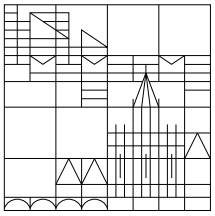
Diploma Thesis

Streaming XML Schema Validation for Relational Tree Encodings

Stefan Klinger

University of Konstanz

Department of Computer & Information Science
Konstanz, Germany



Streaming XML Schema Validation for Relational Tree Encodings
Diploma Thesis, April 2004

Author: Stefan Klinger <stefan.klinger@uni-konstanz.de>

This diploma thesis, as well as the implementation of the proposed algorithm on the enclosed CD, are available online through the Konstanzer Online-Publikations-System (KOPS), following the permanent direct URL

http://www.ub.uni-konstanz.de/kops/volltexte/2004/1234/

First assessor
Prof. Dr. Marc H. Scholl
Databases & Information Systems Group
University of Konstanz, Germany
http://www.inf.uni-konstanz.de/dbis/

Second assessor
Prof. Dr. Gottfried Barthel
Fachbereich Mathematik und Statistik
University of Konstanz, Germany
http://www.mathe.uni-konstanz.de/

Urhebervermerk Ich versichere, daß ich die vorliegende Arbeit selbständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Version of this document: Sun 2004-04-11 15:53:35 CEST

Preface

This diploma thesis introduces a new way of validating relationally encoded XML documents against XML Schema descriptions.

Validation is the process of verifying whether the given document respects a certain structure, and, given that, annotating each document node with the name of its type.

An enumeration of the nodes of the XML document tree is used as *relational tree encoding*. More precisely, during a left-to-right depth-first traversal of the tree, the nodes are annotated with the according pre order and post order indices. This pre/post enumeration was introduced by [5].

An XML Schema [15] description is considered to define a *context free grammar*. Since not all aspects of XML Schema can be expressed by a context free grammar, this thesis' focus is on the according XML Schema subset.

The proposed algorithm is based on the concept of *deriving a regular expression*, which was introduced by [1]. Hence, it is neither necessary to reconstruct the XML tree from its encoding, nor to build a finite state automaton from the XML Schema description. Moreover, the encoded tree is read as a stream, i.e., exactly once, sequentially in *document order*.

This thesis introduces *guards*, an amelioration of regular expressions which integrates information about the hierarchical structure of trees. The concept of derivation is augmented to make use of the pre/post enumeration and the enriched regular expressions. For *one-unambiguous* grammars possessing the *star normal form*, this leads to an algorithm with linear time and space requirements. All grammars induced by XML Schema descriptions are one-unambiguous. However, if star normal form cannot be guaranteed, its absence may lead to exponential time and space requirements in the worst case.

Contents

Chapter 1

Introduction

The Extensible Markup Language (XML) is a plain text format, widely used to store and exchange documents possessing a hierarchical structure. The widespread use of XML as a data storage and exchange format imposed an emerging need for storing XML documents in databases. Since there are mature relational database management systems (RDBMSs) available, an obvious approach is to transform the tree shaped XML documents into tables and store them inside the RDBMS.

However, those database management systems have been unaware of the properties the tree structure imposes on the data, disallowing potential optimizations. Also, query languages like XPath originate in the XML world and are optimized for querying tree shaped data. At first sight, they should not integrate well with relational systems.

For a while now, databases are being augmented to exploit the properties of tree shaped data for the benefit of performant query processing using those tree aware query languages. The Pathfinder working group [10] published several suggestions in this direction.

XML Schema is another W3C specification, allowing the description of structure and semantics of XML documents by means of regular expressions and context free grammars. The process of verifying whether an XML document fulfills an XML Schema description is called *validation*.

The proposed validation algorithm is based on the concept of *deriving a regular expression* [1]. It is neither necessary to reconstruct the XML tree, nor to build a finite state automaton from the XML Schema description. Instead, the algorithm directly works on the relational encoding of the XML document and the XML Schema description.

This thesis' focus is on those parts of XML Schema that can be expressed by context free grammars. Concepts like uniqueness of values, as well as keys and references, or more advanced methods for building content models, like deriving complex types by restriction, are not handled by the proposed algorithm.

