{z_\ell^{\text{true}}, z_\ell^{\text{false}}\} \subseteq L(r_\ell^{\text{all}})$, we have that $w_A \in L(F_{i,j})$. Moreover, as (INT2) and (INT3) state that $\{z_{\ell'}^{\text{true}}, z_{\ell'}^{\text{false}}\} \subseteq L(r^{\text{true,false}})$, we also have that $w_A \in L(W_1) \cap L(W_2)$. The dual statement also holds if $L_{i,j} = \neg x_\ell$.

For the second claim, let $w = \#w_1\$\cdots\$w_n\#$ be a string in $L(F_{i,j}) \cap L(W_1) \cap L(W_2)$, where $F_{i,j} = \#e_1\$\cdots\$e_n\#$. We now need to show that $L_{i,j}$ is true under truth assignment A_w. If $L_{i,j} = x_\ell$, then we have that $e_\ell = r_\ell^{\text{true}}$ by definition of $F_{i,j}$. As w matches $F_{i,j}$, we have that $w_\ell \in L(r_\ell^{\text{true}})$. Hence, by definition of A_w, $A_w(x_\ell) = \text{true}$, so $L_{i,j}$ is true under A_w. If $L_{i,j} = \neg x_\ell$, then we have that $e_\ell = r_\ell^{\text{false}}$. This means that $w_\ell \in r_\ell^{\text{true,false}} \cap r_\ell^{\text{false}}$, as w is also in $L(W_1) \cap L(W_2)$. Consequently, $w_\ell \notin r_\ell^{\text{true}}$ by (INT4). By definition of A_w, we now have that $A_w(x_\ell) = \text{false}$, and A_w again makes $L_{i,j}$ true.

(iii') This follows immediately from condition (INT1).

(iv') Suppose that $u^3wu^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$. We need to show that w matches some F_{i,j_i} in every R_i. Observe that the string u^3wu^3 is of the form $\#y_1\#\#y_2\#\#y_3\#\#y_4\#\#y_5\#\#y_6\#\#y_7\#$, where $y_1, \ldots, y_7$ are non-empty strings over the alphabet $\{a, b, c, \$\}$. Moreover, every string in every $L(F_{i,j})$ is of the form $\#y\#$ where y is a non-empty string over the alphabet $\{a, b, c, \$\}$. Hence, as for every $i = 1, \ldots, k$, $u^3wu^3 \in L(R_i)$, and as none of the strings $y, y_1, \ldots, y_7$ contain the symbol "#", w either matches some $F_{i,j}$ or w matches a sub-expression of N.

Fix an $i = 1, \ldots, k$. Let m be a match between u^3wu^3 and R_i. Let ℓ be the number so that m matches $\#y_1\#\cdots\#y_\ell\#$ onto the left occurrence of N in R_i. We now distinguish between fragments (1–2) and fragment (3).

- In fragments (1–2) we have that $\ell \leq 3$. Towards a contradiction, assume that m matches w onto the left occurrence of N in R_i. But this would mean that m matches a superstring of $\#y_1\#\cdots\#y_4\#$ onto this leftmost occurrence of N, which contradicts that $\ell \leq 3$. Analogously, m cannot match w onto the right occurrence of N in R_i. So, w must match by some $F_{i,j}$ for $j = 1, 2, 3$.
- In fragment (3), we have that $\ell \geq 1$. Again, towards a contradiction, assume that m matches w onto the right occurrence of N in R_i. Observe that any string that matches $NF_{i,1}F_{i,2}F_{i,3}$ is of the form $\#y'_1\#\#y'_2\#\cdots\#y'_{\ell'}\#$, where $\ell' > 3$. As u^3 is not of this form, m cannot match u^3 onto $NF_{i,1}F_{i,2}F_{i,3}$, which is a contradiction. Analogously, m cannot match w against the left occurrence of N in R_i. So, m must match w onto some $F_{i,j}$ for $j = 1, 2, 3$.

This completes the proof of Lemma 9.19 for the fragments (1)–(3).

We still need to deal with the fragments (4) and (5). As in the proof of Lemma 9.12, we will no longer use an alphabet with fixed size. Instead, we use the symbols b_j and c_j for $j = 1, \ldots, n$. Instead of the basic regular expressions r^{true}, r^{false}, $r^{\text{true,false}}$, and r^{all}, we will now have expressions r_j^{true}, r_j^{false}, $r_j^{\text{true,false}}$, and r_j^{all} for every $j = 1, \ldots, n$. We will require that these expressions have the same properties (INT1)–(INT4), but only between the expressions r_j^{true}, r_j^{false}, $r_j^{\text{true,false}}$, and r_j^{all} with the same index j, and α.

The needed regular expressions are then defined as follows:

(4) For RE$(a, (+a)^+)$:

- $\alpha = a$;
- $r_j^{\text{true}} = (a^+ + b_j^+)$;
- $r_j^{\text{false}} = (a^+ + c_j^+)$;
- $r_j^{\text{true,false}} = (b_j^+ + c_j^+)$;
- $r_j^{\text{all}} = (a^+ + b_j^+ + c_j^+)$;
- $z^{\text{true}} = b_j$;
- $z^{\text{false}} = c_j$;
- $U = \alpha^n$;
- $W_1 = W_2 = r_1^{\text{true,false}} \cdots r_n^{\text{true,false}}$; and
- $N = a^+$

(5) For RE$(a^+, (+a))$:

- $\alpha = a$;
- $r_j^{\text{true}} = (a + b_j)$;
- $r_j^{\text{false}} = (a + c_j)$;
- $r_j^{\text{true,false}} = (b_j + c_j)$;
- $r_j^{\text{all}} = (a + b_j + c_j)$;
- $z^{\text{true}} = b_j$;
- $z^{\text{false}} = c_j$;
- $U = \alpha^n$;
- $W_1 = W_2 = r_1^{\text{true,false}} \cdots r_n^{\text{true,false}}$; and
- $N = a^+$

With $w = \#w_1\$ \cdots \$w_n\# \in L(W_1) \cap L(W_2)$ we associate a truth assignment A_w as follows

$$A_w(x_j) := \begin{cases} \text{true}, & \text{if } w_j \in L(r_j^{\text{true}}), \\ \text{false}, & \text{otherwise.} \end{cases}$$

Let $z_j^{\text{false}} \in L(r_j^{\text{false}}) \cap L(r_j^{\text{all}}) \cap L(r_j^{\text{true,false}})$ and $z_j^{\text{true}} \in L(r_j^{\text{true}}) \cap L(r_j^{\text{all}}) \cap L(r_j^{\text{true,false}})$. By (INT2) and (INT3) we know that these strings exist. Notice that $z_j^{\text{false}} \notin L(r_j^{\text{true}})$ and $z_j^{\text{true}} \notin L(r_j^{\text{false}})$ by (INT4). For a truth assignment A, let

$$w_A = \#w_1\$ \cdots \$w_n\#,$$

where $w_j = z_j^{\text{true}}$ if $A(x_j) = \text{true}$ and $w_j = z_j^{\text{false}}$, otherwise.

For each i, j, we set

$$F_{i,j} = \#e_1\$ \cdots \$e_n\#,$$

where for each $\ell = 1, \ldots, n$,

$$e_\ell := \begin{cases} r_\ell^{\text{false}}, & \text{if } L_{i,j} = \neg x_\ell, \\ r_\ell^{\text{true}}, & \text{if } L_{i,j} = x_\ell,, \text{ and} \\ r_\ell^{\text{all}}, & \text{otherwise.} \end{cases}$$

It remains to show that, for each of the fragments, conditions (i')–(iv') hold.

(i') Trivial.

(ii') This can be shown analogously as for the fragments (1)–(3).

(iii') This follows immediately from condition (INT1).

(iv') Suppose that $u^3wu^3 \in L(S_1) \cap L(S_2) \cap L(R_1) \cap \cdots \cap L(R_k)$. We need to show that w matches some F_{i,j_i} in every R_i. To this end, let m_i be a match between u^kwu^k and R_i, for every $i = 1, \ldots, k$. For every $j = 1, \ldots, n$, let Σ_j denote the set $\{b_j, c_j\}$. Observe that the string w is of the form $y_1 \cdots y_n$, where, for every $j = 1, \ldots, n$, y_j is a string in Σ_j^+. Moreover, no strings in $L(N)$ contain symbols from Σ_j for any $j = 1, \ldots, n$. Hence, m_i cannot match any symbol of the string w onto N. Consequently, m_i matches the entire string w onto a subexpression of $F_{i,1}F_{i,2}F_{i,3}$ in every R_i.

Further, observe that every string in every $F_{i,j}$, $j = 1, 2, 3$, is of the form $y'_1 \ldots y'_n$, where each y'_j is a string in $(\Sigma_j \cup \{a\})^+$. As m_i can only match symbols in Σ_j onto subexpressions with symbols in Σ_j, m_i matches w onto some $F_{i,j}$.

This completes the proof of Lemma 9.19. □

The following Lemma makes the difference between INCLUSION and INTERSECTION NON-EMPTINESS apparent. Indeed, INCLUSION for RE($\mathcal{S} - \{(+w)^*, (+w)^+\}$) expressions is PSPACE-complete, while INTERSECTION NON-EMPTINESS for such expressions is NP-complete. The latter, however, does not imply that INCLUSION is always harder than INTERSECTION NON-EMPTINESS. Indeed, we obtained that for any fixed k, INCLUSION for RE$^{\leq k}$ is in PTIME. Later in this section, we will show that INTERSECTION NON-EMPTINESS is PSPACE-hard for RE$^{\leq 3}$ expressions (Theorem 9.23).

Lemma 9.21 (Theorem 9.18(b)). INTERSECTION NON-EMPTINESS *is in* NP *for RE*($\mathcal{S} - \{(+w)^*, (+w)^+\}$).

Proof. Let $r_1, \ldots, r_k$ be RE($\mathcal{S} - \{(+w)^*, (+w)^+\}$) expressions. We prove that if $\bigcap_{i=1}^k L(r_i) \neq \emptyset$, then the shortest string w in $\bigcap_{i=1}^k L(r_i)$ has a representation as a compressed string of polynomial size. The idea is that the factors of the expressions r_i induce a partition of w into at most kn substrings, where $n = \max\{|r_i| \mid 1 \leq i \leq k\}$. We show that each such substring is either short or it is matched by an expression of the form w^* or w^+. In the latter case, this substring can be written as $(x, 1)(w, i)(y, 1)$, for a suitable i, where x and y are a suffix and prefix of w, respectively. This immediately implies the statement of the theorem, as guessing v and verifying that string(v) is in each $L(r_i)$ is an NP algorithm by Lemma 9.2.

For simplicity, we assume that all r_i have the same number of factors, say n'. Otherwise, some expressions can be padded by additional factors ε. Let, for each i, $r_i = e_{i,1} \cdots e_{i,n'}$ where every $e_{i,j}$ is a factor.

Let $u = a_1 \cdots a_{\min}$ be a minimal string in $\bigcap_{i=1}^{k} r_i$. We will show that there is a polynomial size compressed string v such that string(v)=u. As the straightforward nondeterministic product automaton for $\bigcap_{i=1}^{k} r_i$ has at most n^k states, $|u| \leq n^k$.

Let, for each $i = 1, \ldots, k$, m_i be a match between u and r_i. Notice that, for each $i = 1, \ldots, k$ and $j = 1, \ldots, n'$, there is exactly one pair (ℓ, ℓ') such that $e_{i,j} \in m_i(\ell, \ell')$. We call an interval (p, p') of positions *homogeneous*, if, for each i, there are ℓ, ℓ' and j such that $\ell \leq p$, $p' \leq \ell'$ and $e_{i,j} \in m_i(\ell, \ell')$. Intuitively, an interval is homogeneous if, for each i, all its symbols are subsumed by the same subexpression $e_{i,j}$. We call an interval (p, p') *maximally homogeneous*, if (p, p') is homogeneous, but $(p, p' + 1)$ and $(p - 1, p')$ are not homogeneous.

Let $\ell_0, \ldots, \ell_{\text{maxpos}}$ be a non-decreasing sequence of positions of u such that $\ell_0 = 0, \ell_{\text{maxpos}} = \min$, and each pair $(\ell_p + 1, \ell_{p+1})$ is maximally homogeneous. Notice that $\ell_0, \ldots, \ell_{\text{maxpos}}$ maximally contains kn' positions.

Let, for each $p = 1, \ldots, \min$, u_p denote the substring of u from position $\ell_{p-1} + 1$ until position ℓ_p. We consider each substring u_p separately and distinguish the following cases:

- If u_p is contained in an interval which at least one m_i matches onto an expression of the form e or $e?$ for a disjunction of base symbols e which is (in abbreviated notation) of the form a, w, $(+a)$, or $(+w)$, then $|u_p| \leq |e| \leq n$. We set $v_p = u_p$.

- If u_p is only contained in intervals which are mapped to expressions of the form $(+a)^*$ or $(+a)^+$ (in abbreviated notation) then u_p has length zero or one. Otherwise, deleting a symbol of u_p in u would result in a shorter string u' which is still in every $L(r_p)$, which contradicts that u is a minimal string in $\bigcap_{i=1}^{k} L(r_i)$.

- The only remaining case is that u_p is contained in some interval which at least one m_i matches onto an expression of the form a^*, w^*, a^+, w^+, $(+a^*)$, $(+w^*)$, $(+a^+)$ or $(+w^+)$. Hence, u_p can be written as xw^iy for some i, a postfix x of w and a prefix y of w. Notice that the length of x and y is at most n. We define $v_p = (x, 1)(w, i)(y, 1)$. Of course, $i \leq n^k$, hence $|v_p| = \mathcal{O}(n)$.

Finally, let $v = v_1 \cdots v_{\min}$. Hence, v is a compressed string with string(v)= u and $|v| = \mathcal{O}(kn' \cdot n)$, as required. □

Bala has shown that deciding whether the intersection of RE($(+w)^*$) expressions contains a non-empty string is PSPACE-complete [Bal02]. In general, the intersection non-emptiness problem for such expressions is trivial, as the empty string is always in the intersection. We next present a much simpler version of the proof of Bala (which is a direct reduction from acceptance by a Turing machine that uses polynomial space), adapted to show PSPACE-hardness of INTERSECTION NON-EMPTINESS for RE($a, (+w)^*$) and RE($a, (+w)^*$) expressions. It should be noted that the crucial idea in this proof comes from the proof of Bala [Bal02].

Lemma 9.22 (Theorem 9.18(c), see also [Bal02]). INTERSECTION NON-EMPTINESS *is* PSPACE-*hard for*

(1) RE$(a, (+w)^*)$ *and*

(2) RE$(a, (+w)^+)$.

Proof. We first show that INTERSECTION NON-EMPTINESS is PSPACE-hard for the class RE$(a, (+w)^+)$ and we consider the case of RE$(a, (+w)^*)$ later. In both cases, we use a reduction from CORRIDOR TILING, which is PSPACE-complete (Theorem 3.23).

To this end, let $D = (T, H, V, \overline{b}, \overline{t}, n)$ be a tiling system. Without loss of generality, we assume that $n \geq 2$ is an even number. We construct $n+3$ regular expressions BT, Hor_{even}, Hor_{odd}, $\text{Ver}_1, \dots, \text{Ver}_n$ such that

$$L(\text{BT}) \cap L(\text{Hor}_{\text{even}}) \cap L(\text{Hor}_{\text{odd}}) \cap \bigcap_{j=1}^{n} L(\text{Ver}_j) \neq \emptyset$$

if and only if there exists a correct corridor tiling for D.

Let $T = \{t_1, \dots, t_k\}$ be the set of tiles of D. In our regular expressions, we will use a different alphabet for every column of the tiling. To this end, for every $j = 1, \dots, n$, we define $\Sigma_j := \{t_{i,j} \mid t_i \in T\}$, which we will use to tile the j-th column. We define $\Sigma := \bigcup_{1 \leq j \leq n} \Sigma_j$. For a set of Σ-symbols S, we denote by S the disjunction of all the symbols of S, whenever this improves readability.

We represent possible tilings of D as strings in the language defined by the regular expression

$$\rhd(\triangle^n \Sigma_1 \triangle^n \Sigma_2 \triangle^n \cdots \triangle^n \Sigma_n \triangle^n \#)^* \lhd, \qquad (\star)$$

where $\rhd, \lhd, \triangle$, and $\#$ are special symbols not occurring in Σ. Here, we use the symbols "$\rhd$" and "$\lhd$" as special markers to indicate the begin and the end of the tiling, respectively. Furthermore, the symbol "#" is a separator between successive rows. The symbol "$\triangle$" is needed to enforce the vertical constraints on strings that represent tilings, which will become clear later in the proof. It is important to note that we do not use the regular expression $(\star)$ itself in the reduction, as it is not a RE$(a, (+w)^+)$ expression.

We are now ready for the reduction. We define the necessary regular expressions. Let $\overline{b} = (\text{bot}_1, \dots, \text{bot}_n)$ and $\overline{t} = (\text{top}_1, \dots, \text{top}_n)$.

- The following RE$(a, (+w)^+)$ expression ensures that the tiling begins and ends with the bottom and top row, respectively:

$$\begin{aligned} \text{BT} := \rhd \triangle^n \text{bot}_{1,1} \triangle^n \cdots \triangle^n \text{bot}_{n,n} \triangle^n & \\ (\Sigma \cup \{\#, \triangle\})^+ & \\ \triangle^n \text{top}_{1,1} \triangle^n \cdots \triangle^n \text{top}_{n,n} \triangle^n \# \lhd & \end{aligned}$$

- The following expression verifies the horizontal constraints between tiles in colums ℓ and 1, where ℓ is an even number between 1 and n. Together with Hor_{odd}, this expression also ensures that the strings in the intersection are correct encodings of tilings. (That is, the strings are in the language defined by

($\star$).)

$$\mathrm{Hor}_{\mathrm{even}} := \Bigg(\sum_{\substack{2 \leq \ell \leq n-2 \\ \ell \text{ is even} \\ (t_i,t_j) \in V}} (\triangle^n t_{i,\ell} \triangle^n t_{j,\ell+1}) + (\triangleright \triangle^n \mathrm{bot}_{1,1}) + (\triangle^n \mathrm{top}_{n,n} \triangle^n \# \triangleleft) + \sum_{\substack{1 \leq i \leq k \\ 1 \leq j \leq k}} (\triangle^n t_{i,n} \triangle^n \# \triangle^n t_{j,1}) \Bigg)^+ .$$

The last three disjuncts take care of the very first, very last and all start and end tiles of intermediate rows.

- The following expression verifies the horizontal constraints between tiles in columns ℓ and $\ell + 1$, where ℓ is an odd number between 1 and $n - 1$. Together with $\mathrm{Hor}_{\mathrm{ver}}$, this expression also ensures that the strings in the intersection are correct encodings of tilings. (That is, the strings are in the language defined by ($\star$).)

$$\mathrm{Hor}_{\mathrm{odd}} := \Bigg((\triangleright \triangle^n \mathrm{bot}_{1,1} \triangle^n \mathrm{bot}_{2,2}) + (\triangle^n \mathrm{top}_{n-1,n-1} \triangle^n \mathrm{top}_{n,n} \triangle^n \# \triangleleft) + \sum_{\substack{1 \leq \ell \leq n-1 \\ \ell \text{ is odd} \\ (t_i,t_j) \in V}} (\triangle^n t_{i,\ell} \triangle^n t_{j,\ell+1}) + (\triangle^n \#) \Bigg)^+ .$$

- Finally, for each $\ell = 1, \ldots, n$, the following expressions verify the vertical constraints in column ℓ.

$$\mathrm{Ver}_{\ell} := \Bigg(\sum_{(\mathrm{bot}_\ell, t_i) \in V} (\triangleright \triangle^n \mathrm{bot}_{1,1} \triangle^n \cdots \triangle^n \mathrm{bot}_{\ell,\ell} \triangle^{n-i}) + \sum_{\substack{1 \leq i \leq n \\ 1 \leq j \leq k \\ m \neq \ell}} (\triangle^i t_{j,m} \triangle^{n-i}) + \sum_{1 \leq i \leq n} (\triangle^i \# \triangle^{n-i}) + \sum_{1 \leq i \leq n} (\triangle^i \# \triangleleft) + \sum_{\substack{1 \leq i \leq n \\ (t_i,t_j) \in V}} (\triangle^i t_{i,\ell} \triangle^{n-j}) \Bigg)^+ .$$

The crucial idea is that, when we read a tile in the ℓ-th column, we use the string $\triangle^{n-i}$ to encode that we want the tile in the ℓ-th column of the next row to be t_i. By using the disjunctions over the vertical constraints, we can express all the possibilities of the tiles that we allow in the next column.

We explain the purpose of the five disjunctions in the expression. We can assume here that the strings are already in the intersection of $\mathrm{Hor}_{\mathrm{odd}}$ and $\mathrm{Hor}_{\mathrm{even}}$. This way, we know that *(i)* the strings encode tilings in which every row consists of n tiles, *(ii)* we use symbols from Σ_j to encode tiles from the j-th column, and that *(iii)* the string $\triangle^n$ occurs between every two tiles. The first disjunction allows us to get started on the bottom row. It says that we want the ℓ-th tile in the next row to be a t_i such that $(\mathrm{bot}_\ell, t_i) \in V$. The second, and third disjunction ensure that this number i is "remembered" when reading *(a)* tiles that are not on the ℓ-th column or *(b)* the special delimiter symbol "#" which marks the beginning of the next row. The fourth disjunction can only be matched at the end of the tiling, as the symbol $\lhd$ only occurs once in each correctly encoded tiling. Finally, the fifth disjunction does the crucial step: it ensures that, when we matched the ℓ-th tile t_k in the previous row with an expression ending with $t_{k,\ell}\triangle^{n-i}$, and we remembered the number i when matching all the tiles up to the current tile, we are now obliged to use a disjunct of the form $\triangle^i t_{i,\ell}\triangle^{n-j}$ to match the current tile. In particular, this means that the current tile must be t_i, as we wanted. Further, the disjunction over $(t_i, t_j) \in V$ ensures again that the ℓ-th tile in the next column will be one of the t_j's.

It is easy to see that a string w is in the intersection of BT, $\mathrm{Hor}_{\mathrm{even}}$, $\mathrm{Hor}_{\mathrm{odd}}$, $\mathrm{Ver}_1, \ldots, \mathrm{Ver}_n$, if and only if w encodes a correct tiling for D.

The PSPACE-hardness proof for $\mathrm{RE}(a, (+w)^*)$ is completely analogous, except that every "+" (which is not a disjunction) needs to be replaced by a "$*$". □

Theorem 9.18(d) holds as INTERSECTION NON-EMPTINESS for arbitrary regular expressions is in PSPACE. This concludes the proof of Theorem 9.18.

Recall the notion of one-unambiguous regular expressions from Definition 8.9.

Theorem 9.23. INTERSECTION NON-EMPTINESS *is* PSPACE*-complete for*

(a) one-unambiguous regular expressions; and for

(b) $RE^{\leq 3}$.

Proof. The PSPACE upper bound is immediate as INTERSECTION NON-EMPTINESS is in PSPACE for regular expressions in general.

We proceed by showing the lower bound. It is relatively straightforward to reduce CORRIDOR TILING to these problems. The result for (a) is obtained by defining the regular expressions in such a way that they use separate alphabets for the last rows of the tiling and by using separate alphabets for odd and even rows. The result for (b) is obtained by carefully defining the reduction such that each symbol in the expressions is only used for positions with the same offset in a tiling row and the same parity of row number.

We reduce CORRIDOR TILING, which is PSPACE-complete (Theorem 3.23), to both intersection non-emptiness problems. We first show that INTERSECTION NON-EMPTINESS is PSPACE-hard for one-unambiguous regular expressions and then adapt these expressions so that they only contain up to three occurrences of the same alphabet symbol.

Let $D = (T, H, V, \overline{b}, \overline{t}, n)$ be a tiling system. Without loss of generality, we assume that every correct tiling has an even number of rows and that the tiles T are partitioned into three disjoint sets $T_0 \uplus T' \uplus T''$. The idea is that symbols from T'', and T' are only used on the uppermost and one but uppermost row, respectively. We denote symbols from T' and T'' with one and two primes, respectively. The assumption for an even number of rows simplifies the expressions that check the vertical constraints.

From D we construct regular expressions BT, $\mathrm{Hor}_{t,i}$, $\text{Ver-odd}_{t,j}$, $\text{Ver-even}_{t,j}$, and $\text{Ver-last}_{t,j}$ for every $t \in T$, $i \in \{1, \ldots, n-1, n+1, \ldots, 2n-1\}$, and $j \in \{1, \ldots, n\}$. These expressions are constructed such that, for every string w, we have that

$$w \in L(BT) \cap \bigcap_{t,i} L(\mathrm{Hor}_{t,i}) \cap \bigcap_{t,j} \big(L(\text{Ver-odd}_{t,j}) \cap L(\text{Ver-even}_{t,j}) \cap L(\text{Ver-last}_{t,j})\big)$$
$$\text{if and only if } w \text{ encodes a correct tiling for } D. \qquad (\Diamond)$$

For a set of tiles S, we sometimes denote by S the disjunction of all symbols in S whenever this improves readability. We also write S^i for a sequence of i times S. We define the following regular expressions:

- Let $\overline{b} = b_1 \cdots b_n$ and $\overline{t} = t''_1 \cdots t''_n$. Then the expression

  $$BT := b_1 \cdots b_n \bigg(T_0^n \ \big(T_0^n + (T')^n\big)\bigg)^* t''_1 \cdots t''_n$$

  expresses that *(i)* the tiling consists of an even number of rows, *(ii)* the first row is tiled by $\overline{b}$, and *(iii)* the last row is tiled by $\overline{t}$. It is easy to see that this expression is one-unambiguous.

- The following expressions verify the horizontal constraints. For $t \in T$ and each $j \in \{1, \ldots, n-1, n+1, \ldots, 2n-1\}$, the expression $\mathrm{Hor}_{t,j}$ ensures that the right neighbour of t is correct, where t is a tile
  - in the j-th column when $j \leq n-1$, or
  - in the $(j-n)$-th column when $j > n$.

  Formally, we define for each $t \in T$ and each $j \in \{1, \ldots, n-1, n+1, \ldots, 2n-1\}$ the expression

  $$\mathrm{Hor}_{t,j} := \Big(T^{j-1}\big(t(s_1 + \cdots + s_\ell) + ((T - \{t\})T)\big)T^{2n-j-1}\Big)^*,$$

  where $s_1, \ldots, s_\ell$ are all tiles s_i with $(t, s_i) \in H$. This expression is one-unambiguous.

- Correspondingly, we have expressions for the vertical constraints. For each t and each $j \in \{1, \ldots, n\}$, there are three kinds of expressions checking the vertical constraints in the j-th column for the tile t: $\text{Ver-odd}_{t,j}$ checks the vertical constraints on all odd rows but the last, $\text{Ver-even}_{t,j}$ checks the vertical constraints on the even rows, and $\text{Ver-last}_{t,j}$ checks the vertical constraints on the last two rows. We assume that $\{s_1, \ldots, s_\ell\}$ is the set of tiles s_i with $(t, s_i) \in V$.

  We define the expressions $\text{Ver-odd}_{t,j}$ formally as follows:

  $$\text{Ver-odd}_{t,j} := T_0^{j-1} \Big[\big[(tT_0^{n-1}(s_1 + \cdots + s_\ell)) + ((T_0 - \{t\})T_0^n)\big]T_0^{n-j}(T_0^{j-1} + T'^{j-1})\Big]^* T'^{n-j+1}T''^n$$

Intuitively, every occurrence of the tile t in the j-th column of any odd row (except the last odd row) of a tiling matches the leftmost t in Ver-odd$_{t,j}$. The n-th tile starting from t then has to match the disjunction $(s_1 + \cdots + s_\ell)$, which ensures that the vertical constraint is satisfied. If the tile in the j-th column is different from t (which is handled by the subexpression $(T_0 - \{t\})$, the expression allows every tile in the j-th column in the next row. Further, it is easy to see that every string matching the subexpression in the Kleene star has length $2n$, and that the expression Ver-odd$_{t,j}$ is one-unambiguous.

We define the expressions Ver-even$_{t,j}$ formally as follows:

$$\begin{aligned}\text{Ver-even}_{t,j} := {} & T_0^n T_0^{j-1} \\ & \bigg[\Big(\big(tT_0^{n-j}\big[(T_0^{j-1}(s_1 + \cdots + s_\ell)) + (T'^{j-1}(s'_1 + \cdots + s'_\ell))\big]\big) \\ & \qquad\qquad + ((T_0 - \{t\})T_0^{n-j}(T_0^j + T'^j))\Big) \\ & \qquad\qquad\qquad (T_0^{n-j} + T'^{n-j})(T_0^{j-1} + T''^{j-1})\bigg]^* \\ & \qquad\qquad\qquad T''^{n-j+1}\end{aligned}$$

Here, every occurrence of the tile t in the j-th column of any even row (except the last row) of a tiling matches the leftmost t in Ver-odd$_{t,j}$. Depending whether the n-th tile starting from t is on the row tiles with T' or not, the tile then either has to match the disjunction $(s_1 + \cdots + s_\ell)$ or $(s'_1 + \cdots + s'_\ell)$. This ensures that the vertical constraint is satisfied. If the tile in the j-th column is different from t (which is again handled by the subexpression $(T_0 - \{t\})$, the expression allows every tile in the j-th column in the next row. It is easy to see that every string matching the subexpression in the Kleene star has length $2n$, and that the expression Ver-even$_{t,j}$ is one-unambiguous.

Finally, we define the expressions Ver-last$_{t,j}$ as follows:

$$\begin{aligned}\text{Ver-last}_{t,j} := {} & (T_0^{2n})^* T'^{j-1} \\ & \qquad \Big[\big(t'T'^{n-j}T''^{j-1}(s''_1 + \cdots + s''_\ell)\big) + \big((T' - \{t'\})T'^{n-j}T''^{j}\big)\Big]^* \\ & \qquad T''^{n-j}\end{aligned}$$

If the j-th tile of the second-last row of the tiling is t', it matches the leftmost occurrence of t' in Ver-last$_{t,j}$. The expression then ensures that the j-th tile in the last row is in $\{s''_1 + \cdots + s''_\ell\}$. If the j-th tile of the second-last row is different from t' (which is handled by the subexpression $(T' - \{t'\})$), the expression allows every tile in T'' in the j-th position of the last row. It is easy to see that this expression is one-unambiguous.

From the above discussion, it now follows that $(\Diamond)$ holds. Hence, we have shown (a).

We proceed by showing how the maximal number of occurrences of each alphabet symbol can be reduced to three. Notice that we can assume that the given tiling system D uses pairwise disjoint alphabets T_j to tile the j-th column of a tiling. Moreover, we can also assume that D uses pairwise disjoint alphabets to tile the odd rows and the even rows of the tiling, respectively. Hence, the tiles of D are partitioned into the pairwise disjoint sets $T_{j,\text{odd}}$, $T_{j,\text{even}}$, $T'_{j,\text{odd}}$, and $T''_{j,\text{even}}$ for every $j = 1, \ldots, n$. Here, $T'_{j,\text{odd}}$ and $T''_{j,\text{even}}$ are the sets that are used to tile the j-th column of the second-last and the last row of the tiling, respectively.

When we assume that D meets these requirements, it is straightforward to verify that, in the above defined regular expressions, every tile of T occurs at most three times. This concludes the proof of (b). □

Corollary 9.24. INTERSECTION NON-EMPTINESS *is* PSPACE-*complete for one-unambiguous* $RE^{\leq 3}$ *expressions.*

A tractable fragment is the following. It is the class of RE^+ expressions we defined in Chapter 6.

Theorem 9.25. INTERSECTION NON-EMPTINESS *is in* PTIME *for* $RE(a, a^+)$.

Proof. Suppose we are given $\text{RE}(a, a^+)$ expressions $r_1, \ldots, r_n$ in sequence normal form. We describe a PTIME method to decide non-emptiness of $L(r_1) \cap \cdots \cap L(r_n)$.

First of all, the intersection can only be non-empty, if all r_i have the same number m of factors and, for each $j \leq n$, the j-th factor of each r_i has the same base symbol a_j. That is, each r_i can be written as $e_{i,1} \cdots e_{i,n}$ and each $e_{i,j}$ is of the form $a_j[k^i_j, l^i_j]$, for some k^i_j, l^i_j, where $k^i_j \geq 1$.

Let, for each $j \leq m$, $p_j := \max\{k^i_j \mid i \leq n\}$ and $q_j := \min\{l^i_j \mid i \leq n\}$. It is easy to check that $L(r_1) \cap \cdots \cap L(r_n)$ is non-empty, if and only if, for each $j \leq m$, $p_j \leq q_j$. □

10

Minimization of Schema Languages

The concept of unranked regular tree languages lies at the formal basis for many XML schema languages such as DTD, XML Schema and Relax NG. However, both DTD and XML Schema lack the expressive power to define every unranked regular tree language (see Chapter 8 for more details). This situation is different for Relax NG. Not only is the design of Relax NG based on unranked tree automata theory, validators for Relax NG are even typically implemented as tree automata [vdV03].

Tree automata for unranked trees are not only useful in the area of schema languages. They are used as a toolbox in numerous areas of XML-related research such as path and pattern languages [NS00, Sch04] and XML querying [FGK03, NS02]. The focus of the present chapter is on studying the problem of efficiently minimizing such automata.

Besides being a fundamental problem of theoretical interest, the minimization problem for tree automata or for XML schemas also has its use in practical applications. In the context of XML schema languages, minimized schemas would improve the running time or memory consumption for document validation. For static tests involving schemas, such as typechecking for XML transformations, a schema minimizer can be used as a preprocessor to improve the running time of the typechecker. The problem of minimizing the number of states of an unranked tree automaton is particularly relevant for classes of *deterministic automata*, since, for these automata, minimization can be done both efficiently and leads to unique canonical representatives of regular languages, as is well-known for string languages and ranked tree languages. It is also well-known that minimal non-deterministic automata are neither unique, nor efficiently computable [JR93, Mal04].

The investigation of efficient minimization of bottom-up deterministic automata for unranked tree languages started quite recently [CLT05, RB04]. The deterministic devices considered there, however, differ from the standard deterministic automata in database theory — the bottom-up deterministic unranked tree automata (DTAs) of Brüggemann-Klein, Murata, and Wood [BKMW01]. In this chapter, we investigate efficient[1] minimization starting from such DTAs.

The transition relation of DTAs uses regular string languages over the states of the automaton to express horizontal recursion. However, it is not specified how these regular string languages should be represented. In practice, this is usually done by finite automata or regular expressions. If we allow for non-deterministic finite automata in DTAs, then minimization becomes PSPACE-hard, because minimization is already PSPACE-hard for the non-deterministic finite automata. As we are interested in *efficient* minimization, we restrict the finite subautomata in DTAs to be deterministic too. That is, we study the DTA(DFA)s as defined in Section 2.2.

We prove two unexpected results for these DTA(DFA)s. We present a counterexample for the uniqueness of minimal DTA(DFA)s that represent a given regular language. We then prove that minimization becomes NP-complete. Both results are in strong contrast to what is known for bottom-up deterministic automata in the ranked case. Our NP-hardness proof refines the proof techniques from [JR93, Mal04], showing NP-hardness of minimization for classes of finite string automata with a limited amount of non-determinism.

Even though minimization for DTA(DFA)s is intractable, there exist automata models for unranked trees that do allow for efficient minimization. Examples of such models are *stepwise tree automata* [CNT04], *parallel tree automata* [CLT05, RB04], and bottom-up deterministic automata over the standard *first-child next-sibling encoding* of regular tree languages. As each of these models allows for tractable minimization and unique minimal representatives, we compare the models in terms of succinctness. We obtain that stepwise tree automata yield the smallest representations of unranked tree languages. In general, they are quadratically smaller than parallel tree automata and exponentially smaller than tree automata over the first-child next-sibling encoding (up to inversion).

Finally, we investigate top-down deterministic models for unranked trees, which of interest as they form a theoretical basis for XML Schema [SMT05]. As we argued in Chapter 8, XML schemas with the EDC constraint can be abstracted as *single-type extended DTDs*, which are top-down deterministic, but not very expressive. A more expressive notion is the notion of *restrained competition extended DTDs* that we defined in Chapter 8. We show that the latter notion still allows for a polynomial time minimization algorithm and unique minimal models for a regular language. Moreover, when given an input that satisfies the single-type restriction, our minimization algorithm outputs a minimal single-type model. It therefore also minimizes single-type models.

[1] That is, PTIME, under the assumption that PTIME $\neq$ NP.

10.1 Conventions and Notations

In order to improve the presentation and readability of the present chapter, we use some conventions and notations that, in some cases, slightly deviate from the rest of the dissertation.

- We assume here that, in a *binary tree*, every node either has 0 or 2 children. Hence, compared with the chapters in Part I, we disallow that a node has only 1 child. Formally, we assume that the set $b\mathcal{T}_\Sigma$ consists of *binary* Σ*-trees*, in which rank_Σ in the binary alphabet $(\Sigma, \text{rank}_\Sigma)$ is a function from Σ to $\{0, 2\}$.
- As we argued in the introduction, we are interested in classes which allow for *efficient* minimization. Since minimization is already PSPACE-complete for NFAs, we do not investigate investigate DTA(NFA)s. Hence, the abbreviation DTA in the present chapter always stands for DTA(DFA).
- As the present chapter is concerned about minimizing the number of *states* or *types* needed to represent a language, we will use the number of states and types of a finite automaton or EDTD as the measure for its *size*. In particular, we say that

  (1) the *size* $|N|$ of an NFA $N = (Q, \Sigma, \delta, I, F)$ is $|Q|$;
  (2) the *size* $|B|$ of a BTA $B = (Q, \Sigma, \delta, F)$ is $|Q|$;
  (3) the *size* $|B|$ of a DTA(DFA) $B = (Q, \Sigma, \delta, F)$ is $|Q| + \sum_{q \in Q, a \in \Sigma} |D_{q,a}|$, where $D_{q,a}$ is the DFA representing the regular language $\delta(q, a)$; and,
  (4) the *size* $|D|$ of an EDTD(DFA) $E = (\Sigma, \Delta, d, s_d, \mu)$ is $|\Delta| + \sum_{a^i \in \Delta} |D_{a^i}|$, where $d(a^i) = D_{a^i}$.

- Some finite automata in the present chapter use the same set for their states and their alphabet. To improve readability for defining the transitions of such an automaton we adopt the following conventions:
  - when $A = (Q, \Sigma, \delta, I, F)$ is an NFA, we sometimes say that $q_1 \overset{a}{\to} q_3$ *is a transition of* A when $q_3 \in \delta(q_1, a)$; and,
  - when $A = (Q, \Sigma, \delta, F)$ in a BTA, we sometimes say that $a(q_1, q_2) \to q_3$ or $a \to q$ *is a transition of* A when $\delta(q_3, a) = \{q_1 q_2\}$ or $\delta(q, a) = \{\varepsilon\}$, respectively.

10.2 Complexity of Minimization

The central decision problem of this chapter is the *minimization problem*, which is parametrized by a class $\mathcal{C}$ of automata. Minimization is closely related to equivalence, inclusion, and universality. In the present section, we define these problems formally and present an overview over existing and new complexity results for automata minimization.

We say that a finite automaton A over strings, binary trees, or unranked trees is *universal* if $\Sigma^* \subseteq L(A)$, $b\mathcal{T}_\Sigma \subseteq L(A)$, or $\mathcal{T}_\Sigma \subseteq L(A)$, respectively.

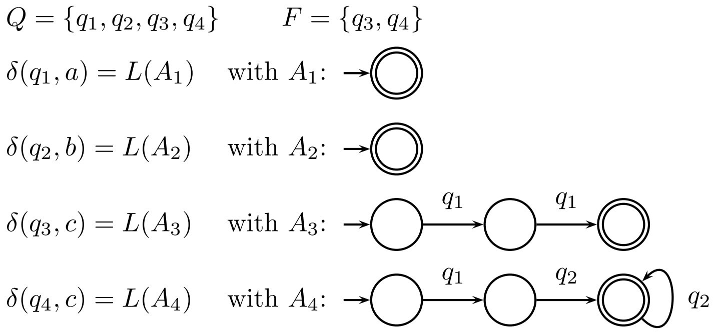

$Q = \{q_1, q_2, q_3, q_4\}$
$F = \{q_3, q_4\}$
$\delta(q_1, a) = L(A_1)$ with A_1:
$\delta(q_2, b) = L(A_2)$ with A_2:
$\delta(q_3, c) = L(A_3)$ with A_3:
q_1
q_1
$\delta(q_4, c) = L(A_4)$ with A_4:
q_1
q_2
q_2

Figure 10.1: Example for a DTA (Q, Σ, δ, F) of size 12.

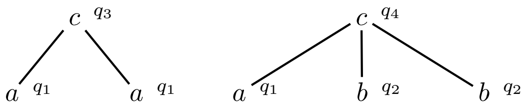


Figure 10.2: Two successful runs by the DTA in Figure 10.1 annotated to the trees.

MINIMIZATION: Given an automaton $A \in \mathcal{C}$ and a natural number $m \in \mathbb{N}$, does there exist an $A' \in \mathcal{C}$ such that A and A' accept the same language and the size of A' is at most m?

EQUIVALENCE: Given $A, B \in \mathcal{C}$, does $L(A) = L(B)$ hold?

INCLUSION: Given automata $A, B \in \mathcal{C}$, does $L(A) \subseteq L(B)$ hold?

UNIVERSALITY: Given an automaton $A \in \mathcal{C}$, is A universal?

The minimization problem for a class $\mathcal{C}$ of automata can the be solved by a non-deterministic algorithm that, on input A and m, guesses another A' with size at most m and tests EQUIVALENCE between A and A'.

As we will see in Table 10.1, it often holds that UNIVERSALITY is easier than MINIMIZATION. This will be useful to prove lower bounds for MINIMIZATION problems.

Everyone agrees that a DFA is indeed a deterministic device. When, during a computation, the automaton is in a certain state at a certain node, the next state is always uniquely determined. We raise the question whether a DTA(DFA) is a fully deterministic representation of unranked tree languages or not? Clearly, every state computed by a run is uniquely determined due to bottom-up determinism. The internal computation inside of the automata in the transition function is deterministic too, since performed by DFAs. However, choice is needed when one has to decide which transition to apply for a given letter. It requires guessing or testing the possibilities. Intuitively, DTA(DFA)s represent the internal regular languages over states by *a disjoint union of DFAs,* which is in fact an unambiguous representation with one non-deterministic step: the choice of the initial state.

The DTA(DFA) in the example in Figure 10.1 has two rules for the letter c. In the first successful run in Figure 10.2, we have to chose the upper rule, in the second one the lower rule. It is precisely this representation as a disjoint union of DFAs that will lead to the NP-hardness of DTA(DFA) minimization.

10.2.1 Result Overview

In Table 10.1, we collect complexity results about automata minimization and the related problems.

	EQUIVALENCE, INCLUSION, UNIVERSALITY	MINIMIZATION
DFA	NLOGSPACE	in PTIME [HMU01]
UFA	in PTIME [SH85]	in NP (from EQUIVALENCE) NP-hard [JR93]
NFA	PSPACE [SM73]	PSPACE [SM73]
DBTA	in PTIME [CDG$^+$01] PTIME-hard [Coo74]	in PTIME [CDG$^+$01] PTIME-hard (from UNIVERSALITY)
UBTA	PTIME [Sei90]	in NP (from EQUIVALENCE) NP-hard (from UFAs)
NBTA	EXPTIME [Sei90]	in EXPTIME (from EQUIVALENCE) EXPTIME-hard (from UNIVERSALITY)
DTA	in PTIME (Theorem 10.1) PTIME-hard (from DBTAs)	in NP (from EQUIVALENCE) **NP-hard (Theorem 10.11)**
UTA	in PTIME (Theorem 10.1) PTIME-hard (from UBTAs)	in NP (from EQUIVALENCE) NP-hard (from UFAs)
NTA	in EXPTIME (from NBTAs) EXPTIME-hard (from NBTAs)	in EXPTIME (from EQUIVALENCE) EXPTIME-hard (from UNIVERSALITY)

Table 10.1: Complexity overview for nondeterministic, unambiguous, and bottom-up and/or deterministic automata for stings, binary trees, and unranked trees. Here, DTA, UTA and NTA stand for DTA(DFA), UTA(UFA), and NTA(NFA), respectively.

For finite automata, all presented results are well known, perhaps with the exception for UFAs, for which EQUIVALENCE, INCLUSION, and UNIVERSALITY are in PTIME, while MINIMIZATION is NP-complete. For traditional tree automata, the situation is well established too. The PTIME lower bound for UNIVERSALITY of DBTAs follows from a straightforward reduction from PATH SYSTEMS, which is known to be PTIME-complete [Coo74]. The reduction is similar to the one for proving that EMPTINESS for DTD(DFA)s is PTIME-hard in Proposition 3.16. Notice that the same complexities hold for UFAs and UBTAs, even though the proofs for the upper bounds become more involved for UBTAs. For the EXPTIME-hardness of NBTA minimization, note that we can immediately reduce NBTAUNIVERSALITY to MINIMIZATION, since an automaton A with alphabet Σ is universal if and only if *(i)* $|A| = 1$, *(ii)* $a \in L(A)$ for every a with $\text{rank}_\Sigma(a) = 0$, and *(iii)* $b(a\ a) \in L(A)$ for every a, b with $\text{rank}_\Sigma(b) = 2$ and $\text{rank}_\Sigma(a) = 0$.

Results for Unranked Tree Automata

For NTA(NFA)s, the EXPTIME-hardness of EQUIVALENCE, INCLUSION, and UNIVERSALITY are immediate from the binary case, since every NBTA can be encoded in PTIME into an NTA. The EXPTIME upper bound carries over from the case of traditional tree automata for binary trees, based on some binary encoding for unranked trees (Proposition 3.8). Alternatively, the EXPTIME upper bound also immediately follows from Theorem 4.1.

The EXPTIME-hardness of MINIMIZATION follows from a reduction from UNIVERSALITY similarly to the case of traditional tree automata: An NTA A with alphabet Σ is universal if and only if *(i)* $|A| = |\Sigma| + 1$, *(ii)* for every $a \in \Sigma$, $a \in L(A)$, and, *(iii)* for every $a, b \in \Sigma$, $a(b) \in L(A)$.

Theorem 10.1. INCLUSION *for DTA(DFA)s and UTA(UFA)s is in* PTIME.