Chapter 2, *Languages and Forests*, recalls and formalizes the concepts of regular languages, regular expressions and context free grammars. Trees and forests are introduced as an extension of regular languages, and the regular expressions are extended to describe them. A simple relational encoding for XML documents, the pre/post enumeration, is reviewed.

Chapter 3, *The Derivation*, introduces the concept of the derivation of a regular expression according to a relational encoding of a forest.

Chapter 4, *XML Schema Type Information*, shows how the proposed calculus integrates with XML and XML Schema, and how XML Schema type information is collected during validation.

This chapter also contains all-embracing examples, illustrating the relational tree encoding, how XML Schema is translated into context free grammars, and how type information is generated during the validation process.

Chapter 5, *Complexity*, gives a theoretical discussion of the runtime behavior. It turns out that two properties of the regular expressions, the *one-unambiguity* and the *star normal form*, influence the runtime of the algorithm. Results of real life tests are also given, highlighting the performance of the algorithm.

Chapter 6, *Implementation*, finally comments an implementation of the algorithm and associated data structures in Haskell, a purely functional programming language.

Appendix A, *XML Schema Constraints*, notes those restrictions given by the XML Schema standard that are referred to in this document.

Appendix B, *The Accompanying Software*, documents the software that is part of this thesis. It explains how to compile the source code, and which software is required. It also describes some experiments introduced theoretically in this thesis, and how you can build your own XML validator.

Chapter 2

Languages and Forests

We discuss XML tree structures and XML Schema descriptions by means of *regular languages* and *context free grammars*. We introduce these concepts from scratch to familiarize the reader with the notation used.

2.1 Regular Languages

2.1.1 Definition Let $\mathcal{A}$ be a finite set. We call it the **alphabet** . Then

$$\mathcal{A}^* := \{a_1 \dots a_n \mid n \in \mathbb{N}, 1 \leq i \leq n \Rightarrow a_i \in \mathcal{A}\}$$

is the set of finite **words** on $\mathcal{A}$, where the special symbol

$$\epsilon := () \in \mathcal{A}^*$$

denotes the **empty word** (i.e., $a_1 \dots a_n, n = 0$). Note that $\epsilon \notin \emptyset$, and throughout this thesis, always assume $0 \in \mathbb{N}$.

The elements of $\mathcal{A}$ are the atoms of our discussion. Their analogy in XML are the XML element nodes and the text nodes.

2.1.2 Definition The **concatenation** of two words is defined by

$$\begin{array}{ccccccc} \cdot : & \mathcal{A}^* & \times & \mathcal{A}^* & \longrightarrow & \mathcal{A}^* \\ & a_1 \dots a_n & , & a_{n+1} \dots a_m & \longmapsto & a_1 \dots a_m \end{array}$$

where $m, n \in \mathbb{N}, m \geq n$. We will simply write vw instead of $v \cdot w$. Additionally, for $n \in \mathbb{N}, v \in \mathcal{A}^*$ we write v^n to denote $\underbrace{v \cdot \ldots \cdot v}_{n \text{ times}}$. With this we have $v^0 = \epsilon$.

2.1.3 Definition The power set

$$\mathcal{L}_{\mathcal{A}} := \mathcal{P}(\mathcal{A}^*)$$

is called the set of **languages over the alphabet** $\mathcal{A}$.

These languages form a superset of the *regular* languages to be defined below.

2.1.4 Definition The **concatenation** of languages is defined by concatenating their words:

$$\begin{array}{rccccc} \cdot : & \mathcal{L}_\mathcal{A} & \times & \mathcal{L}_\mathcal{A} & \longrightarrow & \mathcal{L}_\mathcal{A} \\ & K & , & L & \longmapsto & \{v \cdot w \,\big|\, v \in K, w \in L\} \ . \end{array}$$

Again, we use KL to denote $K \cdot L$. The **alternation** of languages simply is their union:

$$\begin{array}{rccccc} | : & \mathcal{L}_\mathcal{A} & \times & \mathcal{L}_\mathcal{A} & \longrightarrow & \mathcal{L}_\mathcal{A} \\ & K & , & L & \longmapsto & K \cup L \ . \end{array}$$