Proof. Given two UTA(UFA)s, we can translate them in PTIME into UBTAs with respect to a binary encoding of unranked trees. For DTA(DFA)s and with respect to the standard first-child next-sibling encoding of unranked trees, this has been proposed in Lemma 4.24 of [GKPS05]. Due to the work of Seidl, we can test inclusion of UBTAs in PTIME [Sei90]. □

Even though the proposed PTIME algorithm seems overly complicated, it is, to the best of our knowledge, not known whether the "standard" inclusion test of DTA(DFA)s works in PTIME. The standard test would, given two DTA(DFA)s A and B, test whether $L(A)$ has an empty intersection with the complement of $L(B)$. The difficulty of this approach lies in finding a small DTA(DFA) for the complement of $L(B)$. This is *not* trivial unless B is *complete*, then one simply has to switch final and non-final states. (A NTA $B = (Q, \Sigma, \delta, F)$ is complete when, for every $w \in Q^*$, there exists a transition $\delta(q, a) = L$ such that $w \in L$.)

From Theorem 10.1 we can immediately derive the following corollary.

Corollary 10.2. MINIMIZATION *of DTA(DFA)s and UTA(UFA)s is in* NP.

There remains one further result in Table 10.1 that we have not discussed so far. This is the NP-hardness result of DTA minimization, which is the subject of Section 10.3. Alternative notions of bottom-up determinism for other kinds of automata on unranked trees will be discussed in Section 10.4. Restrained competition schemas are studied in Section 10.5.

10.3 Minimizing DTAs

In this section we study the minimization problem of DTAs. We show two unexpected negative results:

(1) There are regular tree languages for which no unique (up to isomorphism) minimal DTA exists.

Given a regular tree language L, there does *not* exist an (up to isomorphism) unique minimal DTA that accepts L.

(2) The minimization problem for DTAs even turns out to be NP-complete.

10.3.1 Minimal Automata are not Unique

We show the non-uniqueness by means of an example. Consider the regular string languages L_1, L_2, and L_3 defined by the regular expressions

$$(bbb)^*,\ b(bbbbbb)^*,\ \text{and}\ bb(bbbbbbbbb)^*,$$

respectively. Notice that L_1, L_2 and L_3 are pairwise disjoint, and that the minimal DFAs A_1, A_2, and A_3 accepting L_1, L_2, and L_3 have 3, 6, and 9 states, respectively. Notice that the minimal DFAs B_1 and B_2 accepting $L_1 \cup L_2$ and $L_1 \cup L_3$ (which are depicted as parts of Figure 10.3) have 6 and 9 states, respectively. Define L to be the language $L_1 \cup L_2 \cup L_3$ and consider the tree language $T = \{r(a(w)) \mid w \in L\}$.

There exist two non-isomorphic minimal DTAs for T. The first one, $N_1 = (Q_1, \Sigma, \delta_1, F_1)$, is defined in Figure 10.3(a). Notice that the size of N_1 is

$$|Q_1| + 1 + |B_1| + |A_3| + 2 = 4 + 2 + 6 + 9 + 1 = 22.$$

The other automaton, $N_2 = (Q_2, \Sigma, \delta_2, F_2)$, is defined in Figure 10.3(b). Notice that the size of N_2 is

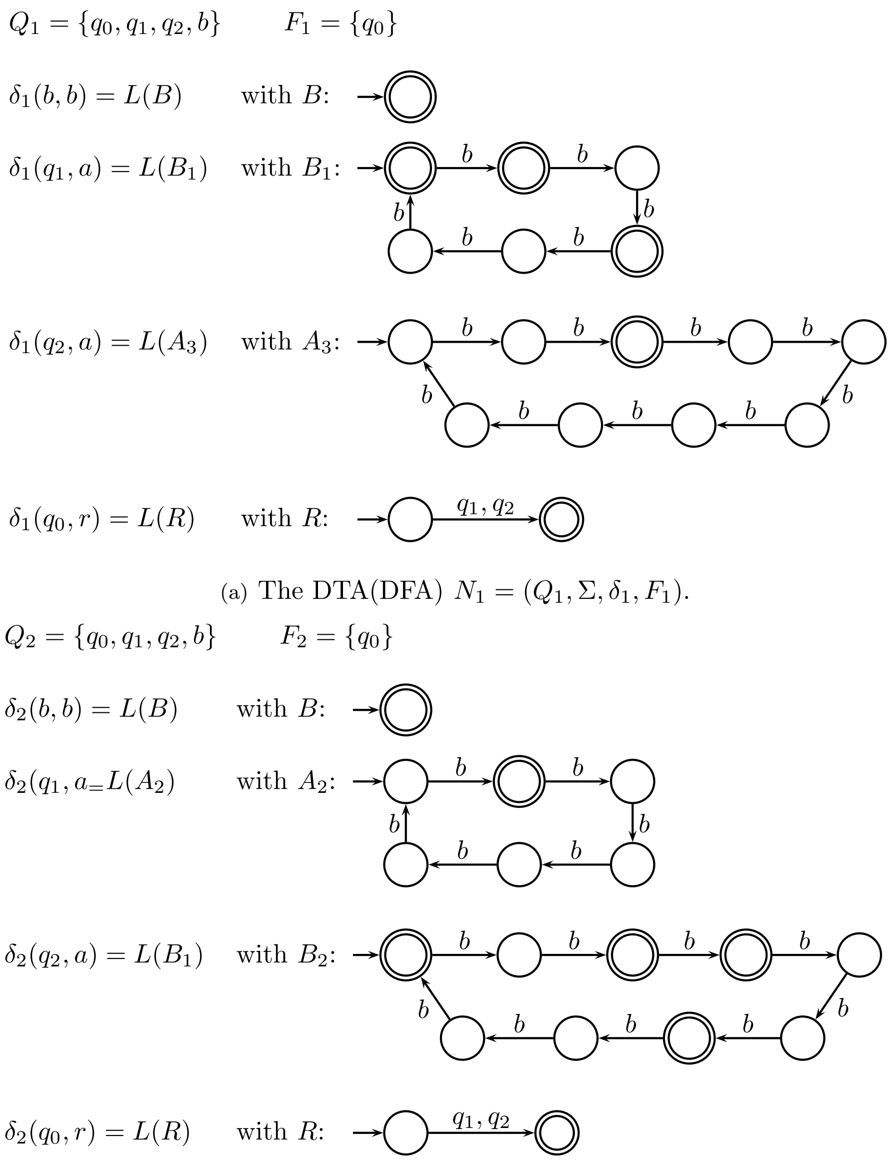
$$|Q_2| + 1 + |B_2| + |A_2| + 2 = 4 + 2 + 9 + 6 + 1 = 22.$$

Of course, there are other possibilities to write $L = L_1 \cup L_2 \cup L_3$ as a disjoint union of regular languages. The obvious combinations one can make with A_1, A_2 and A_3 lead to DTAs of size 26 (using A_1, A_2 and A_3), 28 (using $(A_2 \cup A_3)$ and A_1) and 24 (one automaton for L).

For the following argument, we make use of several well-known theorems for regular string languages for a one-letter alphabet:

Theorem 10.3 (for example, [PS02]). *A language L over $\{a\}$ is regular if and only if there are two integers $n_0 \geq 0, k \geq 1$, such that for any $n \geq n_0$, $a^n \in L$ if and only if $a^{n+k} \in L$. Moreover, when L is regular, the minimal DFA for L contains a cycle with k states.*

We show that no other combination of splitting L into a union of regular languages results in a smaller DTA accepting T. First, observe that any DTA N defining T needs at least three states in states(N), since all trees in T have depth three (that is, they contain three nodes on the shortest path from the root to a leaf). However, as argued above, the minimal size of such a DTA with three states is $3 + 1 + 18 + 2 = 24$. The only way to obtain a smaller equivalent DTA is then to define L as a union of DFAs, of which the sum of the number of states is strictly smaller than $9 + 6 = 15$.



$Q_1 = \{q_0, q_1, q_2, b\}$ $F_1 = \{q_0\}$

$\delta_1(b, b) = L(B)$ with B:

$\delta_1(q_1, a) = L(B_1)$ with B_1:

$\delta_1(q_2, a) = L(A_3)$ with A_3:

$\delta_1(q_0, r) = L(R)$ with R:

(a) The DTA(DFA) $N_1 = (Q_1, \Sigma, \delta_1, F_1)$.

$Q_2 = \{q_0, q_1, q_2, b\}$ $F_2 = \{q_0\}$

$\delta_2(b, b) = L(B)$ with B:

$\delta_2(q_1, a_{=}L(A_2)$ with A_2:

$\delta_2(q_2, a) = L(B_1)$ with B_2:

$\delta_2(q_0, r) = L(R)$ with R:

(b) The DTA(DFA) $N_2 = (Q_2, \Sigma, \delta_2, F_2)$.

Figure 10.3: Two equivalent minimal DTA(DFA)s that are not isomorphic.

However, if we write L as a union of DFAs, there must be at least one DFA D_1 that accepts an infinite number of strings in L_2. It is easy to see that D_1 has a cycle with 6 states, as D_1 may not accept strings *not* in L (applying Theorem 10.3 with any $k \leq 6$ would imply that $L(D_1) \not\subseteq L$). Analogously, we can argue that there must be at least one DFA D_2 that accepts an infinite number of strings in L_3. If $D_2 \neq D_1$, then we can obtain analogously that D_2 has a cycle with 9 states. If $D_1 = D_2$, we obtain analogously that D_1 has at least 18 states. Therefore, the above automata are indeed minimal for T, and as Figure 10.3 shows, they are clearly not isomorphic.

10.3.2 Minimization is NP-Complete

As Section 10.3.1 illustrates, the problem of defining a regular string language as a small disjoint union of DFAs lies at the heart of the minimization problem for DTAs. We refer to this problem as MINIMAL DISJOINT UNION and we define it formally later in this section.

In this section, we show that MINIMAL DISJOINT UNION is NP-complete by a reduction from VERTEX COVER. Actually, MINIMAL DISJOINT UNION is even NP-complete when we are asked to define a regular string language as a small disjoint union of *two* DFAs. The proof for this result is technically the hardest proof in the chapter, the reduction is technical but interesting in its own right: it shows that minimizing finite string automata with a very limited amount of non-determinism is NP-complete.

We start by formally defining the decision problems that are of interest to us. Given a graph $G = (V, E)$ such that V is its set of vertices and $E \subseteq V \times V$ is its set of edges, we say that a set of vertices $VC \subseteq V$ is a *vertex cover* of G if, for every edge $(v_1, v_2) \in E$, VC contains v_1, v_2, or both. We can assume without loss of generality that G does not contain *self-loops*, that is, edges of the form (v_1, v_1).

If B and C are finite collections of finite sets, we say that B is a *normal basis* of C if, for each $c \in C$, there is a pairwise disjoint subcollection B_c of B whose union is c. For $m \in \mathbb{N}_0$, we say that B is a *K-separable normal basis* if B can be written as a disjoint union $B_1 \uplus \cdots \uplus B_K$ and, for each $j = 1, \ldots, K$, the subcollection B_c of B contains at most one element from B_j. The *size* of a collection of finite sets is the number of finite sets it contains.

We say that a collection C of sets contains *obsolete* symbols if there exist two elements $a \neq b$ such that, for every $c \in C, a \in c \Leftrightarrow b \in c$.

We consider the following decision problems.

VERTEX COVER: Given a pair (G, k) where G is a graph and k is an integer, does there exist a vertex cover of G of size at most k?

NORMAL SET BASIS: Given a pair (C, s) where C is a finite collection of finite sets and s is an integer, does there exist a normal basis of C of size at most s?

K-SEPARABLE NORMAL SET BASIS: Given a pair (C, s) where C is a finite collection of finite sets and s is an integer, does there exist a K-separable normal set basis of C of size at most s?

K-MINIMAL DISJOINT UNION: Given a pair (M, ℓ) where M is a DFA and ℓ is an integer, do there exist DFAs $M_1, \ldots, M_K$ such that

(1) $L(M) = L(M_1) \cup \cdots \cup L(M_K)$; and

(2) for every $i \neq j$, $L(M_i) \cap L(M_j) = \emptyset$; and

(3) $\sum_{i=1}^{K} |M_i| \leq \ell$?

The first two problems are known to be NP-complete [JR93]. We will show that the last two problems are NP-complete (for $K \geq 2$) as well.

We start by showing that NORMAL SET BASIS and K-SEPARABLE NORMAL SET BASIS are NP-complete for every $K \geq 2$. We revisit a slightly modified reduction which is due to Jiang and Ravikumar [JR93], as our further results heavily rely on a construction in their proof.

Lemma 10.4 (Jiang and Ravikumar [JR93])**.** NORMAL SET BASIS *is* NP*-complete.*

Proof. Obviously, NORMAL SET BASIS is in NP. Indeed, given an input (C, s) for NORMAL SET BASIS, the NP algorithm simply guesses a collection B and verifies whether it is a normal set basis for C of size at most s.

We show that NORMAL SET BASIS is NP-hard by a reduction from VERTEX COVER. Given an input (G, k) of VERTEX COVER, where $G = (V, E)$ is a graph and k is an integer, we construct in LOGSPACE an input (C, s) of NORMAL SET BASIS, where C is a finite collection of finite sets and s is an integer. In particular, (C, s) is constructed such that

G has a vertex cover of size at most k if and only if

C has a normal basis of size at most s.

For a technical reason which will become clear later in the chapter, we assume without loss of generality that $k < |E| - 3$. Notice that, under this restriction, VERTEX COVER is still NP-complete problem under LOGSPACE reductions.

Formally, let $V = \{v_1, \ldots, v_n\}$. For each $i = 1, \ldots, n$, define c_i to be the set $\{x_i, y_i\}$ corresponding to the node v_i. Let (v_i, v_j) be in E with $i < j$. To each such edge we associate five sets as follows:

$$\begin{aligned} c_{ij}^1 &:= \{x_i, a_{ij}, b_{ij}\}, \\ c_{ij}^2 &:= \{y_j, b_{ij}, d_{ij}\}, \\ c_{ij}^3 &:= \{y_i, d_{ij}, e_{ij}\}, \\ c_{ij}^4 &:= \{x_j, e_{ij}, a_{ij}\}, \text{ and} \\ c_{ij}^5 &:= \{a_{ij}, b_{ij}, d_{ij}, e_{ij}\}. \end{aligned}$$

Figure 10.4 contains a graphical representation of the constructed sets $c_i, c_j, c_{ij}^1, \ldots, c_{ij}^5$ for some $(v_i, v_j) \in E$.

Then, set

$$C := \{c_i \mid 1 \leq i \leq n\} \cup \{c_{ij}^t \mid (v_i, v_j) \in E, i < j, \text{ and } 1 \leq t \leq 5\}$$

and

$$s := n + 4|E| + k.$$

Notice that the size of C is $n + 5|E|$ and that C does not contain obsolete symbols. Obviously, C and s can be constructed from G and k in polynomial time.

We show that the given reduction is also correct, that is, that G has a vertex cover of size at most k if and only if C has a normal set basis of size at most s.

($\Rightarrow$): Let G have a vertex cover VC of size k. We need to show that C has a normal basis B of size $s = n + 4|E| + k$.

Thereto, we define a collection B of sets as follows. For every $v_i \in V$,

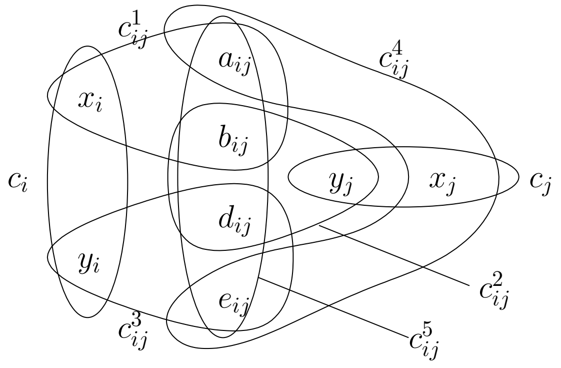


Figure 10.4: The constructed sets $c_i, c_j, c_{ij}^1, \ldots, c_{ij}^5$ in the proof of Lemma 10.4.

- if $v_i \in VC$, we include both $\{x_i\}$ and $\{y_i\}$ in B;
- otherwise, we include $c_i = \{x_i, y_i\}$ in B.

The number of sets included in B so far is $2k + (n - k) = k + n$. Let $e = (v_i, v_j)$ (where $i \leq j$) be an arbitrary edge in G. Since VC is a vertex cover, either v_i or v_j (or both) is in VC. When v_i is in VC, we additionally include the sets

$$\begin{array}{ll} r_{ij}^1 = \{a_{ij}, b_{ij}\}, & r_{ij}^2 = \{d_{ij}, e_{ij}\}, \\ r_{ij}^3 = \{y_j, b_{ij}, d_{ij}\}, \text{ and } & r_{ij}^4 = \{x_j, a_{ij}, e_{ij}\} \end{array}$$

in B. When v_i is not in VC, we additionally include the sets

$$\begin{array}{ll} r_{ij}^5 = \{a_{ij}, e_{ij}\}, & r_{ij}^6 = \{b_{ij}, d_{ij}\}, \\ r_{ij}^7 = \{x_i, a_{ij}, b_{ij}\}, \text{ and } & r_{ij}^8 = \{y_i, d_{ij}, e_{ij}\} \end{array}$$

in B. This completes the definition of B. Notice that, when $v_i \in VC$, c_{ij}^1, c_{ij}^3, and c_{ij}^5 can be expressed as a disjoint union of members of B as

$$c_{ij}^1 = \{x_i\} \uplus r_{ij}^1, \qquad c_{ij}^3 = \{y_i\} \uplus r_{ij}^2, \qquad c_{ij}^5 = r_{ij}^1 \uplus r_{ij}^2$$

and that $c_{ij}^2 = r_{ij}^3$ and $c_{ij}^4 = r_{ij}^4$ are members of B. Analogously, when $v_i \notin VC$, c_{ij}^2, c_{ij}^4, and c_{ij}^5 can be expressed as a disjoint union of members of B as

$$c_{ij}^2 = \{y_j\} \uplus r_{ij}^6, \qquad c_{ij}^4 = \{x_j\} \uplus r_{ij}^5, \qquad c_{ij}^5 = r_{ij}^5 \uplus r_{ij}^6$$

and $c_{ij}^1 = r_{ij}^7$ and $c_{ij}^3 = r_{ij}^8$ are members of B. Since the total number of sets included in B for each edge is four, the size of B is $(k + n) + 4|E| = s$. From the foregoing argument it is also obvious that B is a normal basis of C.

Notice that B is a 2-separable normal set basis for C. Indeed, we can partition B into the sets

$$\begin{aligned} B_1 = \{\{x_i\}, \{x_j, y_j\} \mid v_i \in VC, v_j \notin VC\} \\ \cup \{r_{ij}^2, r_{ij}^3 \mid (v_i, v_j) \in E, i < j, v_i \in VC\} \\ \cup \{r_{ij}^6, r_{ij}^7 \mid (v_i, v_j) \in E, i < j, v_i \notin VC\} \end{aligned}$$

and

$$B_2 = \{\{y_i\} \mid v_i \in VC\} \\ \cup \{r_{ij}^1, r_{ij}^4 \mid (v_i, v_j) \in E, i < j, v_i \in VC\} \\ \cup \{r_{ij}^5, r_{ij}^8 \mid (v_i, v_j) \in E, i < j, v_i \notin VC\},$$

which satisfy the necessary condition.

($\Leftarrow$): Suppose that C has a normal basis B of size at most $s = n + 4|E| + k$. We can assume without loss of generality that no proper subcollection of B is a normal basis. We show that G has a vertex cover VC of size at most k. Define $VC = \{v_i \mid$ both $\{x_i\}$ and $\{y_i\}$ are in $B\}$. Let k' be the number of elements in VC. The number of sets in B consisting of only x_i and/or y_i is at least $n+k'$. This can be seen from the fact that B must have the subset c_i for all i such that $v_i \notin VC$. Thus there are $n-k'$ such sets, in addition to $2k'$ singleton sets corresponding to i's such that $v_i \in VC$. Let $E' \subseteq E$ be the set of edges covered by VC, that is, $E' = \{(v_i, v_j) \mid v_i$ or v_j is in $VC\}$. The following observation can easily be shown:

Observation: For any $e \in E$ at least four sets of B (in addition to sets $c_i, c_j, \{x_i\}$, and $\{x_j\}$) are necessary to cover the five sets c_{ij}^t, $t = 1, \ldots, 5$. Further, at least five sets (in addition to sets $c_i, c_j, \{x_i\}$, and $\{x_j\}$) are required to cover them if $e \notin E'$.

Now the total number of sets needed to cover C is at least $n + k' + 4|E'| + 5(|E| - |E'|)$, which we know is at most $s = n + 4|E| + k$. Hence, we obtain that $n + k' + 5|E| - |E'| \leq n + 4|E| + k$, which implies that $k' + |E| - |E'| \leq k$. We conclude the proof by showing that there is a vertex cover VC' of size $|E| - |E'| + k'$. Add one of the end vertices of each edge $e \in E - E'$ to VC. This vertex cover is of size $|E| - |E'| + k' \leq k$. □

In the proof of Lemma 10.4, we have shown that, if G has a vertex cover of size k, then C has a 2-separable normal set basis of size s. Conversely, we have shown that, if C has a normal set basis of size s (which is allowed to be K-separable for any $K \geq 2$), G has a vertex cover of size k. Hence, we immediately obtain the following proposition:

Proposition 10.5. *There exists a set of inputs* $\boldsymbol{I}$ *for* NORMAL SET BASIS*, such that*

- NORMAL SET BASIS *is* NP*-complete for inputs in* $\boldsymbol{I}$*; and,*
- *for each* $(C, s) \in \boldsymbol{I}$*, the following are equivalent:*
    1. *C has a normal set basis of size s.*
    2. *C has a K-separable normal set basis of size s for any $K \geq 2$.*

Moreover, C does not contain obsolete symbols and, for each (C, s) in $\boldsymbol{I}$, we have that $s < |C| - 3$.

Proof. The set $\mathbf{I}$ is obtained by applying the reduction in Lemma 10.4 to inputs of VERTEX COVER. In this reduction we observed that C was constructed such that it does not contain obsolete symbols. For the size constraint, we have to recall the assumption in Lemma 10.4, that $k < |E| - 3$. Hence, we obtain that $s = n + 4|E| + k < n + 5|E| - 3 = |C| - 3$. □

The intuition behind Proposition 10.5 is that, C has a normal set basis of size s if and only if C has a 2-separable normal set basis of size s for any input (C, s) in $\mathbf{I}$. Of course, the latter property does not hold for the set of all possible inputs for the normal set basis problem.

Since the proof of Lemma 10.4 shows that NORMAL SET BASIS is an NP-complete problem for inputs in $\mathbf{I}$, we immediately obtain the following:

Corollary 10.6. *For every $K \geq 2$, K-*SEPARABLE NORMAL SET BASIS *is* NP-*complete.*

Our next goal is to show a result for MINIMAL DISJOINT UNION which is similar to Proposition 10.5. However, in order to apply the result immediately to DTA MINIMIZATION later, we need to treat a minor technical issue. (Readers who are only interested in the NP-hardness of K-MINIMAL DISJOINT UNION can safely skip the following definition.) Due to the fact that the internal DFAs of NTAs do not read alphabet symbols, but states of the tree automaton, we need to take extra care of the languages we define in the reduction for the MINIMAL DISJOINT UNION problem: we will require that the languages do not contain *interchangeable symbols*, which is defined as follows.

Definition 10.7. Given a string language L over an alphabet Σ, we say that two symbols $a, b \in \Sigma$, $a \neq b$, are *interchangeable with respect to* L if, for every two Σ-strings u and v, we have that $uav \in L \Leftrightarrow ubv \in L$. We say that L *contains interchangeable symbols* if there exist $a, b \in \Sigma$, $a \neq b$, which are interchangeable with respect to L. $\diamond$

We are now ready to show the following lemma.

Lemma 10.8. *For every $K \geq 2$, K-*MINIMAL DISJOINT UNION *is* NP-*complete.*

Proof. The NP upper bound follows from the fact that we can guess a disjoint union of sufficiently small size and verify in PTIME that it is equivalent (see also Section 10.2.1, where we recall that testing equivalence of unambiguous string automata is in PTIME).

For the lower bound, we reduce from 2-SEPARABLE NORMAL SET BASIS. To this end, let (C, s) be an input of 2-SEPARABLE NORMAL SET BASIS. Hence, C is a collection of n sets and s is an integer. According to Proposition 10.5, we can assume without loss of generality that $(C, s) \in \mathbf{I}$, that is, C has a normal set basis of size s if and only if C has a K-separable normal set basis of size s for any $K \geq 2$. Moreover, we can assume that $s < n - 3$.

We construct in LOGSPACE an input (M, ℓ) of MINIMAL DISJOINT UNION such that

C has a normal set basis of size at most s if and only if
K-MINIMAL DISJOINT UNION is true for (M, ℓ) for any $K \geq 2$.

Intuitively, M accepts the language $\{ca \mid c \in C \text{ and } a \in c\}$, which is a finite language of words of length two.

We state the following claim, which we prove later. The claim is needed later in the chapter but is not important for the proof of the present lemma.

Claim 10.9. *$L(M)$ does not contain interchangeable symbols.*

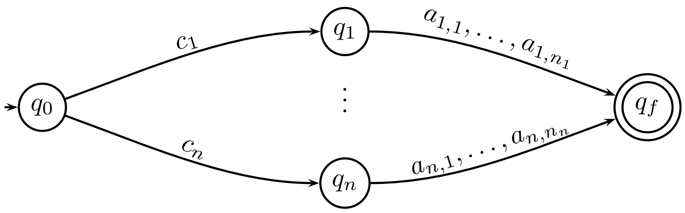


Figure 10.5: Illustration of a fragment of the constructed automaton M in the proof of Lemma 10.8.

Formally, let $C = \{c_1, \ldots, c_n\}$ and $c_i = \{a_{i,1}, \ldots, a_{i,n_i}\}$. Then, M is defined over alphabet

$$\Sigma_M = \bigcup_{1 \leq i \leq n} \{c_i, a_{i,1}, \ldots, a_{i,n_i}\}.$$

The state set of M is $Q_M = \{q_0, q_1, \ldots, q_n, q_f\}$, and the initial and final state sets of M are $\{q_0\}$ and $\{q_f\}$, respectively. The transitions of M are depicted in Figure 10.5 and are formally defined as follows:

- for every $i = 1, \ldots, n$, $\delta(q_0, c_i) = \{q_i\}$; and
- for every $i = 1, \ldots, n$, $j = 1, \ldots, n_i$, $\delta(q_i, a_{i,j}) = \{q_f\}$.

Finally, define

$$\ell := s + 4.$$

Obviously, M and ℓ can be constructed from G and k in polynomial time. Observe that, as C is a set, and hence does not contain $c_i = c_j$ with $i \neq j$, M is a minimal DFA for $L(M)$.

We now show that,

(a) if C has a 2-separable normal basis of size at most s, then 2-MINIMAL DISJOINT UNION is true for (M, ℓ); and

(b) if K-MINIMAL DISJOINT UNION is true for (M, ℓ) for any $K \geq 2$, then C has a 2-separable normal basis of size at most s.

This proves the lemma, since a disjoint union of two DFAs can also be seen as a disjoint union of K DFAs where $K - 2$ DFAs have an empty state set.

(a) Assume that C has a normal set basis of size s. Hence, C has a 2-separable normal set basis of size s. We need to show that there exist two DFAs M_1 and M_2 such that

(1) $L(M) = L(M_1) \cup L(M_2)$; and

(2) $L(M_1) \cap M(A_2) = \emptyset$; and

(3) $|M_1| + |M_2| \leq \ell$,

where $\ell = s + 4$.

Thereto, let $B = \{r_1, \ldots, r_s\}$ be the 2-separable normal set basis of C of size s. Also, let B_1 and B_2 be disjoint subsets of B such that each element of C is either an element of B_1, an element of B_2, or a disjoint union of an element of B_1 and an element of B_2.

To describe M_1 and M_2, we first fix the representation of each set c in C as a disjoint union of the basic sets in B. Say that each basic member of B in this representation *belongs to c*.

We define the state sets Q_1 and Q_2 of M_1 and M_2 as

$$Q_1 = \{q_0^1, q_f^1\} \cup \{r_i \in B_1\}$$

and

$$Q_2 = \{q_0^2, q_f^2\} \cup \{r_i \in B_2\},$$

respectively. The transition functions δ_1 and δ_2 of M_1 and M_2 are defined as follows. For every $i = 1, \ldots, n$, $j = 1, \ldots, s$, and $x = 1, 2$,

- $\delta_x(q_0^x, c_i)$ contains r_j, if $r_j \in B_x$ and r_j belongs to c_i; and
- $\delta_x(r_j, a)$ contains q_f^x, if $r_j \in B_x$ and $a \in r_j$.

Notice that the sum of the sizes of M_1 and M_2 is $|B| + 4 = s + 4 = \ell$, which fulfills condition (3). By construction, we have that $L(M_1) \cup L(M_2) = L(M)$, which fulfills condition (2).

We argue that M_1 is deterministic (M_2 follows analogously). By construction, M_1 has only one start state, and all transitions going to its final state are deterministic. Hence, it remains to show that the sets of the form $\delta_1(q_0^1, c_i)$ are singletons. Towards a contradiction, assume that $\delta_1(q_0^1, c_i)$ contains the elements r_j and $r_{j'}$ with $j \neq j'$. But this means that both r_j and $r_{j'}$ belong to c_i, which contradicts the definition of B_1.

We still have to show that $L(M_1) \cap L(M_2)$ is empty. Towards a contradiction, assume that the string $c_i a$ is in $L(M_1) \cap L(M_2)$. Let r_j (respectively, $r_{j'}$) be the state that M_1, (respectively, M_2) reaches after reading c_i. By construction of M_1 and M_2, we have that $j \neq j'$. But this means that both r_j and $r_{j'}$ belong to c_i, and their intersection contains a, which contradicts that B is a normal set basis.

(b) Assume that $L(M)$ can be accepted by a disjoint union of the DFAs $M_1, \ldots, M_{K'}$ with $K' \leq K$, the sum of the sizes of $M_1, \ldots, M_{K'}$ is at most ℓ, and for every $i = 1, \ldots, K'$, $L(M_i) \neq \emptyset$. We can assume that every $M_i = (Q_i, \Sigma_M, \delta_i, I_i, F_i)$ is minimal. We need to show that there exists a normal basis for C of size at most $s = \ell - 4$.

Recall that we assumed that $s < n - 3$. Hence, we have that $\ell = s + 4 < n + 3 = |M| - 1$. As we observed that M is a minimal DFA for $L(M)$, it must be the case that $K' \geq 2$.

Let, for every $i = 1, \ldots, K'$, q_0^i and q_f^i be the initial and final state of M_i, respectively. Since $M_1, \ldots, M_{K'}$ accept a finite set of strings of length 2, we can divide the union of the state sets of $M_1, \ldots, M_{K'}$ into three sets $\mathcal{Q}_0$, $\mathcal{Q}_1$, and $\mathcal{Q}_2$ such that the only transitions in M_i are from $\mathcal{Q}_0$ to states in $\mathcal{Q}_1$ and from states in $\mathcal{Q}_1$ to states in $\mathcal{Q}_2$. For each state $q \in \mathcal{Q}_1$, define a set $B_q = \{a \mid \delta_i(q, a) = \{q_f^i\}, 1 \leq i \leq K'\}$.

As $K' \geq 2$, we have that the size of $B = \{B_q \mid q \in \mathcal{Q}_1\}$ is at most $\ell - 4$. We show that the collection B is also a normal basis of C.

By definition of $L(M)$, we have that every $c \in C$ is the union of $B_c := \{B_q \mid \delta_i(q_0^i, c) = \{q\}\}$. It remains to show that B_c is also a disjoint subcollection of B. When B_c contains only one set, there is nothing to prove. Towards a contradiction, assume that B_c contains two different sets B_{q_1} and B_{q_2} such that $a \in B_{q_1} \cap B_{q_2}$. As every M_i is deterministic, we have that $q_1 \in Q_{i_1}$ and $q_2 \in Q_{i_2}$ with $i_1 \neq i_2$. But this means that $ca \in L(M_{i_1}) \cap L(M_{i_2})$, which contradicts that $M_1, \ldots, M_{K'}$ is a disjoint union.

Hence, B is a normal basis of C. □

It remains to prove Claim 10.9.

Proof of Claim 10.9. *$L(M)$ does not contain interchangeable symbols.*

Proof. Recall that M accepts a language $\{ca \mid c \in C \text{ and } a \in c\}$ of strings of length 2, for a collection of sets C. We denote by E the set $\{a \mid c \in C \text{ and } a \in c\}$ of elements of sets in C.

By definition of $L(M)$, we have that the alphabet C that we use for the letters of the first position is disjoint from the alphabet E that we use for the letters of the second position. Hence, symbols from C are never interchangeable with symbols from E.

We prove the remaining cases by contraposition:

- Suppose that c_1 and c_2 are different elements from C and that c_1 and c_2 are interchangeable. By definition of $L(M)$, this means that c_1 and c_2 contain precisely the same elements, which contradicts that they are different elements from C.

- Suppose that a_1 and a_2 are different elements from E an that a_1 and a_2 are interchangeable. By definition of $L(M)$ this means that a_1 is contained in precisely the same sets as a_2. But this means that C contains obsolete symbols, which contradicts that we chose (C, s) in a set $\mathbf{I}$ satisfying the conditions in Proposition 10.10. □

The following proposition is the counterpart of Proposition 10.5 for the MINIMAL DISJOINT UNION problem.

Proposition 10.10. *There exists a set of inputs $\boldsymbol{J}$ for* MINIMAL DISJOINT UNION, *such that*

*(1) for each $(M, \ell) \in \boldsymbol{J}$, K-*MINIMAL DISJOINT UNION *is* NP*-complete for inputs in $\boldsymbol{J}$; and,*

*(2) for each $(M, \ell) \in \boldsymbol{J}$, (M, ℓ) has a solution for K-*MINIMAL DISJOINT UNION *if and only if (M, ℓ) has a solution for 2-*MINIMAL DISJOINT UNION.

Moreover, $L(M)$ does not contain interchangeable symbols and $\ell < |M| - 1$.

Proof. The set $\mathbf{J}$ is obtained by applying the reduction in Lemma 10.8 to the inputs $\mathbf{I}$ of NORMAL SET BASIS in Proposition 10.5. The correctness of conditions (1) and (2) then immediately follows from conditions (a) and (b) in the proof of Lemma 10.8, and the fact that a 2-MINIMAL DISJOINT UNION is also a K-MINIMAL DISJOINT UNION for every $K > 2$, in which $K - 2$ DFAs have an empty state set. It follows immediately from Claim 10.9 that $L(M)$ does not contain interchangeable symbols. The size constraint is obtained by observing that, in the proof of Lemma 10.8, we assumed that $s < n - 3$, which implied that $\ell < |M| - 1$. □

We are now ready to prove the main result of the present section.

Theorem 10.11. *DTA* MINIMIZATION *is* NP*-complete.*

Proof. The upper bound follows from Corollary 10.2. Given a DTA A and an integer m, the NP algorithm guesses an automaton B of size at most m and verfies in PTIME whether it is equivalent to A.

For the lower bound, we reduce from 2-MINIMAL DISJOINT UNION. Given a DFA $M = (Q_M, \Sigma_M, \delta_M, I_M, F_M)$ and integer ℓ, we construct a DTA A and an integer m such that A has an equivalent DTA of size m if and only if M can be written as a disjoint union of DFAs for which the size does not exceed ℓ. Intuitively, we construct A such that it accepts the trees of the form $r(w)$, where the root node is labeled with a special symbol $r \notin \Sigma_M$ and the string w is in $L(M)$.

According to Proposition 10.10, we can assume without loss of generality that $(M, \ell) \in \mathbf{J}$, which implies that $\ell = |M| - 1$ and that $L(M)$ does not contain interchangeable symbols.

We define $A = (Q_A, \Sigma_A, \delta_A, F_A)$ formally as follows. The set Σ_A is $\{r\} \uplus \Sigma_M$. Its state set Q_A is $\{q_r\} \uplus \Sigma_M$, and its set of final states $F_A = \{q_r\}$. For every $a \in \Sigma_M$, we define the transition $\delta_A(a, a) = \{\varepsilon\}$. We also define $\delta_A(q_r, r) = L(M)$. Finally, let $m = 2 + 2|\Sigma_M| + \ell$. Obviously, A and m can be constructed in polynomial time from (M, ℓ). We now show that

K-MINIMAL DISJOINT UNION is true for (M, ℓ) for any $K \geq 2$
if and only if $L(A)$ can be accepted by a DTA of size m.

($\Rightarrow$) Suppose that K-MINIMAL DISJOINT UNION is true for (M, ℓ) for any $K \geq 2$. According to Proposition 10.10, there exist DFAs M_1 and M_2 such that

(1) $L(M) = L(M_1) \cup L(M_2)$; and

(2) $L(M_1) \cap M(A_2) = \emptyset$; and

(3) $|M_1| + |M_2| \leq \ell$.

We construct a DTA $B = (\Sigma_A, Q_B, \delta_B, F_B)$ as follows. Its state set Q_B is $\Sigma_M \uplus \{r_1, r_2\}$. The set of accept states F_B is $\{r_1, r_2\}$. Finally, the transition function is defined as follows:

- $\delta_B(r_1, r) = L(M_1)$;
- $\delta_B(r_2, r) = L(M_2)$; and

- $\delta_B(a,a) = \{\varepsilon\}$ for every $a \in \Sigma_M$.

Obviously, $L(B) = L(A)$. The size of B is

$$\begin{aligned}|B| &= |Q_B| + |M_1| + |M_2| + \sum_{a\in\Sigma_M} 1\\ &= \ell + |\Sigma_M| + 2 + |\Sigma_M|\\ &= 2 + 2|\Sigma_M| + \ell\\ &= m\end{aligned}$$

($\Leftarrow$) Suppose that there exists a DTA $B = (\Sigma_A, Q_B, \delta_B, F_B)$ for $L(A)$ of size at most $m = 2 + 2|\Sigma_M| + \ell$. We state the following claims (which we prove later):

Claim 10.12. *B has at least $|\Sigma_M|$ non-accepting states.*

As B is bottom-up deterministic and only accepts trees of depth two, Claim 10.12 induces a bijection ϕ between states of B and Σ_M-symbols: for every state $q \in Q_B$, $\phi(q)$ is the unique symbol $a \in \Sigma_M$ such that $\delta_B(a,a) = \{\varepsilon\}$. We denote the homomorphic extension of ϕ also by ϕ.

Claim 10.13. *B has at least two accepting states.*

Let $r_1, \ldots, r_x$ be the accepting states of B, where $x > 1$. Let for every $i = 1, \ldots, x$, M_i' be the minimal DFA such that $\delta_B(r_i, r) = L(M_i')$. It is easy to see that from each M_i', an automaton M_i'' can be constructed which is of the same size and accepts $\phi(L(M_i'))$. Moreover, since B is bottom-up deterministic, the languages $L(M_i')$ are pairwise disjoint. As ϕ is bijective, the languages $\phi(L(M_i'))$ are also pairwise disjoint. The total size of $\sum_{i=1}^{x} |M_i''|$ is $m - 2|\Sigma_M| - x \leq \ell$. Hence, 2-MINIMAL DISJOINT UNION for (M, ℓ) is true. According to Proposition 10.10, we also have that K-MINIMAL DISJOINT UNION is true for (M, ℓ) for every $K \geq 2$. □

It remains to prove Claims 10.12 and 10.13.

Proof of Claim 10.12. *B has at least $|\Sigma_M|$ non-accepting states*

Proof. First observe that $L(B)$ contains only trees of depth two. We say that B *assigns* a state $q \in \text{states}(B)$ to a label $a \in \Sigma_M$ if $\delta_B(q,a) = \varepsilon$.

We first argue that, B assigns only non-accepting states to labels in Σ_M. Indeed, should B assign an accepting state to some $a \in \Sigma_M$, then the tree a, which has depth one, should be in $L(B)$, which is a contradiction.

We now show that B needs at least $|\Sigma_M|$ *different* non-accepting states to assign to the leaves. Towards a contradiction, suppose that B uses lesser than $|\Sigma_M|$ non-accepting states. As B is bottom-up deterministic, there exist two alphabet symbols a and b to which B assigns the same state q in every successful run of B. However, this means that, for every two Σ_M-strings u and v, we have that $uav \in L(M) \Leftrightarrow ubv \in L(M)$. This contradicts that $L(M)$ does not contain interchangeable symbols, which was shown in Proposition 10.10. □

Proof of Claim 10.13. *B has at least two final states.*

Proof. We recall that $|A| = 1 + 2|\Sigma_M| + |M|$ and $|B| \leq 2 + 2|\Sigma_M| + \ell$, so $|B| < |A|$. Towards a contradiction, suppose that B has only one accepting state q_f. Then B has exactly one transition of the form $\delta_B(q_f, r) = L(M')$, where M' is a DFA accepting $\phi^{-1}(L(M))$. However, as M' accepts a language isomorphic to $L(M)$, and M is a minimal automaton, the size of M' is at least $|M|$. But this means that the size of B is at least $1 + 2|\Sigma_M| + |M|$, which is a contradiction. □

10.4 Solutions for Efficient Minimization

As we have shown, DTA minimization is unfeasible even when the horizontal languages are represented by DFAs. The problem is raised when using multiple rules for the same label, for recognizing these horizontal regular languages.

Three alternative notions of bottom-up deterministic tree automata for unranked trees were proposed recently, each of them yielding a solution to the problem. The contribute different notions of automata and bottom-up determinism for unranked trees, which lead to unique minimal automata and polynomial time minimization. However, as we will see in this section, they do not lead to minimal automata of the same size.

First, *stepwise tree automata* [CNT04] are an algebraic notion of automata for unranked trees which also correspond to automata over binary trees by means of a binary encoding. Second, *parallel tree automata (PTAs)* alter the rule format of NTAs and have been independently proposed in [RB04] and [CLT05]. Third, one can use tree automata that operate on the standard *first-child next-sibling encoding* of unranked into binary trees (see, for example, [FGK03]).

10.4.1 Stepwise Tree Automata

Stepwise tree automata have been introduced as an algebraic notion of automata for unranked trees [CNT04]. In this section, we show that regular unranked tree languages are recognized by unique minimal deterministic stepwise tree automata, and formulate the corresponding Myhill-Nerode property.

Definition 10.14. A *stepwise tree automaton (STA)* for unranked trees over Σ is a 5-tuple $A = (Q, \Sigma, \delta, (I^a)_{a\in\Sigma}, F)$ where Q is the set of states, Σ is the alphabet, $\delta : Q \times Q \to Q$ is the transition function, $I^a \subseteq Q$ is a set of initial states for each $a \in \Sigma$, and $F \subseteq Q$ is a set of final states. Moreover, for each $a \in \Sigma$, $A_a = (Q, Q, \delta, I^a, F)$ is a nondeterministic finite automaton. ◇

We call A *(bottom-up) deterministic* if every finite automaton A_a is a DFA.

A *run* of an STA A on an unranked tree t is a function $\lambda : \text{Nodes}(t) \to Q$ such that, for every node $u \in \text{Nodes}(t)$ labeled with a and with n children $u1, \ldots, un$, it holds that

$$\lambda(u) \in \delta^*(q, \lambda(u1) \ \cdots \ \lambda(un))$$

for some $q \in I^a$, where δ^* denotes the homomorphic extension of δ to strings. In other words, the state of a node is computed by running the NFA A_a, with initial states I^a determined by the label of the node, on the sequence of children states. A

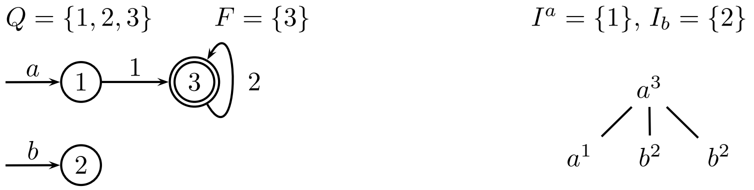


Figure 10.6: An STA $(Q, \{a, b\}, \delta, (I^a)_{a \in \Sigma}, F)$ recognizing $a(ab^*)$ and one of its successful runs. Initial states for a are pointed to by arrows labeled by a. The state 3 of the root is obtained by running the automaton with intial states for a on the word 122.

run is *accepting* if the root is labeled by an accepting state, that is, $\lambda(\varepsilon) \in F$. A tree t is *accepted* if there is an accepting run of A on t. The set of all accepted trees is denoted by $L(A)$ and is called the *language defined by* A.

An STA A is *unambiguous* if, for every tree $t \in L(A)$, A has a unique accepting run on t. An example for a deterministic STA is given in Figure 10.6.

In the present perspective, it is not very clear that STAs can be determinized without altering the language of unranked trees it recognizes, and that all regular languages of unranked trees are recognized by a unique minimal deterministic stepwise tree automaton (up to isomorphism).

Thereto, we observe in the following section that STAs are in fact traditional tree automata over a binary encoding of unranked trees. In order to differentiate between the unranked tree language and the binary tree language a stepwise automaton defines, we write $L^u(A)$ for the language of unranked trees recognized by A.

Curried Binary Encoding

We can identify stepwise tree automata with binary tree automata that operate on Curried binary encodings of unranked trees. While Definition 10.14 provides the clearest way to present stepwise tree automata in examples, the present characterization is mostly more convenient in proofs. It allows to carry over results directly from the theory of traditional tree automata.

We consider the binary alphabet $\Sigma_@ = \Sigma \uplus \{@\}$ in which all labels in Σ have rank zero and @ has rank two. The idea of the Curried encoding is to identify an unranked tree with a lambda term. The tree $a(b\ c\ d)$, for instance, designs the application of function a to the arguments b, c, d. Its Curried encoding $(((a@b)@c)@d)$ applies function a to the same arguments, but one by one (which is the reason for the term *stepwise*). Formally, we define the Curried encoding $\text{curry}(t)$ of an unranked tree t as follows:

(i) $\text{curry}(a) = a$;

(ii) $\text{curry}(a(t_1 \cdots t_n)) = @(\text{curry}(a(t_1 \cdots t_{n-1}))\ \text{curry}(t_n))$

Every STA $A = (Q, \Sigma, \delta, (I^a)_{a \in \Sigma}, F)$ is isomorphic to a binary tree automaton $\text{BTA}(A) =$

$$\begin{array}{ll} Q = & \{1,2,3\} \\ \text{transitions}: & \{a \to 1, \quad @(1,1) \to 3, \\ & \;\; b \to 2, \quad @(3,2) \to 3 \;\} \\ F = & \{3\} \end{array}$$

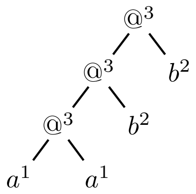


Figure 10.7: The STA in Figure 10.6 with unranked language $a(ab^*)$ as standard tree automaton (Q, Σ, δ, F) on Curried encodings.