2.1.5 Notation The operator $\cdot$ has higher precedence than $|$.

2.1.6 Remark Obviously, for $K, L \in \mathcal{L}_\mathcal{A}$, we have $K|L = L|K$.

2.1.7 Definition For

$$\mathbf{R} := \{(m,n) \,\big|\, m \in \mathbb{N}, n \in \mathbb{N} \cup \{\infty\}, 1 \leq n, 0 \leq m \leq n\}$$

and $K \in \mathcal{L}_\mathcal{A}$ we define

$$\forall (m,n) \in \mathbf{R} : K^{m,n} := \{v^k | k \in \mathbb{N}, m \leq k \leq n, v \in K\} \ .$$

2.1.8 Notation We use ^{m,n} with higher precedence than $\cdot$.

2.1.9 Definition The set

$$\begin{array}{rll} \mathcal{R} := \mathcal{R}_\mathcal{A} := \min\limits_{\subseteq} & \{R\big| & R \subseteq \mathcal{L}_\mathcal{A}, \\ & & \{\epsilon\} \in R, \\ & & \emptyset \in R, \\ & & \forall a \in \mathcal{A} : \{a\} \in R, \\ & & \forall K, L \in R : \forall (m,n) \in \mathbf{R} : K \cdot L, K|L, K^{m,n} \in R \\ & \} & \end{array}$$

is called the set of **regular languages** over $\mathcal{A}$.

To describe a regular language, we use a notation closely related to the construction of languages by means of concatenation and alternation: the regular expressions. In fact, the following definitions are merely a formalization of how we already describe a regular language *literally*.

2.1.10 Definition The **regular expression alphabet** is defined by

$$\mathcal{B} := \mathcal{B}_\mathcal{A} := \mathcal{A} \dot\cup \{|, \cdot, (,), \oslash, ^{\mathbf{m,n}} \,\big|\, (m,n) \in \mathbf{R}\} \ .$$

Again, ϵ denotes the empty word.

This enriches $\mathcal{A}$ with the symbols used to denote regular expressions. This is necessary since operations modifying the regular expressions will be introduced later, demanding to distinguish carefully between regular expressions denoted literally in this document, and their formal representation as a sequence of symbols taken from $\mathcal{B}$.

The following definition hides this purely technical issue, building a straightforward bridge between the abstract object of an expression and its notation.

2.1.11 Definition For all $x, y \in \mathcal{B}^*$, $(m, n) \in \mathbf{R}$ we define with strictly decreasing precedence:

$$\begin{array}{rccc} {}^{m,n}: & \mathcal{B}^* & \longrightarrow & \mathcal{B}^* \\ & x & \longmapsto & x^{\mathbf{m,n}} \end{array},$$

$$\begin{array}{rccccc} \cdot: & \mathcal{B}^* & \times & \mathcal{B}^* & \longrightarrow & \mathcal{B}^* \\ & x & , & y & \longmapsto & x \cdot y \end{array},$$

$$\begin{array}{rccccc} |: & \mathcal{B}^* & \times & \mathcal{B}^* & \longrightarrow & \mathcal{B}^* \\ & x & , & y & \longmapsto & (x|y) \end{array}.$$

The symbol $\oslash$ denotes $\oslash \in \mathcal{B}$.

2.1.12 Notation We write xy instead of $x \cdot y$ for convenience.

2.1.13 Definition The set of **regular expressions** is defined as

$$\begin{aligned} \mathfrak{R} := \mathfrak{R}_{\mathcal{A}} := \min_{\subseteq} \quad \{R \big| \quad & R \in \mathcal{L}_{\mathcal{B}}, \\ & \epsilon, \oslash \in R \\ & \forall a \in \mathcal{A} : a \in R, \\ & \forall x, y \in R : \forall (m, n) \in \mathbf{R} : xy, x|y, x^{m,n} \in R \\ \} \ . & \end{aligned}$$

2.1.14 Definition The **matching** relation "$\vDash$" $\subset \mathcal{A}^* \times \mathfrak{R}$ is defined as follows:

$$\begin{array}{l} \forall w \in \mathcal{A}^* : \forall x, y \in \mathfrak{R} : \forall (m, n) \in \mathbf{R} : \\ \quad \begin{array}{rcll} \epsilon & \vDash & \epsilon & \\ w & \vDash & xy & :\Longleftrightarrow \quad \exists u, v \in \mathcal{A}^*, w = uv : u \vDash x \wedge v \vDash y \\ w & \vDash & x|y & :\Longleftrightarrow \quad w \vDash x \vee w \vDash y \\ w & \vDash & x^{m,n} & :\Longleftrightarrow \quad \exists k \in \mathbb{N},\ m \leq k \leq n : w \vDash x^k \ . \end{array} \end{array}$$

The symbol $\oslash$ (read: *nothing*) is not matched by anything, i.e., it represents the empty language.

2.1.15 Definition A regular expression x **accepts** a language L, iff

$$\forall w \in L : w \vDash x$$

holds.

2.1.16 Definition We call two regular expressions **equivalent** , iff they accept the same languages. A regular expression is called equivalent to a language, iff it accepts exactly that language (i.e., accepts exactly the words in that language). For $x, y \in \mathfrak{R}, L \in \mathcal{R}$ this means that:

$$\begin{aligned} x = y &:\Longleftrightarrow \forall w \in \mathcal{A}^* : (w \vDash x \Longleftrightarrow w \vDash y) \ , \\ x = L &:\Longleftrightarrow \forall w \in \mathcal{A}^* : (w \vDash x \Longleftrightarrow w \in L) \ . \end{aligned}$$

2.1.17 Remark By construction, we have:

$$\begin{aligned} \forall L \in \mathcal{R} : \ \exists x \in \mathfrak{R} : \ L = x \ , \\ \forall x \in \mathfrak{R} : \ \exists L \in \mathcal{R} : \ L = x \ . \end{aligned}$$

2.1.18 Remark Due to Remark 2.1.6 we have $\forall x, y \in \mathfrak{R} : x|y = y|x$. And obviously $\forall x \in \mathfrak{R} : x\oslash = \oslash \, ^\wedge \, x|\oslash = x$ holds.

2.2 Trees & Forests

Since this thesis is about validation of XML documents, we need to leave the world of strictly sequential structures. XML document collections assure the structure of a forest, each single document is a tree.

In this section we formalize these forests, and augment the idea of regular languages to apply to them. In terms of graph theory, a forest is an undirected, acyclic and simple graph. However, we use another approach to formalize them with the additional property of the order in which the children of a node may appear.

Let us start from a finite set V of **nodes** (or **vertices**, hence the letter V). We build forests from these nodes, just like $V = \{a, \ldots, h\}$ is used to build the examples in Figure I on page 13. XML code that describes these forests is given as well. XML and XML Schema code snippets are marked throughout this document by a vertical rule on their left hand side.

2.2.1 Definition The alphabet we use to denote forests,

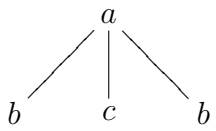
$$\mathcal{A} := \mathcal{A}_V := V \dot\cup \{\langle, \rangle\} \ ,$$

is called the **forest alphabet** over V. It contains all the nodes plus two additional symbols that are used to denote the subtree construction. Again, we use ϵ to denote the **empty forest** in $\mathcal{A}^*$.

2.2.2 Definition Two operations are defined on $\mathcal{A}^*$ with strictly decreasing precedence: The **subtree** construction

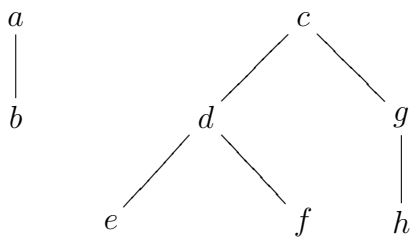
$$\begin{array}{lccccc} \langle\rangle : & V & \times & \mathcal{A}^* & \longrightarrow & \mathcal{A}^* \\ & a & , & x & \longmapsto & a\langle x\rangle \ , \end{array}$$