$(Q, \Sigma_@, \delta_b, F)$, whose states are those of A. We identify the transitions as follows:

$$\begin{array}{lll} q_1 \overset{q_2}{\to} q \text{ in } A & \text{is identified with} & @(q_1, q_2) \to q \text{ in BTA}(A) \\ q \in I^a \text{ in } A & \text{is identified with} & a \to q \text{ in BTA}(A) \end{array}$$

In the remainder of the chapter, we will therefore identify the STA A with BTA(A). For an STA A, we use the notation $L^b(A)$ to refer to the binary tree language accepted by BTA(A). The binary tree language $L^b(A)$ of a stepwise tree automaton A over Σ is the language recognized by the corresponding tree automaton for binary trees over $\Sigma_@$.

Proposition 10.15. *For every STA A, curry$(L^u(A)) = L^b(A)$. Furthermore, A is deterministic if and only if it is deterministic as a traditional tree automaton on binary trees.*

Proof. Let $A = (Q, \Sigma, \delta_A, (I^a)_{a \in \Sigma}, F)$ be an STA. Let, for every $a \in \Sigma$, A_a be the NFA $(Q, Q, \delta_{A_a}, I^a, F)$, and let BTA$(A) = (Q, \Sigma, \delta_b, F)$. Notice that $\delta_A = \delta_{A_a}$, but, for clarity later in the proof, we denote the transition functions differently. We show for all unranked trees t over Σ that $\delta_A^*(t) = \delta_b^*(\text{curry}(t))$. The proof is by induction on the structure of unranked trees.

For the base case, let $t = a$. Then $q \in \delta_A^*(t)$ if and only if $q \in \delta_{A_a}^*(p, \varepsilon)$ for some $p \in I^a$, if and only if $q \in I^a$, if and only if $a \to q$ is a transition in BTA(A), if and only if $q \in \delta_b^*(t) = \delta_b^*(\text{curry}(t))$. For the inductive case, we assume $t = a(t_1 \cdots t_n)$. It then holds that $q \in \delta_A^*(t)$ if and only if $q \in \delta_{A_a}^*(p, \delta_A^*(t_1) \cdots \delta_A^*(t_n))$ for some $p \in I_a$. By induction, this is equivalent to $q \in \delta_{A_a}^*(p, \delta_b^*(\text{curry}(t_1)) \cdots \delta_b^*(\text{curry}(t_n)))$, which holds if and only if $q \in \delta_b^*(@(\cdots @(a \text{ curry}(t_1)) \cdots) \text{ curry}(t_n))$ given the correspondence of the automaton rules. By definition of the Curried encoding, the latter is equivalent to $q \in \delta_b^*(\text{curry}(a(t_1 \cdots t_n)))$. □

As a consequence, we can deteminize every stepwise tree automaton seen as a traditional tree automaton, without changing its languages of unranked trees.

Theorem 10.16. *Every regular language of unranked trees is recognized by an up to isomorphism unique minimal deterministic STA. Minimization of determinstic STAs is in* PTIME.

Proof. It is well-known that every regular unranked tree language can be recognized by an STA. STAs can be determinized as binary tree automata without changing the unranked tree language. The minimal deterministic STA for a language of unranked trees L is the minimal bottom-up deterministic BTA for the binary tree language curry(L). This follows from Proposition 10.15. It can be computed by the usual algorithm for minimizing traditional tree automata. □

Myhill-Nerode Property

Myhill and Nerode characterized regular languages in terms of congruences induced by the language, proved the existence of minimal deterministic automata for regular languages, and characterized such automata in terms of the congruence.

The Myhill-Nerode property holds generally for algebraic notions of automata (see, for example, [Cou89]) and thus for finite automata over strings, traditional tree automata [Koz92, TW68], and stepwise tree automata [CNT04]. A Myhill-Nerode inspired theorem for NTAs was shown in Theorem G in [BKMW01]. Remarkably, this theorem does not lead to minimal automata. Another Myhill-Nerode inspired theorem for tree automata for unranked trees was shown by Thomas et al. [CLT05], which we treat in Section 10.4.2.

In this section, we formulate the Myhill-Nerode theorem for stepwise tree automata on unranked trees, by translating the Myhill-Nerode theorem for traditional tree automata for binary trees via Currying. Our main motivation for discussing the Myhill-Nerode theorem is that the present version has the advantages of the two other Myhill-Nerode inspired theorems, while not sharing their disadvantages: *(i)* it leads to unique minimal deterministic automata, which can be computed in PTIME, *(ii)* it uses a single, natural congruence relation, and *(iii)* it allows to carry over the minimization algorithm directly from traditional tree automata. Moreover, we show later that it leads to the smallest minimal deterministic automata, when compared to the *parallel tree automata*from [CLT05, RB04] and to traditional tree automata over the standard first-child next-sibling encoding (Sections 10.4.2 and 10.4.3).

A *binary context* C is a function mapping binary trees to binary trees. A context can be represented by a *pointed binary tree*, that is, a binary tree over the alphabet $\Sigma \uplus \{\bullet\}$ that contains a single occurrence of the symbol "$\bullet$" which we call the *hole marker*. The hole marker is always at a leaf. Context application $C[t]$ to a binary tree t replaces the hole marker in C by t.

An *unranked context* C is a tree over the unranked alphabet $\Sigma \uplus \{\bullet\}$ that contains a single occurrence of the hole marker, but this time possibly labeling an internal node. Given an unranked context C and an unranked tree $t = a(t_1 \cdots t_n)$, we define *context application* $C[t]$ inductively as follows:

(i) $\bullet(t'_1 \cdots t'_m)[a(t_1 \cdots t_n)] = a(t_1 \cdots t_n t'_1 \cdots t'_m)$

(ii) $a(t'_1 \cdots t'_i \cdots t'_m)[t] = a(t'_1 \cdots t'_i[t] \cdots t'_m)$ where t'_i contains the hole marker.

We claim that the unranked contexts and context applications that we defined are precisely the Curried versions of the binary contexts.

Lemma 10.17. *If C is an unranked context and t is an unranked tree, we have that curry$(C[t])$ = curry(C)[curry(t)].*

The proof is by straightforward induction on the structure of contexts.

The following definitions are parametric, in that they apply to unranked trees as well as to binary trees. A *congruence* on trees is an equivalence relation $\equiv$ such that, for every context C, if $t_1 \equiv t_2$ then $C[t_1] \equiv C[t_2]$. We refer to the number of equivalence classes of an equivalence relation as the *index* of the equivalence relation. An equivalence relation is of *finite index* when there are only a finite number of equivalence classes. Given a tree language L, we define the congruence $\equiv_L$ induced by L through:

$$t_1 \equiv_L t_2 \text{ if and only if for every context } C\text{: } C[t_1] \in L \Leftrightarrow C[t_2] \in L.$$

Theorem 10.18 (Myhill-Nerode)**.** *For any binary or unranked tree language L it holds that L is a regular tree language if and only if its congruence $\equiv_L$ has finite index. Furthermore, there exists an (up to isomorphism) unique minimal bottom-up deterministic (stepwise) tree automaton for all regular languages L. The size of this automaton is equal to the index of $\equiv_L$.*

The proof of this theorem is immediate from the binary case [Koz92] and Lemma 10.17.

10.4.2 Parallel Tree Automata

Parallel tree automata are automata for unranked trees which have been independently proposed in [RB04] and [CLT05] for efficient minimization. In this section, we compare parallel NTAs and stepwise tree automata with respect to the size of minimal deterministic automata and their Myhill-Nerode theorems.

The idea of parallel tree automata is to start with an NTA and to merge all its NFAs for the same alphabet symbol into one NFA. When applied to DTA(DFA)s, this solves the main reason why efficient minimalization fails. In order to distiguish final states of different NFAs after the merge, an explicit output function is added. It should be noted that, in the original papers, parallel tree automata are simply called "tree automata"[CLT05, RB04]. We have given them the name *parallel* tree automata as they can be seen as NTAs in which the finite automata for an alphabet symbol are executed in parallel.

Definition 10.19. A *nondeterministic parallel tree automaton (NPTA)* is a tuple $A = (Q, \Sigma, (A^a)_{a\in\Sigma}, F, o)$ where Q is a final set of states, every $A^a = (Q_a, Q, \delta_a, I_a, F_a)$ is an NFA, $F \subseteq Q$ is the set of final states, and $o : \cup_{a\in\Sigma} F_a \to Q$ is an output function. $\diamond$

A *run* λ of an NPTA A on an unranked tree t over Σ is a function $\lambda : \text{Nodes}(t) \to Q$ such that, for every $u \in \text{Nodes}(t)$ with label a and n children $u1, \ldots, un$,

$$\lambda(u) \in \delta_a^*\big(q, o(\lambda(u1)) \cdots o(\lambda(un))\big)$$

for some $q \in I_a$, where δ_a^* denotes the homomorphic extension of the transition function δ_a. A run λ on t is *accepting* if $o(\lambda(\varepsilon)) \in F$. A tree t is *accepted* by A if there exists an accepting run of A on t. We denote set of trees accepted by A by $L(A)$.

$$Q = \{1', 2', 3'\} \qquad F = \{3'\} \qquad o(1) = 1',\ o(2) = 2',\ \text{and } o(3) = 3'$$

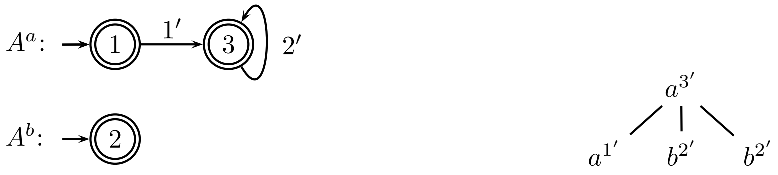


Figure 10.8: A DPTA $(Q, \Sigma, (A^a)_{a\in\Sigma}, F, o)$ for $a(ab^*)$ and one of its runs. The corresponding STA is given in Figure 10.7.

A *(bottom-up) deterministic PTA (DPTA)* is an NPTA for which every A^a is a *DFA*. An *unambiguous PTA (UPTA)* is an NPTA A, such that, for every $t \in L(A)$, there exists a unique accepting run of A on t.

An example for the minimal DPTA for the language $a(ab^*)$ is given in Figure 10.8.

Although not explicitly stated in [CLT05], we note that it is assumed that the state sets of the automata A^a in PTAs are disjoint. The latter can be concluded from Theorem 10.21.

Theorem 10.20 ([CLT05], see also [RB04])**.** *Every regular language of unranked trees is recognized by a unique minimal DPTA (up to isomorphism). Furthermore, minimization is in* PTIME*.*

Myhill-Nerode Property

Cristau, Löding, and Thomas [CLT05] prove a Myhill-Nerode property for DPTAs. The latter property for DPTAs allows us to compare the size of minimal deterministic PTAs with minimal deterministic STAs.

A *pointed tree* C over Σ is an unranked context over Σ such that the unique node in C that is labeled by "$\bullet$" is a leaf. For a tree language L, the equivalence relation $\sim_L$ is defined as

$$t_1 \sim_L t_2 \text{ if and only if for every pointed tree } C\text{: } C[t_1] \in L \Leftrightarrow C[t_2] \in L.$$

For two trees $t = a(t_1 \cdots t_k)$ and $t' = a(t'_1 \cdots t'_\ell)$, define

$$t \odot t := a(t_1 \cdots t_k t'_1 \cdots t'_\ell).$$

For $a \in \Sigma$, let $\mathcal{T}^a_\Sigma$ denote the set of Σ-trees which have a as their root label. Then, the equivalence relation $\overset{\rightarrow}{\sim}_L$ is defined for all $t_1, t_2 \in \mathcal{T}^a_\Sigma$ by

$$t_1 \overset{\rightarrow}{\sim}_L t_2 \text{ if and only if } \forall t \in \mathcal{T}^a_\Sigma : t_1 \odot t \sim_L t_2 \odot t.$$

Theorem 10.21 (Theorem 1 in [CLT05], rephrased)**.** *For every regular tree language L, the size of the minimal DPTA accepting L is $S_L + \sum_{a\in\Sigma} S^a_L$, where*

- *S_L denotes the number of equivalence classes of the relation $\sim_L$; and,*
- *for each $a \in \Sigma$, S^a_L denotes the number of equivalence classes of the relation $\overset{\rightarrow}{\sim}_L$ in the set $\mathcal{T}^a_\Sigma$.*

Size Comparison with Stepwise Tree Automata

We are now ready to show the following proposition:

Proposition 10.22. *For every regular tree language L, the size of the minimal deterministic STA accepting L is at most the size of the minimal deterministic PTA. Moreover, the minimal deterministic PTA is at most quadratically larger than the minimal deterministic STA.*

Proof. Let L be a regular tree language. We show that the sum of the indices of the equivalence relations $\sim_L$ and $\overrightarrow{\sim}_L$ is at least as large as the index of the equivalence relation $\equiv_L$.

To this end, assume that there exist two Σ-trees t_1, t_2, such that $t_1 \not\equiv_L t_2$. If $\mathrm{lab}^{t_1}(\varepsilon) \neq \mathrm{lab}^{t_2}(\varepsilon)$, then we immediately have that $t_1 \not\overrightarrow{\sim}_L t_2$, as the relation $\overrightarrow{\sim}_L$ is only defined between trees with the same root label.

Assume that $\mathrm{lab}^{t_1}(\varepsilon) = \mathrm{lab}^{t_2}(\varepsilon) = a$. Hence, we can assume without loss of generality that there exists a context C such that $C[t_1] \in L$, while $C[t_2] \notin L$ (the symmetric case is analogous). Let $u \in \mathrm{Nodes}(C)$ be the unique node such that $\mathrm{lab}^C(u) = \bullet$. We now define the following trees:

(a) Let C' be the tree obtained from C by removing all subtrees of the node u. Hence, u is a leaf in C'.

(b) Let t' be the tree obtained by taking the subtree of C rooted at u, and relabeling the root with a (instead of $\bullet$).

We now claim that $t_1 \not\overrightarrow{\sim}_L t_2$. Indeed, we have that $C'[t_1 \odot t'] = C[t_1] \in L$, while $C'[t_2 \odot t'] = C[t_2] \notin L$, which shows that $t_1 \odot t' \not\sim_L t_2 \odot t'$.

It remains to show the quadratic bound on the size increase. Given a deterministic STA $B = (Q_B, \Sigma, \delta_B, (I_B^a)_{a\in\Sigma}, F_B)$, we construct an equivalent deterministic PTA $A = (Q_B, \Sigma, (A^a)_{a\in\Sigma}, F_A, o)$ of size $\mathcal{O}(|\Sigma| \cdot |B|)$, which proves the claim. Thereto, let, for every $a \in \Sigma$, $B_a = (Q_B, Q_B, \delta_B, I_B^a, F_B)$ be the DFA associated to the STA. Intuitively, every DFA A^a is simply a copy of B_a, adapted such that the state sets of the A^a are pairwise disjoint. Formally, let, for every $a \in \Sigma$, $Q_a = \{q_a \mid q \in Q_B\}$. For every $a \in \Sigma$, we define the DFA $A^a = (Q_a, Q_B, \delta_a, I_a, F_a)$ as follows. For every transition $q^1 \xrightarrow{p} q^2$ in B_a, A^a contains the transition $q_a^1 \xrightarrow{p} q_a^2$. Finally, we define $o(q_a) = q$ for every $q_a \in Q_a$. □

Proposition 10.23. *There exists a family of unranked regular tree languages $(L_n)_{n\in\mathbb{N}}$ for which the minimal deterministic PTA is quadratically larger than the minimal deterministic STA.*

Proof. Let Σ_n be the alphabet $\{1, \ldots, n, a\}$ and define the languages

$$L_n := \{j(\underbrace{a \cdots a}_{n}) \mid 1 \leq j \leq n\}.$$

Figure 10.9 shows a deterministic STA $A = (Q_A, \Sigma, (I^a)_{a\in\Sigma}, F)$ of size $\mathcal{O}(n)$ accepting L_n.

$Q_A = \{a, 0, 1, 2, \ldots, n\} \qquad F_A = \{n\}$
$I^a = \{a\} \qquad I^j = \{0\}$ for all $1 \leq j \leq n$

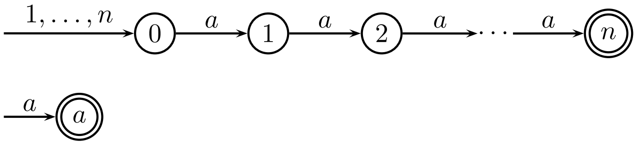


Figure 10.9: Deterministic STA for the language L_n of Proposition 10.23.

Second, we show that the minimal DPTA for L_n has at least n^2 states. Intuitively, the minimal DPTA for L_n needs n different finite string automata (one for each i) with n states each (to accept a language consisting of a single string of length n).

Formally, we argue that the equivalence relation $\overset{\rightarrow}{\sim}_{L_n}$ induces at least n^2 different equivalence classes, which proves the proposition, according to Theorem 10.21. Thereto, suppose that $t_1 = i(a^k)$ and $t_2 = j(a^\ell)$ are two trees with $i, j, k, \ell = 1, \ldots, n$.

- If $i \neq j$, then t_1 and t_2 are clearly in different equivalent classes, because the relation $\overset{\rightarrow}{\sim}_{L_n}$ is only defined between trees with the same root.
- If $i = j$ and $k \neq \ell$, then let t be the tree $i(a^{n-k})$. Then we have that $t \odot t_1 \in L_n$ while $t \odot t_2 \notin L_n$. If we take the context $C = \bullet$, we have that $C[t_1] \in L_n$ and $C[t_2] \notin L_n$. Hence, t_1 is in a different equivalence class that t_2.

It follows that the relation $\overset{\rightarrow}{\sim}_{L_n}$ induces at least n^2 different equivalence classes. □

10.4.3 Standard Binary Encoding

Another approach towards efficient minimization for automata representing unranked tree languages is to use the *first-child next-sibling* encoding [FGK03, Nev02, Suc01].

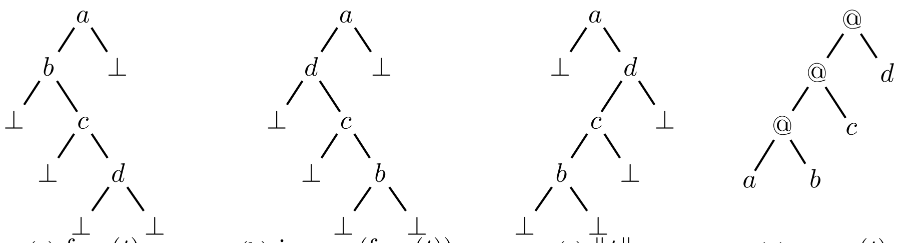
The first-child next-sibling enoding fcns(t) of some unranked tree t over Σ is a binary tree over the alphabet $\Sigma_\perp = \Sigma \uplus \{\perp\}$, where the first-child relation is associated with the first position, and the next-sibling relation with the second position.

The idea of using the first-child next-sibling encoding for minimization, is to represent a regular language of unranked trees L by a minimal DBTA for the language of its binary encoding fcns(L), similarly as we did in Section 10.4.1 for STAs that recognize the binary tree language curry(L).

Inversion

The goal of this section is to compare the size of the DBTAs for fcns(L) and curry(L) for regular languages of unranked trees L. Figure 10.10 illustrates these two binary encodings and two others at the example of the unranked tree $t = a(bcd)$.

The first important difference between fcns(L) and curry(L) is that sequences of children are inverted. When traversing fcns(t) bottom-up, the sequence bcd is encountered in inverted order dcb, while it occurs in the original order in curry(t).



(a) fcns(t) (b) inverse(fcns(t)) (c) $[\![t]\!]$ (d) curry(t)

Figure 10.10: Four binary encodings of the unranked tree $t = a(bcd)$: first-child next-sibling fcns(t), inverted first-child next-sibling inverse(fcns(t)), previous-sibling last-child $[\![t]\!]$, and the Curried encoding curry(t).

It is well known for minimal DFAs that language inversion leads to an exponential blow-up of the minimal size. As a consequence, there is in general an exponential blow-up between the minimal DBTAs for fcns(L) and curry(L) in both directions. This holds, for instance, for the tree languages $L_n = \{c(w) \in \mathcal{T}_\Sigma \mid w \in (a+b)^n a(a+b)^*\}$ when going from minimal DBTAs for curry(L_n) to fcns(L_n), where $n \in \mathbb{N}$. For the translation in the other direction, the exponential blow-up occurs for the tree languages $L'_n = \{c(w) \in \mathcal{T}_\Sigma \mid w \in (a+b)^* a(a+b)^n\}$.

We wish to ignore such succinctness differences due to inversion. Actually, we wish to compare the *previous-sibling last-child encoding* $[\![.]\!]$ (which is defined formally later) with Currying. The previous-sibling last-child encoding $[\![.]\!]$ is equal to the inverted first-child next-sibling encoding, except that the first and second child are switched for all nodes. The reason for turning to this previous-sibling last-child encoding is because it facilitates and improves the readability of constructions later in the chapter. Of course, switching every first and second child in binary trees has no effect on the size of minimal DBTAs. We illustrate an example of the discussed encodings in Figure 10.10.

The main difference that remains between the previous-sibling last-child encoding $[\![t]\!]$ and curry(t) in the above example, is that t's root's label a is located at the root of $[\![t]\!]$, while it is found in the leftmost leaf of curry(t). In bottom-up processing, one sees leafs first, so the Curried encoding should have advantages for minimization.

Size Comparison to Stepwise Tree Automata

We show that minimal deterministic STAs for languages L of unranked trees are at most quadratically larger than DBTAs for the previous-sibling last-child encoding $[\![L]\!]$, and that the blow-up is exponential in the other direction.

Let us define the *previous-sibling last-child encoding* $[\![t]\!]$ of some unranked tree t of Σ more formally. It is a binary tree over the alphabet $\Sigma_\perp = \Sigma \uplus \{\perp\}$, where the previous-sibling relation is associated with the first position and the last-child relation with the second position:

$$\begin{array}{rcl} [\![a(t_1 \cdots t_n)]\!] & := & [\![\langle a(t_1 \cdots t_n)\rangle]\!] \\ [\![\langle t_1 \cdots t_n a(s_1 \cdots s_m)\rangle]\!] & := & a([\![\langle t_1 \cdots t_n\rangle]\!][\![\langle s_1 \cdots s_m\rangle]\!]) \\ [\![\langle\rangle]\!] & := & \perp \end{array}$$

In order to relate the language $\llbracket L \rrbracket$ to curry(L), we define a tree transformation “shift” that transforms a tree $\llbracket t \rrbracket$ to curry(t). Intuitively, the transformation processes the tree $\llbracket t \rrbracket$ in a top-down manner and moves the labels of parents (in the unranked tree) downwards. On the example in Figure 10.10(c), it would move the a downwards to obtain the tree in Figure 10.10(d). Formally, the transformation is defined as follows:

$$\begin{array}{rcl} \text{shift}(a(\perp\ t)) & := & \text{shift}_a(t) \\ \text{shift}_a(b(t_1\ t_2)) & := & @(\text{shift}_a(t_1)\ \text{shift}_b(t_2)) \\ \text{shift}_a(\perp) & := & a \end{array}$$

The following simple equality will be useful in the proofs to come:

$$\text{shift}(\llbracket a(t_1 \cdots t_n) \rrbracket = \text{shift}_a(\llbracket \langle t_1 \cdots t_n \rangle \rrbracket) \qquad (\dagger)$$

It holds by definition of the encoding $\llbracket . \rrbracket$ and the shift transformation:

$$\begin{array}{rcl} \text{shift}(\llbracket a(t_1 \cdots t_n) \rrbracket & = & \text{shift}(\llbracket \langle a(t_1 \cdots t_n \rangle \rrbracket) \\ & = & \text{shift}(a(\llbracket \langle \rangle \rrbracket \llbracket \langle t_1 \cdots t_n \rangle \rrbracket) \\ & = & \text{shift}_a(\llbracket \langle t_1 \cdots t_n \rangle \rrbracket) \end{array}$$

Proposition 10.24. *For every unranked tree t over Σ, shift$(\llbracket t \rrbracket) =$ curry(t).*

Proof. The proof is by induction on the structure of unranked trees. The base case $t = a$ is simple: shift$(\llbracket a \rrbracket) = \text{shift}_a(\perp) = a =$ curry(a).

In the induction, we have $t = a(t_1 \cdots t_n b(s_1 \cdots s_m))$, where n and m can be zero, so we can apply the equation $(\dagger)$ and the definitions of $\llbracket . \rrbracket$ and shift$_a$:

$$\begin{array}{rcl} \text{shift}(\llbracket t \rrbracket) & = & \text{shift}_a(\llbracket \langle t_1 \cdots t_n b(s_1 \cdots s_m)\rangle \rrbracket) \\ & = & \text{shift}_a(b(\llbracket \langle t_1 \cdots t_n \rangle \rrbracket\ \llbracket \langle s_1 \cdots s_m \rangle \rrbracket) \\ & = & @(\text{shift}_a(\llbracket \langle t_1 \cdots t_n \rangle \rrbracket)\ \text{shift}_b(\llbracket \langle s_1 \cdots s_m \rangle \rrbracket)) \end{array}$$

We are now in the position to apply the induction hypothesis, and to conclude by the definition of the Curried encoding:

$$\begin{array}{rcl} \text{shift}(\llbracket t \rrbracket) & = & @(\text{curry}(a(t_1 \cdots t_n))\ \text{curry}(b(s_1 \cdots s_m))) \\ & = & \text{curry}(a(t_1 \cdots t_n b(s_1 \cdots s_m))) \qquad \square \\ & = & \text{curry}(t) \end{array}$$

Our next goal is to encode NBTAs over $\Sigma_\perp$ into NBTAs over $\Sigma_@$ that recognize the shifted language. The size should grow no more than quadratically and bottom-up determinism should be preserved.

The idea of the automata conversion is to memorize node labels that have been shifted down. In bottom-up processing, these labels will be seen earlier than needed, so we simply memorize them in the state when moving upwards. Given an NBTA $A = (Q_A, \Sigma_\perp, \delta_A, F_A)$ over $\Sigma_\perp$, we define an NBTA $B = (Q_B, \Sigma_@, \delta_B, F_B)$ over $\Sigma_@$ such that $Q_B = Q_A \times \Sigma$. We write $p[a]$ for pairs (p, a) where $p \in Q_A$ and $a \in \Sigma$, to stress that a is the label being remembered. Note that $|B| = |\Sigma| \cdot |A|$, so the size increases at most quadratically. The rules of B are produced by the inference rules in Figure 10.11. Finally, the final states F_B of B are defined as $F_B = \{p[a] \mid \delta^*_A(a(\perp\ p)) \cap F_A \neq \emptyset\}$.

$$S1 \quad \frac{\bot \to p \text{ is a transition of } A \qquad a \in \Sigma}{a \to p[a] \text{ is a transition of } B}$$

$$S2 \quad \frac{b(p_1, p_2) \to p \text{ is a transition of } A \qquad a \in \Sigma}{@(p_1[a], p_2[b]) \to p[a] \text{ is a transition of } B}$$

Figure 10.11: Converting an NBTA A for the previous-sibling last-child encoding of unranked trees into an STA B.

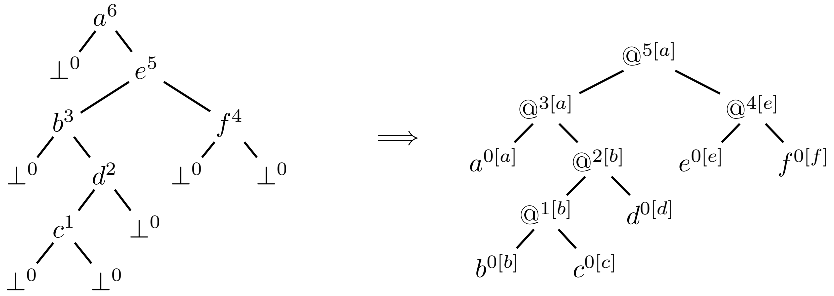


Figure 10.12: A run of some tree automaton A on the previous-sibling last-child encoding of the unranked tree $a(b(c\ d)\ e(f))$ and the corresponding run of B on the Curried encoding.

Here, we denoted by $\delta_A^*(a(\bot\ p))$ the set $\{\delta_A^*(a(\bot\ t)) \mid p \in \delta_A^*(t)\}$.

We illustrate the conversion in Figure 10.12. It presents a run of some automaton A on the previous-sibling last-child encoding of the unranked tree $a(b(cd)\ e(f))$ and the corresponding run of B on the Curried encoding.

In Lemma 10.25 and Proposition 10.26, A is always an NBTA accepting an encoding $\llbracket L \rrbracket$ of an unranked regular tree language $L \subseteq \mathcal{T}_\Sigma$.

Lemma 10.25. *Let t be a binary tree over $\Sigma_\bot$, $A = (Q_A, \Sigma_\bot, \delta_A, F_A)$ a NBTA over $\Sigma_\bot$, and $B = (Q_B, \Sigma_@, \delta_B, F_B)$ be the above defined STA. It then holds for all $p \in Q_A$ and $a \in \Sigma$ that*

$$\delta_B^*(\mathit{shift}_a(t)) = \{p[a] \mid p \in \delta_A^*(t)\}.$$

Proof. By induction on the structure of t. If $t = \bot$ then the lemma follows from the definition of shift_a and inference rule S1:

$$\begin{aligned} p[a] \in \delta_B^*(\text{shift}_a(t)) \quad & \text{if and only if } p[a] \in \delta_B^*(a) \\ & \text{if and only if } a \to p[a] \text{ is a transition of } B \\ & \text{if and only if } \bot \to p \text{ is a transition of } A \\ & \text{if and only if } p \in \delta_A^*(t) \end{aligned}$$

In the inductive case, $t = b(t_1 t_2)$ for some $b \in \Sigma$ and binary trees t_1, t_2 over $\Sigma_\bot$. We first show that $\{p[a] \mid p \in \delta_A^*(t)\} \subseteq \delta_B^*(\text{shift}_a(t))$. To this end, take $a \in \Sigma$ and assume that $p \in \delta_A^*(t)$. We need to show that $p[a] \in \delta_B^*(\text{shift}_a(t))$. As $p \in \delta_A^*(t)$, there exists a transition $b(p_1, p_2) \to p$ of A such that $p_1 \in \delta_A^*(t_1)$ and $p_2 \in \delta_A^*(t_2)$. Since

$b(p_1, p_2) \to p$ is a transition of A, we can apply inference rule S2 of the construction of B in Figure 10.11, implying that, since $a \in \Sigma$, $@(p_1[a], p_2[b]) \to p[a])$ is a transition in B. The induction hypothesis applied to t_1 and t_2 yields that, for every $a_1, a_2 \in \Sigma$, $\delta^*_B(\text{shift}_{a_1}(t_1)) = \{p_1[a_1] \mid p_1 \in \delta^*_A(t_1)\}$ and $\delta^*_B(\text{shift}_{a_2}(t_2)) = \{p_2[a_2] \mid p_2 \in \delta^*_A(t_2)\}$. Hence, we have that

$$p[a] \in \delta^*_B(@(\text{shift}_a(t_1)\ \text{shift}_b(t_2))) = \delta^*_B(\text{shift}_a(t)).$$

For the inclusion $\delta^*_B(\text{shift}_a(t)) \subseteq \{p[a] \mid p \in \delta^*_A(t)\}$, let $a \in \Sigma$ and $p[a] \in \delta^*_B(\text{shift}_a(t))$. By definition of shift_a, we have that $p[a] \in \delta^*_B(@(\text{shift}_a(t_1)\ \text{shift}_b(t_2)))$. By induction, we have $\delta^*_B(\text{shift}_a(t_1)) = \{p_1[a] \mid p_1 \in \delta^*_A(t_1)\}$ and $\delta^*_B(\text{shift}_b(t_2)) = \{p_2[b] \mid p_1 \in \delta^*_A(t_2)\}$. Hence, there exist $p_1, p_2 \in Q_A$ such that $@(p_1[a], p_2[b]) \to p[a]$ is a transition in B. According to inference rule S2, we have that $b(p_1, p_2) \to p$ is a transition of A. Hence, $p \in \delta^*_A(b(t_1 t_2)) = \delta^*_A(t)$. □

Proposition 10.26. *Let $A = (Q_A, \Sigma_\perp, \delta_A, F_A)$ be a NBTA and $B = (Q_B, \Sigma_@, \delta_B, F_B)$ be the above defined STA. Then $L(B) = \textit{shift}(L(A))$.*

Proof. Let $t \in L(A)$ and let $s = \text{shift}(t)$. By definition of the shift function, we have that $t = a(\perp\ t_2)$ for some $a \in \Sigma$ and $t_2 \in b\mathcal{T}_{\Sigma_\perp}$, such that $\text{shift}(t) = \text{shift}_a(t_2)$. Furthermore, there exists a $p \in F_A \cap \delta^*_A(t)$. Let $p_2 \in \delta^*_A(t_2)$ be such that $a(p_1, p_2) \to p$ is a transition of A for some $p_1 \in Q_A$. Notice that $p_2[a] \in F_B$. Lemma 10.25 proves $p[a] \in \delta^*_B(\text{shift}_a(t_2))$. Thus, $s = \text{shift}_a(t_2) \in L(B)$.

For the converse, let $s \in L(B)$. The shift function is one-to-one and onto, so there exists some tree t such that $s = \text{shift}(t)$. It remains to show that $t \in L(A)$. By definition of the shift function, t has the form $a(\perp\ t_2)$ and $s = \text{shift}(t) = \text{shift}_a(t_2)$. There exists a $p[a] \in F_B$ such that $p[a] \in \delta^*_B(\text{shift}_a(t_2))$. By Lemma 10.25, it follows that $p \in \delta^*_A(t_2)$. By definition of F_B it holds that $\delta^*_A(a(\perp\ p)) \cap F_A \neq \emptyset$, where $\delta^*_A(a(\perp\ p))$ denotes $\{\delta^*_A(a(\perp\ t)) \mid p \in \delta^*_A(t)\}$. Thus, $t = a(\perp\ t_2) \in L(A)$ so that $s = \text{shift}(t) \in \text{shift}(L(A))$. □

Theorem 10.27. *For every regular language L of unranked trees over Σ, the size of the minimal DBTA the previous-sibling last-child encoding $\llbracket L \rrbracket$ is at most quadratically smaller that the minimal deterministic STA for L.*

Proof. Let $A = (Q_A, \Sigma_\perp, \delta_A, F_A)$ be the minimal deterministic DBTA recognizing $\llbracket L \rrbracket$. The above defined DBTA $B = (Q_B, \Sigma_@, \delta_B, F_B)$ is deterministic and a factor of $|\Sigma|$ larger than A and recognizes $\text{curry}(L)$:

$$\begin{array}{rcll} L(B) & = & \text{shift}(L(A)) & \text{by Proposition 10.26} \\ & = & \text{shift}(\llbracket L \rrbracket) & \\ & = & \text{curry}(L) & \text{by Proposition 10.24} \end{array}$$

By Proposition 10.15, the minimal deterministic STA recognizing L is thus smaller or equal in size to B, that is, at most a factor of $|\Sigma|$ larger than A. □

We give two examples relating minimal DBTAs with respect to the previous-sibling last-child encoding to minimal deterministic stepwise tree automata. The first example proves that the quadratic construction of Theorem 10.27 is optimal. The second one illustrates that minimal DBTAs over the previous-sibling last-child encodings can be exponentially larger than minimal STAs.

Proposition 10.28. *There exists an infinite class of languages $(L_n)_{n\in\mathbb{N}}$ such that for every L_n, the minimal deterministic STA for L_n is quadratically larger than than the minimal DBTA for $\llbracket L_n \rrbracket$.*

Proof. For every $n \in \mathbb{N}$, we define a tree language L_n such that the minimal STA for L_n is quadratically larger than the minimal DBTA accepting $\llbracket L_n \rrbracket$. Indeed, consider, for every $n \in \mathbb{N}$, the regular tree language $L_n = \{b_i(b_i(\underbrace{a \cdots a}_{n})) \mid 1 \leq i \leq n\}$ over the alphabet $\Sigma_n = \{b_1, \ldots, b_n, a\}$.

The following DBTA $A_n = (Q_n, \Sigma_n \cup \{\perp\}, \delta_n, F_n)$ recognizes $\llbracket L_n \rrbracket$, has $n+2$ states $\{a_1, \ldots, a_n, b_1, \ldots, b_n, \perp, \mathrm{ok}\}$, where ok is the only final state, and the following transitions:

- $\perp \to \perp$;
- $a(\perp\ \perp) \to a_1$;
- $a(a_k\ \perp) \to a_{k+1}$, for every $1 \leq k < n$;
- $b_i(\perp\ a_n) \to b_i$, for every $1 \leq i \leq n$; and,
- $b_i(\perp\ b_i) \to \mathrm{ok}$.

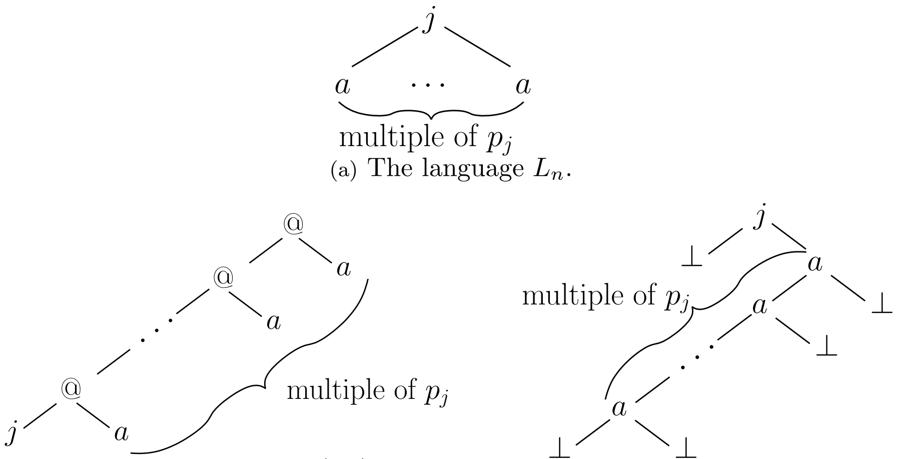
We show that the minimal stepwise automaton for L_n has size at least n^2+n+2. To this end, we apply the Myhill-Nerode Theorem 10.18 for stepwise tree automata. We show that index of $\equiv_{L_n}$ is at least n^2+n+2. It is easy to see that the sets L_n, $\{a\}$, and $\{b_i\}$ for $i = 1, \ldots, n$ form $n+2$ equivalence classes of $\equiv_{L_n}$. Furthermore, consider the trees $t_{i_1,j_1} = b_{i_1}(a^{j_1})$ and $t_{i_2,j_2} = b_{i_2}(a^{j_2})$ for $1 \leq i_1, i_2, j_1, j_2 \leq n$. Suppose that $i_1 \neq i_2$ or $j_1 \neq j_2$, and consider the context $C = b_{i_1}(\bullet(a^{n-j_1}))$. Then we have that $C[t_{i_1,j_1}] \in L_n$, while $C[t_{i_2,j_2}] \notin L_n$. Hence, every t_{i_1,j_1} and t_{i_2,j_2} are in different equivalence classes when $i_1 \neq i_2$ or $j_1 \neq j_2$, which implies that that the index of $\equiv_{L_n}$ is at least n^2+n+2. □

The translation from minimal deterministic stepwise automata to minimal DBTA for the previous-sibling last-child encoding of its language can encoding can be exponential in the worst case, as we show next.

Proposition 10.29. *There exists an infinite class of languages $(L_n)_{n\in\mathbb{N}}$ such that for every L_n, the minimal deterministic STA for L_n is exponentially more succinct than the minimal DBTA for the encoding $\llbracket L_n \rrbracket$.*

Proof. The proof is based on the fact that the smallest DFA for the union of an arbitrary number of DFAs can be exponentially larger than the sum of their sizes (see, for example, [PS02]). Indeed, let A_j to be the minimal DFA accepting $(a^{p_j})^*$, where p_j denotes the j-th prime number. Then, the minimal size of the DFA for $(a^{p_1})^* \cup \cdots \cup (a^{p_n})^*$ is $\prod_{j=1,\ldots,n} p_j$, which is exponentially larger than $\sum_{j=1,\ldots,n} p_j$ when n is arbitrary. The proposition now holds for the tree languages L_n with alphabet $\{1, \ldots, n, a\}$:

$$L_n := \bigcup_{j=1,\ldots,n} \{j(w) \mid w \in L(A_j)\}.$$



(a) The language L_n.

(b) The language curry(L_n).

(c) The language $\llbracket L_n \rrbracket$.

Figure 10.13: Illustration of the languages used in the proof of Proposition 10.29.

We first show that, for every $n \in \mathbb{N}$, there exists a stepwise automaton for L_n of size $\sum_{j=1,\ldots,n} p_j$. Let B_n be the minimal DFA with alphabet $\{1, \ldots, n, a\}$, that accepts the string language $\cup_{j=1}^{n} j(a^{p_j})^*$. The size of B_n is $2 + \sum_{j=1,\ldots,n} p_j$. It can be turned into an STA $S_n = (Q, \Sigma, \delta, (I_a)_{a\in\Sigma}, F)$ for L_n by setting $I_a = a$, $I_j = \delta^*_{B_n}(j)$, removing the initial state of B_n and adding the state a. This results in an STA of size $2 + \sum_{j=1,\ldots,n} p_j$.

It remains to show that the minimal DBTA for $\llbracket L_n \rrbracket$ has size $2 + \prod_{j=1,\ldots,n} p_j$. We show that the index of $\equiv_{\llbracket L_n \rrbracket}$ is at least that large. We only consider equivalence classes that contain a tree t for which there exists a context C such that $C[t] \in L_n$. One equivalence class of $\equiv_{\llbracket L_n \rrbracket}$ consists precisely of the trees in $\llbracket L_n \rrbracket$. Notice that these trees always have some j as their root symbol. A second equivalence class consists of the singleton $\{\perp\}$. The remaining N equivalence classes consist of trees that have their root labeled with a. These equivalence classes are isomorphic to the equivalence classes induced by the minimal DFA for $(a^{p_1})^* \cup \cdots \cup (a^{p_n})^*$. Indeed, let ϕ be the function that maps the binary trees of the form $a(a(\cdots a(\perp \perp) \cdots \perp) \perp)$ (with k occurrences of a) to the string a^k. Then, ϕ is isomorphic. It is easy to see that a set of trees S is an equivalence class of $\equiv_{\llbracket L_n \rrbracket}$ if and only if $\phi(S)$ is an equivalence class of $\equiv_{(a^{p_1})^* \cup \cdots \cup (a^{p_n})^*}$. Hence, $N = \prod_{j=1,\ldots,n} p_j$. □

10.4.4 Converting DTAs into Stepwise Tree Automata

In this section, we discuss how NTAs (and DTAs) can be converted to STAs. We also present some comparative results regarding their minimal size.

First, we convert NTAs into NPTAs. Given an NTA $A = (Q, \Sigma, \delta, F)$, we define a parallel tree automaton PTA(A) $:= (Q, \Sigma, (A^a)_{a\in\Sigma}, F, o)$ with the same states and final states such that:

- A^a is the union of all NFAs B for which $\delta(q, a) = L(B)$ for some $q \in Q$; and,

- $o(F_B) = q$, for every $\delta(q,a) = L(B)$ with $B = (Q_B, Q, \delta_B, I_B, F_B)$.

This transformation preserves unambiguity but not determinism, that is, DTA(DFA)s are mapped to UTA(UFA)s. The reason why determinism fails is that the union of DFAs with disjoint languages is unambiguous, but not necessarily deterministic (it may have multiple initial states).

As we showed in Proposition 10.22, every PTA can be rewritten as an STA with fewer or equally many states. In fact, this rewriting can be performed by a natural transformation which preserves determinism. As we explain next, the idea is to unify all NFAs of a PTA into a single NFA.

Let $A = (Q_A, \Sigma, (A^a)_{a\in\Sigma}, F_A, o)$ be an NPTA and let, for every $a \in \Sigma$, $A^a = (Q_a, Q_A, \delta_a, I_a, F_a)$. We define an STA $B = (Q_B, \Sigma, \delta_B, (I_B^a)_{a\in\Sigma}, F_B)$ that recognizes the same language. Its set of states Q_B is $\uplus_{a\in\Sigma} Q_a$ and the set of final states F_B is defined as $o^{-1}(F_A) = \{q \mid o(q) \in F_A\}$. The transitions of B are then given by the following two inference rules:

$$\frac{q_1 \xrightarrow{p} q_2 \text{ is a transition in } A^a \qquad q \in o^{-1}(p)}{q_1 \xrightarrow{q} q_2 \text{ is a transition in } B} \qquad \frac{q \in I^a \qquad a \in \Sigma}{q \in I_B^a}$$

The STA in Figure 10.7, for instance, is the translation of the PTA in Figure 10.8. The main difference is that the STA shares the states of all NFAs of the PTA. It is this kind of sharing that allows an STA to be more succinct in some cases.

In general, the above translation preserves runs, successful runs, unambiguity, tree languages, determinism, and the number of states. By composing the two above automata conversions, we obtain the following.

Proposition 10.30. *Every DTA(DFA) or UTA(UFA) can be translated in* PTIME *to an equivalent unambiguous PTA or unambiguous STA with equally many states.*

Of course, all these classes of automata allow for determinization, but possibly with exponential blow up. Hence, minimal deterministic STAs are at most exponentially larger than minimal DTA(DFA)s.

Conversely, it can be shown analogously as in the proof of Proposition 10.29 that minimal deterministic STAs can also be at least exponentially larger than minimal DTA(DFA)s. However, a bit of care needs to be taken for the disjointness condition of the regular languages in DTAs.

Essentially, it has to be argued that, in general, a disjoint union of DFAs can be exponentially smaller than the minimal DFA accepting the union of the languages. This can be done as follows. Let, for every $j \in \mathbb{N}$, p_j denote the j-th prime number. We now define, for every $n \in \mathbb{N}$, a language L_n for which the minimal DFA has size at least $\prod_{j=1}^{n} p_j$ and which can represented with a disjoint union DFAs of which the sum of the sizes is at most $\sum_{j=1}^{n} p_n^3$.

Thereto, let $n \in \mathbb{N}_0$. Then, define, for each $j = 1, \ldots, n$, A_j^n to be the minimal DFA accepting the language defined by the regular expression $b^j(b^{p_n}b^{p_j})^*$. Obviously, the intersection of the languages defined by the A_j^n is empty, as each A_j^n only accepts strings of which the length is j modulo p_n. As the size of each A_j^n is smaller than p_n^3, the sum of the sizes of the A_j^n is bounded from above by $\sum_{j=1}^{n} p_n^3$.

Finally, we have to argue that the minimal DFA for L_n has size exponential in n. This follows from Theorem 4 in [PS02], where it is shown that the minimal DFA for the union of $L(A_1^n) \cup \cdots \cup L(A_n^n)$ is at least $\mathrm{LCM}(p_1, \ldots, p_n)$, which is $\prod_{j=1}^{n} p_j$. Here, LCM denotes the least common multiple. Hence, we have obtained the following.