1. A forest made of only one tree.




```
<a>
  <b/>
  <c/>
  <b/>
</a>
```

2. Two separate trees in a forest.




```
<a> <b/> </a>
<c>
  <d>
    <e/> <f/>
  </d>
  <g> <h/> </g>
</c>
```

Figure I: Two Forests

which introduces hierarchical structure, and the **concatenation**

$$\begin{array}{rccccc} \cdot : & \mathcal{A}^* & \times & \mathcal{A}^* & \longrightarrow & \mathcal{A}^* \\ & x & , & y & \longmapsto & xy \ . \end{array}$$

2.2.3 Definition The set of **forests** over V is defined as

$$\begin{aligned} \mathcal{F} := \mathcal{F}_V := \min_{\subseteq} \quad \{F \big| \quad & F \in \mathcal{L}_{\mathcal{A}}, \\ & \epsilon \in F, \\ & \forall a \in V : \forall f, g \in F : a\langle f \rangle g \in F \\ \} \ . & \end{aligned}$$

2.2.4 Notation For convenience, we omit the brackets $\langle\rangle$, if a node does not have children, i.e., we identify the node $a \in V$ with the forest $a\langle\rangle \in \mathcal{F}$.

2.2.5 Remark Now the examples from above can be denoted as follows:

1. $a\langle bcb \rangle$.
2. $a\langle b \rangle c\langle d\langle ef \rangle g\langle h \rangle\rangle$.

2.2.6 Definition The equality relation "$=$" $\subseteq \mathcal{F} \times \mathcal{F}$ is defined by

$$\begin{aligned} \epsilon &= \epsilon \\ a\langle f \rangle g = b\langle h \rangle i \quad &:\Longleftrightarrow \quad a = b \wedge f = h \wedge g = i \ . \end{aligned}$$

2.2.7 Definition The forest $a\langle f \rangle g$ is called **tree** , iff $g = \epsilon$.

2.2.8 Definition The **size** of a forest is defined as

$$\begin{array}{rccl} \sigma: & \mathcal{F} & \longrightarrow & \mathbb{N} \\ & \epsilon & \longmapsto & 0 \\ & a\langle f\rangle g & \longmapsto & 1+\sigma f+\sigma g \end{array}$$

and simply counts the nodes the forest consists of.

2.3 Regular Expression meets Forest

2.3.1 Definition As in Definition 2.1.10, we use an extension of the alphabet $\mathcal{A}$ to represent the operations defined on the set of forests: The **regular expression alphabet for trees** is defined as

$$\mathcal{B} := \mathcal{B}_V := \mathcal{A}_V \dot\cup \{|, \cdot, (,), \oslash, ^{\mathbf{m,n}}, \langle, \rangle \mid (m,n) \in \mathbf{R}\} \ .$$

2.3.2 Definition Like above, operations on $\mathcal{B}^*$ are defined. These are, with strictly decreasing precedence:

$$\begin{array}{rccccl} \langle\rangle: & V & \times & \mathcal{B}^* & \longrightarrow & \mathcal{B}^* \\ & a & , & x & \longmapsto & a\langle x\rangle \ , \end{array}$$

$$\begin{array}{rccl} {}^{m,n}: & \mathcal{B}^* & \longrightarrow & \mathcal{B}^* \\ & x & \longmapsto & x^{\mathbf{m,n}} \ , \end{array}$$

$$\begin{array}{rccccl} \cdot: & \mathcal{B}^* & \times & \mathcal{B}^* & \longrightarrow & \mathcal{B}^* \\ & x & , & y & \longmapsto & (x\cdot y) \ , \end{array}$$

$$\begin{array}{rccccl} |: & \mathcal{B}^* & \times & \mathcal{B}^* & \longrightarrow & \mathcal{B}^* \\ & x & , & y & \longmapsto & (x|y) \ . \end{array}$$

For convenience, we write xy for $x \cdot y$. Again, the symbol $\oslash$ denotes $\oslash \in \mathcal{B}$.