Proposition 10.31. *There exists an infinite class of languages $(L_n)_{n \in \mathbb{N}}$ such that for every L_n, the minimal deterministic STA for L_n is exponentially larger than the minimal DTA(DFA) for L_n.*

10.5 Restrained Competition Schemas

We now focus on a class of top-down deterministic models. According to the definition of Brüggemann-Klein, Murata and Wood, a NTA A is top-down deterministic if, for every transition $\delta(q, a) = L$, where L is a regular language over states, the language L contains at most one string of length n for every $n \in \mathbb{N}$ [BKMW01].

In this section, however, we treat a different notion, namely the class of restrained competition EDTDs, defined in Chapter 8. Under the assumption that EDTDs are represented as EDTD(DFA)s, we show that the latter notion still allows for

(i) a PTIME minimization algorithm; and

(ii) uniqueness up to isomorphism of the minimal model.

Analogously as we did for unranked tree automata previously in the chapter, we will use DFAs to represent internal regular languages in (extended) DTDs. Recall from the beginning of the chapter, that the *size* $|D|$ of an extended DTD $D = (\Sigma, \Delta, d, s_d, \mu)$ is $|\Delta| + \sum_{a^i \in \Delta} |D_{a^i}|$, where $d(a^i) = D_{a^i}$.

Given an EDTD D, we say that *minimizing D* is the act of finding an EDTD D' such that $L(D) = L(D')$ and D' is the smallest EDTD with this property. The goal of this section is to prove the following theorem:

Theorem 10.32. *Every restrained competition EDTD(DFA) can be minimized in polynomial time. This minimal restrained competition deterministic EDTD(DFA) is unique up to isomorphism.*

We first give the minimization algorithm and we then prove Theorem 10.32 in a series of lemmas.

Let $D = (\Sigma, \Delta, d, s_d, \mu)$ be a restrained competition EDTD. The following algorithm minimizes D:

(1) *Reduce D*, that is,

   (a) remove all symbols a^i from Δ for which $L((D, a^i)) = \emptyset$, and the corresponding rules $a^i \to D_{a^i}$ from d; and,

   (b) remove all symbols a^i from Δ which are not reachable in d, and the corresponding rules $a^i \to D_{a^i}$.

(2) Test, for each a^i and a^j in Δ, $i < j$, whether $L((D, a^i)) = L((D, a^j))$. If $L((D, a^i)) = L((D, a^j))$, then

(a) replace all occurrences of a^j in the definition of d by a^i. That is, let, for every $b^k \in \Delta$, D_{b^k} be the DFA such that d contains the rule $b^k \to D_{b^k}$. Then, replace every transition $\delta(q_1, a^j) = \{q_2\}$ of D_{b^k} by $\delta(q_1, a^i) = \{q_2\}$.

(b) remove a^j from Δ; and,

(c) remove the rule $a^j \to D_{a^j}$ from d.

(3) For each rule $a^i \to D_{a^i}$ in d, minimize D_{a^i}.

We argue that the algorithm can be executed in polynomial time. Step (1) can be performed in polynomial time by a polynomial number of reachability and emptiness tests of DTDs. Testing whether a symbol is reachable is in NLOGSPACE, by a straightforward reduction to graph reachability. Testing whether a DTD(DFA) defines an empty language is in PTIME due to Proposition 3.16. The equivalence test in step (2) is in PTIME by Theorem 9.4. We note that the theorem is formulated in terms of regular expressions, but that the proposed algorithm also works for EDTD(DFA)s. Indeed, in that case, the Glushkov automata do not have to be considered, as the EDTD(DFA) already contains DFAs for the internal languages. A crucial observation, however, is that the removal of the types in all transitions of a DFA for a restrained competition language preserves determinism. Step (3) can be carried out in polynomial time since minimizing DFAs is in polynomial time [HMU01].

Let $D_{\min}$ be the EDTD obtained by applying the above algorithm on a restrained competition EDTD D. It remains to show that $D_{\min}$ is the minimal restrained competition EDTD for $L(D)$. More formally, we have to show that

(a) $D_{\min}$ is restrained competition;

(b) $L(D_{\min}) = L(D)$; and that

(c) every minimal restrained competition EDTD D_0 for $L(D)$ is isomorphic to $D_{\min}$.

Obviously (a) and (b) hold. We proceed with showing (c).

Lemma 10.33. *Let D_1 and D_2 be reduced, restrained competition EDTD(DFA)s such that $L(D_1) = L(D_2)$ and let $t \in L(D_1) = L(D_2)$. Let t'_1 and t'_2 be the unique witnesses of t for D_1 and D_2, respectively, and let u be a node in t. Then $L((D_1, lab^{t'_1}(u))) = L((D_2, lab^{t'_2}(u)))$.*

Proof. Let a^i and a^j be the label of u in t'_1 and t'_2, respectively.

If $|L((D_1, a^i))| = |L((D_2, a^j))| = 1$, the proof is trivial. We show that $L((D_1, a^i)) \subseteq L((D_2, a^j))$. The other inclusion follows by symmetry.

Towards a contradiction, assume that there exists a tree $t_0 \in L((D_1, a^i)) - L((D_2, a^j))$. As D_1 is reduced, there exists a tree T_0 in $L(D_1)$, such that *(i)* t_0 is a subtree of T_0 at some node v; and, *(ii)* $\mathrm{lab}^{T'_0}(v) = a^i$, where T'_0 is the unique witness of T_0 in D_1. As $\mathrm{lab}^{t'_1}(u) = a^i = \mathrm{lab}^{T'_0}(v)$, the tree $t_3 = t[u \leftarrow t_0]$ is also in $L(D_1)$. As D_1 and D_2 are equivalent, t_3 is also in $L(D_2)$. Notice that u has the same ancestor-sibling-string in t and in $t[u \leftarrow t_0]$. By Theorem 8.19, D_2 has ancestor-sibling-based types, which implies that $\mathrm{lab}^{t'_3}(u) = a^j$ for the unique witness t'_3 of t_3 in D_2. Therefore, $t_0 \in L((D_2, a^j))$, which leads to the desired contradiction. □

Let $D_{\min} = (\Sigma, \Delta_{\min}, d_{\min}, s_{d_{\min}}, \mu_{\min})$ be an EDTD which is obtained by applying the above minimization algorithm to an EDTD D over Σ. The next lemma states that every minimal restrained competition EDTD has an equal number of types as $D_{\min}$.

Lemma 10.34. *Let $D_0 = (\Sigma, \Delta_0, d_0, s_{d_0}, \mu_0)$ be a minimal restrained competition EDTD for $L(D_{min})$. Then, for every $a \in \Sigma$, we have that*

$$|\{a^i \in \Delta_0 \mid \mu_0(a^i) = a\}| = |\{a^j \in \Delta_{min} \mid \mu_{min}(a^j) = a\}|.$$

Proof. Fix an $a \in \Sigma$ and denote the sets $\{a^i \in \Delta_0 \mid \mu_0(a^i) = a\}$ and $\{a^j \in \Delta_{\min} \mid \mu_{\min}(a^j) = a\}$ by $\text{Types}_0(a)$ and $\text{Types}_{\min}(a)$ respectively. We first show that $|\text{Types}_0(a)|$ cannot be larger than $|\text{Types}_{\min}(a)|$. Towards a contradiction, assume that $|\text{Types}_0(a)| > |\text{Types}_{\min}(a)|$. For every $a^i \in \text{Types}_0(a)$, let t_i be an arbitrary tree so that a^i is a label in the unique witness t'_{i,D_0} of t_i in D_0. Also, let $t'_{i,D_{\min}}$ be the unique witness of t_i in $D_{\min}$.

According to the Pigeonhole Principle, there exist two trees t'_{j,D_0} and t'_{k,D_0} for which the a^j-labeled node u in t'_{j,D_0} and the a^k-labeled node v in t'_{k,D_0} is labeled by the same a^ℓ in both $t'_{j,D_{\min}}$ and $t'_{k,D_{\min}}$.

From Lemma 10.33, it now follows that $L((D_0, a^j)) = L((D_{\min}, a^\ell)) = L((D_0, a^k))$. Therefore, replacing every a^k with a^j in D_0 results in an equivalent, strictly smaller restrained competition EDTD than D_0. This contradicts that D_0 is minimal.

The other direction can be proved completely analogously, with the roles of D_0 and $D_{\min}$ interchanged. Now the contradiction is that $D_{\min}$ cannot be the output of the minimization algorithm, as there still exist a^j and a^k in $\text{Types}_{\min}$ for which $L((D_{\min}, a^j)) = L((D_{\min}, a^k))$. □

We now know that every minimal restrained competition EDTD for $L(D_{\min})$ has the same number of types for each alphabet symbol. We argue that, for every minimal restrained competition EDTD $D_0 = (\Sigma, \Delta_0, d_0, s_{d_0}, \mu_0)$ accepting $L(D_{\min})$, there exists an bijection I between Δ_0 and $\Delta_{\min}$ such that $I(a^i)$ is the unique $a^j \in \Delta_{\min}$ for which $L((D_0, a^i)) = L((D_{\min}, a^j))$. But this immediately follows from Lemma 10.33. Let b^k be an arbitrary symbol in $\Delta_{\min}$. Let L_{b^k} and $L_{I(b^k)}$ denote the languages $L(A_{b^k})$ and $L(A_{I(b^k)})$ for the DFAs $A_{b^k} = d_{\min}(b^k)$ and $A_{I(b^k)} = d_0(I(b^k))$, respectively. Then, we have that $L_{b^k} = \mathbf{I}^{-1}(L_{I(b^k)})$ (where we denoted by $\mathbf{I}$ the homomorphic extension of I to string languages). As minimal DFAs for a given regular language are unique up to isomorphisms, we have the following lemma:

Lemma 10.35. *Every minimal restrained competition EDTD D_0 for $L(D_{min})$ is isomorphic to D_{min}.*

Theorem 10.32 now follows from Lemma 10.35.

The next corollary is immediate from the observation that, given a single-type EDTD, the minimization algorithm also returns a single-type EDTD.

Corollary 10.36. *Every single-type EDTD(DFA) can be minimized in* PTIME. *This minimal single-type EDTD is unique up to isomorphism.*

11

Discussion

The focus of the second part of this dissertation was mainly on the expressive power and complexity of XML schema languages. Three classes of XML schema languages have been quite extensively studied in Chapters 8, 9, and 10: schemas with the Element Declarations Consistent (EDC) restriction, schemas which are 1-pass preorder typeable (1PPT), and schemas which are top-down typeable (TDT).

We have argued that EDC does not capture the complete class of all efficiently typeable schemas, not in a streaming context and not in a top-down processing context. Indeed, it turns out that the classes of schemas with 1PPT and TDT define more liberal notions of typeability and are quite robust since they can be characterized in several natural ways. Interestingly, the latter semantically defined classes can be captured by EDTDs with restricted regular expressions: restrained competition and unambiguously typed regular expressions. So, the global constraints of 1PPT and TDT are characterized by local constraints on regular expressions. Although restrained competition or unambiguously typed regular expressions are not syntactical, just like the one-unambiguous regular expressions characterizing Unique Particle Attribution (UPA), a polynomial time algorithm exists to recognize them. In terms of expressive power, the classes of schemas with EDC, 1PPT, and TDT form a very natural (strict) hierarchy. They allow types of elements to depend on the labels of (1) their ancestors, (2) their ancestors and their left siblings, and (3) their ancestors and their left and right siblings, respectively. The latter dependencies lead the way to *syntactical* characterizations for schemas with EDC, 1PPT, and TDT. The latter characterizations are given in terms of the ancestor-based, ancestor-sibling-based, and ancestor-all-sibling-based (or spine-based) schemas of Chapter 8.

Such syntactical characterizations are of particular interest. Indeed, in [BMNS05], a practical study investigated to which extent the features not present in DTDs are actually used in XML Schema Definitions (XSDs) occurring in practice. To this end, a corpus of XSDs was harvested from the web, including many high-quality schemas representing various XML standards. Concerning expressive power we were surprised that only 15% of the XSDs in our corpus use typing in a way that goes beyond the power of DTDs. Moreover, of this 15% the vast majority of the schemas use typing in its most simplistic form: types only depend on the parent context. Although it might indeed be the case that advanced expressiveness is not required in practice, another plausible explanation is that the actual modeling power of XSDs remains unclear to most users: the XML Schema specification is very hard to read and the effect of constraints on typing and validation is not fully understood. Thus, the average XML practitioner would benefit from a clear description of what kind of context dependencies can actually be expressed within XML Schema. We believe that the syntactical characterization of the expressive power of XML Schema are quite helpful in this respect.

In Chapter 8, we argued that both EDC and UPA already imply 1PPT (and therefore efficient typing). Thus, with respect to efficient typing, when adhering to UPA, it does not make much sense to also enforce EDC and vice-versa. It should be noted that the class of EDTDs satisfying both EDC and UPA (like XML Schema) are a strict subclass of the EDTDs satisfying only one of EDC and UPA.

In formal language terms, schemas with EDC, 1PPT, and TDT are characterized as single-type, restrained competition, and top-down deterministic EDTDs, respectively. We mentioned in Chapter 8 that Murata, Lee, and Mani already showed that, in terms of expressive power, DTD $\subsetneq$ EDTD$^{\mathrm{st}}$ $\subsetneq$ EDTD$^{\mathrm{rc}}$ $\subsetneq$ EDTD [MLMK05]. They exhibited concrete tree languages that are in one class but not in the other. We extended this hierarchy with the class EDTD$^{\mathrm{td}}$ of top-down deterministic EDTDs, which are more expressive than EDTD$^{\mathrm{rc}}$s but still less expressive than EDTDs, which define the full class of (homogeneous) unranked regular tree languages. Our semantical characterizations provide tools to show inexpressibility for arbitrary tree languages. For instance, using the closure of top-down deterministic EDTDs under spine-guarded subtree exchange, it is immediate that EDTD$^{\mathrm{td}}$ cannot define the set of all Boolean tree-shaped circuits evaluating to true.

Our further investigation leads us to the complexity of several decision problems for the latter restricted EDTDs, such as INCLUSION, EQUIVALENCE, and INTERSECTION NON-EMPTINESS. Naturally, the complexity lower bounds for these problems for the regular expressions used in the EDTDs also carry over to the problems for the EDTDs themselves. However, in Chapter 9, we showed that, for DTDs and single-type EDTDs the complexity *upper bounds* for INCLUSION and EQUIVALENCE for the classes of regular expressions also carry over to the corresponding problems for the EDTDs themselves (Theorem 9.4). A similar result holds for restrained competition and top-down deterministic EDTDs, but it is more restricted, that is, the upper bounds for INCLUSION and EQUIVALENCE carry over from the class of regular expressions *without the types* used in the EDTDs. However, these results do not hold for the INTERSECTION NON-EMPTINESS problem. Even though the complexity upper bounds carry over to DTDs, this is very unlikely to be the case for single-type, restrained competition, or top-down deterministic EDTDs, unless PSPACE = EXPTIME.

We also investigated the MINIMIZATION problem for these schemas. In Chapter 10 we obtained that, when content models are represented by deterministic finite automata, it is possible to minimize restrained competition EDTDs in polynomial time.

In this class, minimal EDTDs are even unique up to isomorphism. Moreover, as the minimization algorithm preserves the single-type property of its input EDTD, we also obtain that the above results hold for single-type extended DTDs. When content models are represented by regular expressions, or nondeterministic automata, the minimization problem for schemas turns PSPACE-hard, since it is at least as hard as the minimization problem for the representation of the content models.

Based on the above results, we make the following recommendation. It is clear that the EDC and UPA constraints imply the existence of efficient typing algorithms. However, we have shown that their expressive power can at least be extended to the level of restrained competition EDTDs without having a significant negative effect on the complexity of validation, or on decision problems such as RECOGNITION, SIMPLIFICATION, INCLUSION, EQUIVALENCE, INTERSECTION NON-EMPTINESS, or MINIMIZATION. So, for these reasons we propose to replace the EDC and the UPA constraints by restrained competition EDTDs.

Although we think the restriction to unambiguous typing increases transparency and efficiency of validation, the recommendations in the present dissertation do not justify the former. For instance, Relax NG as well as the formal model for XML Schema of Siméon and Wadler [SW03] allow ambiguous typing to relieve users from opaque restrictions and reaches the robust class of unranked regular tree languages which are closed under all Boolean operations. Especially in the context of data exchange it is of extreme importance that a schema language is closed under union (which is not the case for XML Schema). However, if unambiguous typing and efficient processing is required, it should not be enforced by ad-hoc restrictions, but by the most liberal ones. We believe the restrictions to 1-pass preorder typeable schemas or top-down typeable schemas are adequate. Moreover, they can be reached by allowing restricted regular expressions or by making use of the equivalent syntactic framework of ancestor-sibling-based or spine-based schemas.

In Chapter 10 we also investigated a larger class of tree languages: we studied MINIMIZATION for automata for unranked regular tree languages. Here, we first studied minimization for the bottom-up deterministic unranked tree automata which are standard in database theory (see Section 2.2), assuming that the languages in their transition function are represented by DFAs. We denote the latter class by DTA(DFA). We showed that the minimization problem for DTA(DFA)s is NP-complete. The latter result is, when extrapolating from known results about string and ranked tree automata, quite unexpected from a "deterministic" representation of regular languages. The source of this complexity is a minor amount of non-determinism that is still present in the manner how DTA(DFA)s are represented. Indeed, DTA(DFA)s still allow to represent regular languages over states by a *disjoint union of DFAs* , as we examplify in Chapter 10. Although the latter representation is *unambiguous*, it is not deterministic, in the sense that there is one nondeterministic step in the choice of an initial state. It can also be observed that the canonical translations of DTA(DFA)s over the well-known ranked encodings result in unambiguous rather than deterministic binary tree automata [GKPS05].

As the NP-hardness result for minimization is somewhat unsatisfying, we investigated several more recent notions of determinism for unranked tree automata. We compared three different notions: deterministic parallel unranked tree automata, which are defined independently in [CLT05] and [RB04], deterministic stepwise tree automata [CNT04], and deterministic ranked tree automata over the *first-child next-sibling* encoding. In general, the stepwise tree automata provide the smallest minimal automata: they are generally quadratically smaller than parallel unranked tree automata and exponentially smaller than ranked tree automata over the first-child next-sibling encoding. Moreover, since they have a direct connection to ranked tree automata through an encoding based on currying, a PTIME minimization algorithm and a Myhill-Nerode theorem is immediate.

In spite of the quadratical difference in minimal size, deterministic stepwise automata and deterministic parallel unranked tree automata are very closely related. Essentially, the differences between parallel unranked tree automata and stepwise automata are that

1. parallel unranked tree automata use an *output function* to relate states of the internal DFAs to the states of the tree automaton; and that

2. parallel unranked tree automata require the state sets of the internal DFAs to be *disjoint*.

While the first difference does not have much effect on the size of minimal deterministic parallel unranked tree automata, it is the second difference that causes them to be quadratically larger than stepwise automata.

Future Work

The results in Part II lend themselves to the design and development of a theoretically well-funded software tool for manupulating XML schemas. Indeed, by implementing the algorithms presented in Part II a tool can be constructed that

(1) minimizes a given schema;

(2) decides whether one schema is included in another;

(3) translates a schema into a schema of a less expressive schema language (such that the constructed schema defines the smallest language containing the given language);

(4) constructs the union, intersection, difference of schemas; and

(5) decides whether a set of schemas defines a common document.

In the above, the word “schema” should be interpreted as the restricted EDTDs we studied in Part II. In an initial phase, a prototype can be built that works on EDTDs, which can be extended to work with DTDs, XML Schema Definitions and Relax NG schemas in a later phase. We believe that such a tool is useful in the process of schema development and in areas such as data integration and translation.

Bibliography

[ABJ98] P. A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV 1998)*, pages 305–318, 1998.

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Symposium on the Theory of Computing (STOC 2004)*, pages 202–211, New York, 2004. ACM Press.

[AMN$^+$03a] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003.

[AMN$^+$03b] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. *Journal of Computer and System Sciences*, 66(4):688–727, 2003.

[Bal02] S. Bala. Intersection of regular languages and star hierarchy. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP 2002)*, pages 159–169, 2002.

[BFC03] V. Benzaken, A. Frisch, and G. Castagna. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, pages 51–63. ACM Press, 2003.

[BFG05] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *Proceedings of the 24th Symposium on Principles of Database Systems (PODS 2005)*, pages 25–36. ACM Press, 2005.

[BFS00] P. Buneman, M. Fernandez, and D. Suciu. UnQl: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[BKMW01] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[BKW98] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[BKW00] A. Brüggemann-Klein and D. Wood. Caterpillars: A context specification technique. *Markup Languages*, 2(1):81–106, 2000.

[BMN02] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[BMNS05] G. J. Bex, W. Martens, F. Neven, and T. Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *Proceedings of the 14th International Conference on World Wide Web (WWW 2005)*, pages 712–721, New York, 2005. ACM Press.

[BNST05] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. Manuscript, 2005.

[BNV04] G. J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML schema: A practical study. In *Proceedings of the 7th International Workshop on the Web and Databases (WebDB 2004)*, pages 79–84, 2004.

[BPSM$^+$04] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML). Technical report, World Wide Web Consortium, February 2004. http://www.w3.org/TR/REC-xml/.

[BPV04] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems*, 29(4):710–751, 2004.

[Buc43] R. Buck. Partition of space. *American Mathematical Monthly*, 50(9):541–544, 1943.

[CD99] J. Clark and S. DeRose. XML Path Language (XPath). Technical report, World Wide Web Consortium, November 1999. http://www.w3.org/TR/xpath.

[CDG$^+$01] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2001. http://www.grappa.univ-lille3.fr/tata.

[CGGLV03] D. Calvanese, De G. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

[CGKV88] S. S. Cosmadakis, H. Gaifman, P. C. Kanellakis, and M. Vardi. Decidable optimization problems for database logic programs. In *Proceedings of the 20th annual ACM Symposium on Theory of Computing (STOC 1988)*, pages 477–490. ACM Press, 1988.

[Chl86] B. S. Chlebus. Domino-tiling games. *Journal of Computer and System Sciences*, 32(3):374–392, 1986.

[Cho02] B. Choi. What are real DTDs like? In *Proceedings of the 7th International Workshop on the Web and Databases (WebDB 2002)*, pages 43–48, 2002.

[Cla99] J. Clark. XSL transformations version 1.0. Technical report, World Wide Web Consortium, August 1999. http://www.w3.org/TR/WD-xslt.

[Cla02] J. Clark. Multi-format schema converter based on RELAX NG. http://www.thaiopensource.com/relaxng/trang.html, 2002.

[CLT05] J. Cristau, C. Löding, and W. Thomas. Deterministic automata on unranked trees. In *Proceedings of the 15th International Symposium on Fundamentals of Computation Theory (FCT 2005)*, pages 68–79, 2005.

[CM01] J. Clark and M. Murata. Relax NG specification. http://www.relaxng.org/spec-20011203.html, December 2001.

[CMV04] C. Sacerdoti Coen, P. Marinelli, and F. Vitali. Schemapath, a minimal extension to XML Schema for conditional constraints. In *Proceedings of the 14th International Conference on World Wide Web (WWW 2004)*, pages 164–174, New York, 2004. ACM Press.

[CNT04] J. Carme, J. Niehren, and M. Tommasi. Querying unranked trees with stepwise tree automata. In *International Conference on Rewriting Techniques and Applications (RTA 2004)*, pages 105–118, 2004.