2.3.3 Definition The **regular expressions for trees** over V are defined as

$$\begin{array}{rcll} \mathfrak{F} := \mathfrak{F}_V := \min\limits_{\subseteq} & \{F\big| & F \in \mathcal{L}_{\mathcal{B}}, & \\ & & \epsilon, \oslash \in F, & \\ & & \forall a \in V: \forall x,y \in F: xy, x|y, x^{m,n}, a\langle x\rangle \in F & \\ & \} \ . & & \end{array}$$

2.3.4 Definition The **matching** relation "$\vDash$" $\subset \mathcal{F} \times \mathfrak{F}$ is defined as follows:

$$\forall f \in \mathcal{F}: \forall x,y \in \mathfrak{F}: \forall (m,n) \in \mathbf{R}: \forall a,b \in V:$$

$$\begin{array}{rcll} \epsilon & \vDash & \epsilon & \\ f & \vDash & x|y & :\Longleftrightarrow \ f \vDash x \ ^\vee f \vDash y \\ f & \vDash & xy & :\Longleftrightarrow \ \exists g,h \in \mathcal{F}, f=gh: g \vDash x \ ^\wedge h \vDash y \\ f & \vDash & x^{m,n} & :\Longleftrightarrow \ \exists k \in \mathbb{N},\ m \le k \le n: f \vDash x^k \\ a\langle f\rangle & \vDash & b\langle x\rangle & :\Longleftrightarrow \ a=b \ ^\wedge f \vDash x \ . \end{array}$$

2.4 Context Free Grammars

To represent XML Schema descriptions, we need more than simple regular expressions. This is, because regular expressions lack any form of recursion, making it impossible to express recursive structures like, e.g., the correctly bracketed terms.

Instead, we use context-free grammars to model XML Schema descriptions.

2.4.1 Definition First, extend the alphabet $\mathcal{B}$ used so far by a finite set T of **nonterminals**:

$$\mathcal{B} := \mathcal{A}_V \dot\cup \{|, \cdot, (,), \oslash, \langle^{\mathbf{m},\mathbf{n}}, \rangle \mid (m,n) \in \mathbf{R}\} \dot\cup T$$

Likewise, extend $\mathfrak{F}$ to

$$\begin{aligned}\mathfrak{F} := \min_{\subseteq} \quad \{F \big| \quad & F \in \mathcal{L}_{\mathcal{B}}, \\ & \epsilon \in F, \quad \oslash \in F, \\ & \forall a \in V : \forall x, y \in F : xy, x|y, x^{\mathbf{m},\mathbf{n}}, a\langle x\rangle \in F, \\ & \forall \mathfrak{t} \in T : \mathfrak{t} \in F \qquad \leftarrow \boxed{\text{this is new}} \\ \} \ .\end{aligned}$$

From now on, until said otherwise, $\mathcal{B}$ and $\mathfrak{F}$ always refer to these extended versions.

Then, add a **lookup mapping**

$$\lambda : T \longrightarrow \mathfrak{F} \ .$$

A pair $G := (\mathfrak{s}, \lambda)$ of a dedicated **start symbol** $\mathfrak{s} \in T$ and a lookup mapping λ is called **grammar** . Note that specifying λ already specifies T, which is the preimage of λ.

Let $\mathcal{G}$ denote the **set of grammars**.

2.4.2 Definition The extension of the alphabet is reflected in the matching relation by adding one simple rule:

$$\forall f \in \mathcal{F} : \forall \mathfrak{t} \in T : f \vDash \mathfrak{t} :\Longleftrightarrow f \vDash \lambda\mathfrak{t} \ .$$

From now on, until said otherwise, $\vDash$ always refers to this extended matching.

2.4.3 Notation For convenience, we write $f \vDash (\mathfrak{s}, \lambda)$ for any grammar $(\mathfrak{s}, \lambda)$ with $f \vDash \mathfrak{s}$.

Potentially, this definition might lead to infinite structures when trying to figure out if a forest matches a given grammar, just consider the following example:

$$V := \{a\} \qquad T := \{\mathfrak{s}\} \qquad \lambda\mathfrak{s} := \mathfrak{s}a|\epsilon \ .$$

Application "$\rightsquigarrow$" of the lookup function λ according to the given definitions results in the following:

$$\begin{aligned} & a \vDash \mathfrak{s} \\ \rightsquigarrow \quad & a \vDash \mathfrak{s}a|\epsilon \\ \rightsquigarrow \quad & a \vDash (\mathfrak{s}a|\epsilon)a|\epsilon \\ & \dots \\ \rightsquigarrow \quad & a \vDash (\dots(\mathfrak{s}a|\epsilon)\dots a|\epsilon)a|\epsilon \ . \end{aligned}$$

Fortunately, the XML Schema standard restricts its grammars to avoid such cases (see Appendix A.1).