[Coo74] S. A. Cook. An observation on time-storage trade-off. *Journal of Computer and System Sciences*, 9(3):308–316, 1974.

[Cou89] B. Courcelle. On recognizable sets and tree automata. In *Resolution of equations in algebraic structures*, pages 93–126, 1989.

[DuC02] B. DuCharme. Filling in the DTD gaps with Schematron. O'Reilly, May 2002. http://www.xml.com/pub/a/2002/05/15/schematron.html.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[EM03] J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39:613–698, 2003.

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 1985.

[FGK03] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 188–197, 2003.

[FR75] J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM Journal on Computing*, 4(1):69–76, 1975.

[GGM$^+$04] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, 2004.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[GKPS05] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of XPath query evaluation and XML typing. *Journal of the ACM*, 52(2):284–335, 2005.

[Glu61] V. Glushkov. Abstract theory of automata. *Usp. Mat. Nauk*, 16(1):3–41, 1961. English translation in Russian Math. Surveys, vol. 16, pp. 1–53, 1961.

[GS97] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer, 1997.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[HO02] L. Hemaspaandra and M. Ogihara. *The Complexity Theory Companion*. Springer, 2002.

[HP03] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[HRS76] Harry B. Hunt III, Daniel J. Rosenkrantz, and Thomas G. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *Journal of Computer and System Sciences*, 12(2):222–268, April 1976.

[Jel01] Rick Jelliffe. The current state of the art of schema languages for XML. Presentation at XML Asia Pacific, Sidney, Australia, 2001.

[Jel05] R. Jelliffe. Schematron, February 2005. http://xml.ascc.net/schematron/.

[Joh90] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, A. R. Meyer, N. Nivat, M. S. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume A, chapter 2, pages 67–161. North-Holland, 1990.

[JR93] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.

[KlS00] N. Klarlund, A. Møller, and M. I. Schwartzbach. The DSD schema language. In *Proceedings of the 3th ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP 2000)*, 2000.

[Koz77] D. Kozen. Lower bounds for natural proof systems. In *Proceedings 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 254–266. IEEE, 1977.

[Koz92] D. Kozen. On the Myhill-Nerode theorem for trees. *Bulletin of the European Association for Theoretical Computer Science*, 147:170–173, 1992.

[KS03] C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. In *Proceedings of the 9th International Workshop on Database Programming Languages (DBPL 2003)*, pages 233–256, Berlin, 2003. Springer.

[LC00] D. Lee and W. W. Chu. Comparative analysis of six XML schema languages. *ACM SIGMOD Record*, 29(3):76–87, 2000.

[Len83] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.

[LLS84] R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer. Alternating pushdown and stack automata. *SIAM Journal on Computing*, 13(1):135–155, 1984.

[LMM00] D. Lee, M. Mani, and M. Murata. Reasoning about XML schema languages using formal language theory. Technical report, IBM Almaden Research Center, 2000. Log# 95071.

[Mal04] A. Malcher. Minimizing finite automata is computationally hard. *Theoretical Computer Science*, 327(3):375–390, 2004.

[Man01] M. Mani. Keeping chess alive - Do we need 1-unambiguous content models? In *Extreme Markup Languages*, Montreal, Canada, 2001.

[Mar04] M. Marx. XPath with conditional axis relations. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT 2004)*, pages 477–494, 2004.

[MLMK05] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):1–45, 2005.

[MN00] S. Maneth and F. Neven. Structured document transformations based on XSL. In *Research Issues in Structured and Semistructured Database Programming (DBPL 1999)*, pages 80–98. Springer, 2000.

[MN04] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proceedings of the 23d Symposium on Principles of Database Systems (PODS 2004)*, pages 23–34. ACM Press, 2004.

[MN05a] W. Martens and F. Neven. On the complexity of typechecking top-down XML transformations. *Theoretical Computer Science*, 336(1):153–180, 2005. Extended abstract appeared in the 9th International Conference on Database Theory (ICDT 2003).

[MN05b] W. Martens and J. Niehren. Minimizing tree automata for unranked trees. In *Prodeedings of the 10th International Symposium on Database Programming Languages (DBPL 2005)*, pages 232–246, 2005.

[MN06] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. *Journal of Computer and System Sciences*, 2006. To Appear. Preliminary version available at http://alpha.uhasselt.be/wim.martens. Extended abstract appeared in the 23d Symposium on Principles of Database Systems (PODS 2004).

[MNG05] W. Martens, F. Neven, and M. Gyssens. On typechecking top-down XML transformations: Fixed input or output schemas. Submitted, available at http://alpha.uhasselt.be/wim.martens. Part of this article appeared as Section 3 in [MN04], 2005.

[MNS04] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Proceedings of the 29th International Symposium on Mathematical Foundations of Computer Science (MFCS 2004)*, pages 889–900, Berlin, 2004. Springer.

[MNS05] W. Martens, F. Neven, and T. Schwentick. Which XML schemas admit 1-pass preorder typing? In *Proceedings of the 10th International Conference on Database Theory (ICDT 2005)*, pages 68–82, Berlin, 2005. Springer.

[MPBS05] S. Maneth, T. Perst, A. Berlea, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of the 24th Symposium on Principles of Database Systems (PODS 2005)*, pages 283–294. ACM Press, 2005.

[MS99a] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDT 1999)*, pages 277–295. Springer, 1999.

[MS99b] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of the Eighteenth Symposium on Principles of Database Systems (PODS 1999)*, pages 215–226. ACM Press, 1999.

[MS04] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[MSV03] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66–97, 2003.

[Nev02] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[New80] D. J. Newman. Simple analytic proof of the prime number theorem. *American Mathematical Monthly*, 87:693–696, 1980.

[NS] F. Neven and T. Schwentick. XML schemas without order. Unpublished manuscript, 1999.

[NS00] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the 19th Symposium on Principles of Database Systems (PODS 2000)*, pages 145–156, 2000.

[NS02] F. Neven and T. Schwentick. Query automata on finite trees. *Theoretical Computer Science*, 275:633–674, 2002.

[NS03] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, pages 315–329, 2003.

[PS02] G. Pighizzini and J. Shallit. Unary language operations, state complexity and Jacobsthal's function. *International Journal of Foundations of Computer Science*, 13(1):145–159, 2002.

[PV00] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of the 19th Symposium on Principles of Database Systems (PODS 2000)*, pages 35–46, New York, 2000. ACM Press.

[RB04] S. Raeymaekers and M. Bruynooghe. Minimization of finite unranked tree automata. Manuscript, 2004.

[Sch04] T. Schwentick. XPath query containment. *SIGMOD Record*, 33(1):101–109, 2004.

[Sed83] R. Sedgewick. *Algorithms.* Addison-Wesley, 1983.

[Sei90] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.

[Sei94] H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

[SH85] R. E. Stearns and H. B. Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985.

[Sip97] M. Sipser. *Introduction to the Theory of Computation.* Brooks/Cole Publishing, 1997.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *Conference Record of Fifth Annual ACM Symposium on Theory of Computing (STOC 1973)*, pages 1–9, New York, 1973. ACM Press.

[SM03] C. M. Sperberg-McQueen. XML Schema 1.0: A language for document grammars. In *XML 2003 - Conference Proceedings*, 2003.

[SMT05] C. M. Sperberg-McQueen and H. Thompson. XML Schema. http://www.w3.org/XML/Schema, 2005.

[Suc01] D. Suciu. Typechecking for semistructured data. In *Proceedings of the 8th Workshop on Data Bases and Programming Languages (DBPL 2001)*, pages 1–20, Berlin, 2001. Springer.

[Suc02] D. Suciu. The XML typechecking problem. *SIGMOD Record*, 31(1):89–96, 2002.

[SV02] L. Segoufin and V. Vianu. Validating streaming XML documents. In *Proceedings of the 21st Symposium on Principles of Database Systems (PODS 2002)*, pages 53–64, New York, 2002. ACM Press.

[SW03] J. Siméon and P. Wadler. The essence of XML. In *30th Symposium on Principles of Programming Languages (POPL)*, pages 1–13, New York, 2003. ACM Press.

[TBMM04] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. Technical report, World Wide Web Consortium, October 2004. http://www.w3.org/TR/xmlschema-1/.

[Toz01] A. Tozawa. Towards static type checking for XSLT. In *Proceedings of the ACM Symposium on Document Engineering (DOCENG 2001)*, pages 18–27, 2001.

[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.

[Var89] M. Y. Vardi. A note on the reduction of two-way automata to one-way automata. *Information Processing Letters*, 30:261–264, March 1989.

[vdV02] E. van der Vlist. *XML Schema*. O'Reilly, 2002.

[vdV03] E. van der Vlist. *Relax NG*. O'Reilly, 2003.

[Woo] P. T. Wood. Containment for XPath fragments under DTD constraints. Full version of [Woo03].

[Woo01] P. T. Wood. Minimising simple XPath expressions. In *Proceedings of the 7th International Workshop on the Web and Databases (WebDB 2001)*, pages 13–18, 2001.

[Woo03] P. T. Wood. Containment for XPath fragments under DTD constraints. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, pages 300–314, 2003.

Samenvatting

XML (eXtensible Markup Language) is tijdens de laatste jaren geëvolueerd tot het standaard-dataformaat voor de uitwisseling van gegevens op het internet [ABS99]. De belangrijkste voordelen van XML zijn dat het een standaard- en intuitive manier voorziet om een zeer groot bereik aan gegevens te modelleren, en dat het gebruikers toelaat om hun eigen tags in documenten te definiëren. Dit laatste geeft gebruikers de mogelijkheid om een eigen formaat voor XML-documenten te ontwikkelen, hetwelk wordt gedefinieerd aan de hand van een *XML-schema*. De aanwezigheid van zo'n schema verbetert de efficiëntie van de automatisering van vele taken, zoals bijvoorbeeld het verwerken van query's, query-optimalisatie en automatische data-integratie.

XML-Typechecking

Op het internet worden schema's gebruikt om de uitwisseling van gegevens te valideren. In een typisch scenario ontwikkelt een zekere gemeenschap van gebruikers een zeker schema en wordt er binnen de gemeenschap afgesproken om enkel XML-gegevens te ontwikkelen die conform zijn met dit schema. Zulke scenario's motiveren het typecheckingprobleem: in compile-time nagaan of elk XML-document dat resulteert uit de toepassing van een zekere transformatie op een geldig invoerdocument, voldoet aan een uitvoerschema [Suc01, Suc02].

Het eerste deel van dit proefschrift bestudeert het typecheckingprobleem voor *XML-naar-XML-transformaties*. Formeel gezien bestaat de invoer van het typecheckingprobleem uit een invoertype, een uitvoertype en een transformatie. Het probleem bestaat er dan in een oplossing te geven voor de vraag of, voor elk document in het invoertype, het resultaat van het toepassen van de transformatie op dit document aan het uitvoertype voldoet. Zulke types modelleren we met formele modellen voor XML-schemas: *Document Type Definitions* (DTDs) en hun robuuste extensie, de *eindige (unranked) boomautomaten* [BKMW01, LMM00, MSV03], welke ook gemodelleerd kunnen worden als *extended DTDs* [PV00, BPV04]. Deze laatste klassen dienen als een formeel model voor de taal Relax NG [CM01].

Het ligt voor de hand dat typechecking afhankelijk is van de gebruikte transformatietaal. Zoals aangetoond door Alon et al. [AMN$^+$03a, AMN$^+$03b], is het zo dat het typecheckingprobleem snel onbeslisbaar wordt wanneer de transformatietalen de mogelijkheid hebben om data-waarden met elkaar te vergelijken. Aan de andere kant, hebben Milo, Suciu, en Vianu geargumenteerd dat XML-documenten geabstraheerd kunnen worden door gelabelde, geordende bomen en dat de kracht van de meeste

XML-transformatietalen kan uitgedrukt worden door k-pebble transducers wanneer data-waarden buiten beschouwing worden gelaten [MSV03]. De auteurs hebben verder bewezen dat het typecheckingprobleem in deze context beslisbaar is. Meer bepaald, gegeven twee types τ_1 en τ_2, voorgesteld door boomautomaten, en een k-pebble transducer T, is het beslisbaar of $T(t) \in \tau_2$ voor elke $t \in \tau_1$. Hier noteren we de boom die bekomen wordt door T uit te voeren op t door $T(t)$. De complexiteit van dit probleem is desalniettemin niet-elementair en kan ook niet verbeterd worden [MSV03].

Om deze hoge complexiteit te verlagen, beschouwen we veel eenvoudigere boomtransformaties die overeenkomen met structurele recursie op bomen [BFS00] en met eenvoudige top-down XSLT-transformaties [BMN02, Cla99]. In essentie willen we transformaties modelleren die gebruikt worden om documenten te herstructureren en om er informatie uit te filteren, dus niet voor geavanceerde ondervragingen. In essentie bestaat een transformatie uit het éénmaal doorlopen van een invoerboom van de wortel tot de bladeren, waarbij elke knoop wordt vervangen door een nieuwe, mogelijk lege, boom.

We bestuderen typechecking-algoritmes die *compleet* zijn. Met andere woorden, algoritmes die voor elke mogelijke invoer het juiste antwoord geven. Deze aanpak staat in contrast met het werk dat gedaan wordt voor algemene XML-programmeertalen, zoals XDuce [HP03] en CDuce [BFC03]. Hier heeft men het doel om typechecking-algoritmes te ontwikkelen die snel en *sound* zijn. Een algoritme is sound als, indien het een positief antwoord teruggeeft, dit antwoord ook correct is. Dit betekent dat het kan voorkomen dat een invoer correct is, maar toch niet aanvaard wordt door de typechecker. Voor algemene XML-programmeertalen is compleet typechecking immers onmogelijk, aangezien de klasse van XML-transformaties onder beschouwing Turing-compleet is. In ons geval heeft het zin om complete typechecking-algoritmes te onderzoeken, daar onze klasse van transformaties enkel eerder eenvoudige transformaties bevat en zeker niet Turing-compleet is.

We bestuderen het typecheckingprobleem in Hoofdstukken 4, 5 en 6. Ons initiële doel is een niet-triviaal scenario te ontdekken waarvoor het typecheckingprobleem in PTIME zit. We parametrizeren het typecheckingprobleem door verschillende beperkingen die we de transformaties opleggen (deleting, non-deleting, begrensd of onbegrensd copying) en we beschouwen zowel boomautomaten als DTDs als invoer- en uitvoerschema's. We tonen aan dat het typecheckingprobleem EXPTIME-compleet is in de meest algemene situatie die we toelaten. Vertrekkende van dit resultaat, experimenteren we met de verschillende beperkingen die we kunnen opleggen aan het typecheckingprobleem, tot we een eerste situatie vinden waarin typechecking in polynomiale tijd is. In deze situatie laten we helemaal geen deletion toe in de transformaties en is het aantal copies dat een transformatie in één stap kan maken op voorhand vastgelegd.

Het scenario dat bestudeerd wordt voor typechecking in Hoofdstuk 4 is vrij algemeen, in de zin dat zowel het invoer- als het uitvoerschema deel uitmaken van de invoer van het typecheckingprobleem. Voor verschillende scenario's is het namelijk denkbaar is dat het invoer en/of uitvoerschema steeds hetzelfde is. Dit kan bijvoorbeeld voorkomen als gegevens telkens worden uitgewisseld van en/of naar een vaste gemeenschap. Daarom herbekijken we de scenario's die we in Hoofdstuk 4 onderzochten en bestuderen we in welke mate de complexiteit verlaagt onder de veronderstelling dat het invoer- en/of uitvoerschema vast is.

Hoewel Hoofdstukken 4 en 5 een vrij gedetailleerd overzicht geven van de complexiteit van typechecking, zijn de gevallen waarin we een polynomiale-tijd typechecking-algoritme gevonden hebben tamelijk gerestricteerd. De voornaamste reden hiervoor is dat elk van deze gevallen deletion in de transformaties volledig uitsluit. Inderdaad, vele eenvoudige filtertransformaties selecteren delen van een invoerdocument terwijl ze gewoonweg de niet-interessante gedeeltes negeren.

Daarom onderzoeken we in Hoofdstuk 6 grotere en meer flexibele klasses waarvoor het typecheckingprobleem in polynomiale tijd zit. Door onze beperkingen op de schematalen en transformaties iets te versoepelen, vinden we verschillende praktisch relevante klasses waarvoor het typecheckingprobleem in polynomiale tijd is. Bovendien laat ons werk een tamelijk volledig beeld zien, aangezien we ook aantonen dat de meeste scenario's niet kunnen uitgebreid worden zonder het typecheckingprobleem minstens NP- of coNP-hard te maken. Met andere woorden, geeft ons werk meer inzicht wanneer men snelle en complete algoritmes kan gebruiken voor het typecheckingprobleem en wanneer men zich best beperkt tot incomplete algoritmes.

Wanneer een gegeven transformatie niet typecheckt ten opzichte van een gegeven invoer- en uitvoerschema, is het belangrijk om de gebruiker een reden te kunnen teruggeven *waarom* de transformatie niet voldoet aan de eisen. Daarom onderzoeken we ook het probleem om een beschrijving van een boom t in het invoerschema te geven zodat de transformatie van t niet in het uitvoerschema zit. We bewijzen dat, voor elk van de praktisch relevante polynomiale-tijd-fragmenten die we geïdentificeerd hebben in Hoofdstuk 6, het eveneens mogelijk is om in polynomiale tijd een beschrijving van zulk een boom t te genereren.

Fundamenten van XML-Schematalen

Het eerste gedeelte van dit proefschrift richt zich vooral op de interactie tussen XML-transformaties en -schematalen. In het tweede gedeelte wijden we onszelf volledig aan de studie van XML-schematalen.

De gebruikelijke abstractie van XML Schema door unranked reguliere boomtalen is niet zeer precies. Daarom geven we in Hoofdstuk 8 meer inzicht over de eigenlijke expressieve kracht van XML Schema, door verschillende karakterisaties te geven van de *Element Declarations Consistent* (EDC) regel. In het bijzonder, laten we zien dat schemas die voldoen aan de EDC-regel in zekere zin enkel kunnen redeneren over reguliere eigenschappen van de voorouders van knopen in een boom. We argumenteren dat er meer robuuste, meer expressieve, maar even handelbare schematalen gevormd kunnen worden door EDC te vervangen door de notie van *1-pass preorder typeability (1PPT)* of *top-down typeability (TDT)*. Deze eerste is de klasse van schemas die toelaat om het type van een element van een streaming document te bepalen op het moment dat zijn opening tag gelezen wordt. De tweede is de klasse van schemas die toelaten het type van een element te bepalen zonder naar zijn afstammelingen of naar de afstammelingen van zijn broers te kijken. We tonen aan dat de klasse van 1PPT-schema's precies de klasse is gedefinieerd door de *restrained competition* grammatica's, welke geïntroduceerd werden door Murata et al. [MLMK05] en dat de TDT-schema's overeenkomen met een natuurlijke veralgemening van top-down-deterministische boomautomaten. Hiervan kunnen we concluderen dat de expressieve kracht van schema's strikt groeit, gaande van EDC naar 1PPT en van 1PPT naar TDT.

We karakterizeren de expressieve kracht van de EDC-regel, 1PPT en TDT in termen van de context van knopen, sluitings-eigenschappen, toegelaten patronen en guarded schemas. Verder tonen we aan dat het beslissen of een schema 1PPT of TDT is in PTIME zit. Beslissen of een willekeurig schema kan geschreven worden als een 1PPT-grammatica, of als één van zijn subklassen, is moeilijker: dit probleem is EXPTIME-compleet.

In Hoofdstuk 9 richten we ons op de complexiteit van verschillende elementaire beslissingsproblemen voor schemas met EDC, 1PPT en TDT. We bestuderen het inclusion-, equivalence- en intersection-non-emptiness-probleem voor schema's die voorkomen in de praktijk. Zulke schema's maken gebruik van reguliere expressies met een zeer eenvoudige structuur: ze bestaan uit een concatenatie van factoren, waar elke factor een disjunctie van strings is, mogelijk uitgebreid met "$*$", "$+$", of "?". Zulke expressies noemen we *CHAin Regular Expressions* (CHAREs). We vinden onder- en bovengrenzen voor de complexiteit van bovengenoemde beslissingsproblemen voor verschillende fragmenten van CHAREs en we beschouwen eveneens beperkingen zoals determinisme en een limiet op het aantal voorkomens van een symbool in een expressie. Voor het equivalence-probleem vinden we echter enkel een initeel PTIME-resultaat en laten we de complexiteit van de meer algemene fragmenten open. We relateren de onderzochte beslissingsproblemen met de optimalisatie van XML-schematalen door aan te tonen dat al onze onder- en bovengrenzen voor de complexiteit voor inclusion en equivalence van klasses van reguliere expressies ook van toepassing zijn op deze beslissingsproblemen voor DTDs en schema's met EDC die deze klasses van reguliere expressies gebruiken. Voor schema's met 1PPT en TDT geldt er een analoog resultaat in de zin that de bovengrenzen overgedragen worden van de overeenkomstige klasses van reguliere expressies *zonder de type-informatie* die in de schema's gebruikt wordt. Voor het intersection-non-emptiness-probleem zijn bovengenoemde resultaten echter niet waar. Hier tonen we aan dat de complexiteiten enkel overdragen naar DTDs. Voor de andere schema's dragen de complexiteiten niet over, tenzij PSPACE = EXPTIME.

Tenslotte, in Hoofdstuk 10, bestuderen we het minimalizatieprobleem voor XML-schema's. Het doel van dit hoofdstuk is verschillende formele modellen te onderzoeken die toelaten om hun aantal toestanden (of types) op een efficiënte manier te minimalizeren.

We beginnen met het bestuderen van de *unranked boomautomaten* die de basis zijn voor Relax NG [MLMK05, CM01]. We tonen aan dat, onder de veronderstelling dat de transitiefunctie voorgestesteld wordt door deterministische eindige automaten, het minimalizatieprobleem voor de bottom-up deterministische unranked boomautomaten gedefinieerd door Brüggeman-Klein, Murata, and Wood NP-compleet is [BKMW01]. Deze automaten worden als een standaardmodel gebruikt in de gegevensbankentheorie. Verder laten we zien dat minimale automaten in deze klasse niet uniek zijn op isomorfie na.

Het bovengenoemde resultaat is niet echt bevredigend. Daarom onderzoeken we meer recente klasses van boomautomaten die *wel* efficient minimalizeerbaar zijn. We doen een vergelijkende studie tussen *stepwise boomautomaten* [CNT04], *paral-*

lele boomautomaten [CLT05, RB04] en bottom-up deterministische boomautomaten over de *first-child next-sibling*-codering van unranked bomen. We tonen aan dat deterministische stepwise automaten onder deze kandidaten de kleinste representaties van reguliere unranked boomtalen vormen.

We sluiten af door het minimalizatieprobleem te onderzoeken voor de restrained competition (of 1PPT) schema's die we eerder in dit deel al onderzochten. In het bijzonder tonen we aan dat het minimalizeren van restrained competition schema's mogelijk is in PTIME en dat ze aanleiding geven tot unieke minimale schemas. Deze resultaten zijn in de veronderstelling dat de inwendige reguliere talen in de schema's niet voorgesteld worden door reguliere expressies, maar door deterministische eindige automaten. Moesten willekeurige reguliere expressies toegelaten zijn, zou het minimalizatieprobleem immers PSPACE-compleet zijn. Verder argumenteren we dat het minimalizatie-algoritme de EDC-eigenschap van zijn invoer behoudt. Als gevolg hiervan, is het ook zo dat EDC-schemas minimalizeerbaar zijn in PTIME.

List of Notations

Index