2.5 The Pre/Post Enumeration

We do not want to handle the XML documents in their text form. Instead documents are stored as a relation inside a RDBMS like, e.g., [9]. The validation algorithm proposed in this thesis is built to work directly on such a relational encoding. One encoding, the pre/post enumeration which was also used to develop the proposed algorithm, is introduced here. Note, however, that the algorithm does not require the pre/post enumeration. Any encoding with similar properties will be sufficient (see *Other enumerations* on page 32).

2.5.1 Definition For $p, q \in \mathbb{N}$ the **pre/post enumeration** is defined by

$$\begin{array}{rccl} \varphi_{p,q}: & \mathcal{F} & \longrightarrow & \mathcal{P}(\mathbb{N} \times V \times \mathbb{N}) \\ & \epsilon & \longmapsto & \emptyset \\ & a\langle f \rangle g & \longmapsto & \{(p, a, x)\} \cup \varphi_{p+1,q} f \cup \varphi_{y,x+1} g \end{array}$$

where

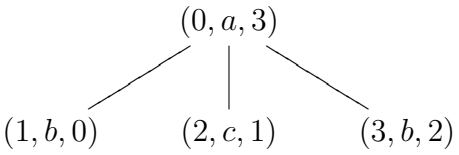
$$\begin{aligned} x &:= 1 + \max_{<} \{l \mid (.,.,l) \in \varphi_{p+1,q} f\} \cup \{q-1\} \\ y &:= 1 + \max_{<} \{l \mid (l,.,.) \in \varphi_{p+1,q} f\} \cup \{p\} \ . \end{aligned}$$

Additionally, we call $\varphi := \varphi_{0,0}$ the **normalized** pre/post enumeration.

This formalizes two possible *left-to-right depth first search* (DFS) enumerations. Each node a is decorated with two counters during the DFS run, resulting in a (p, a, q) tuple: The *pre* value p is annotated and increased whenever DFS *reaches* a node for the first time. The *post* value q is annotated and increased whenever DFS *leaves* a node for the last time. This enumeration was introduced by [5].

The two forests from Figure I on page 13 are enumerated as depicted in Figure II on page 17.

1. $a\langle bcb\rangle$:



2. $a\langle b\rangle c\langle d\langle ef\rangle g\langle h\rangle\rangle$:

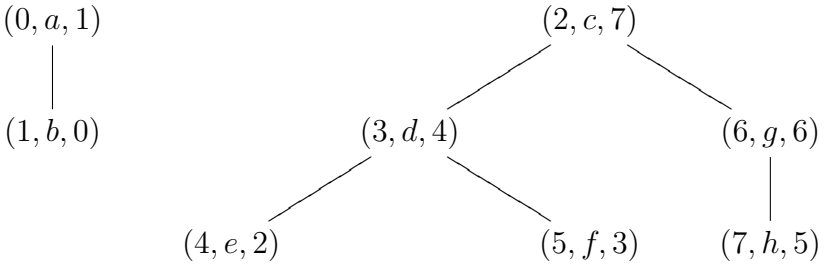


Figure II: Enumerated Forests from Figure I on page 13

2.5.2 Notation The enumerated nodes will be referred to simply by calling them *nodes*, too. Let us define some "access functions" to the pre value, name, and post value of these nodes. For $\alpha = (p, a, q) \in \mathbb{N} \times V \times \mathbb{N}$ we define

$$\grave{\alpha} := p \qquad \breve{\alpha} := a \qquad \acute{\alpha} := q$$

For any relational symbol $\equiv$ and any accent $\tilde{} \in \{\grave{}, \breve{}, \acute{}\}$ we write

$$\alpha \tilde{\equiv} \beta :\Longleftrightarrow \tilde{\alpha} \equiv \tilde{\beta} \ .$$

Additionally, for an enumerated forest $M = \varphi f$, let

$$\tau_M := \min_{\breve{<}} M$$

hence $\breve{\tau}_M$ is the name of the leftmost root node in an enumerated forest.

2.5.3 Remark Due to these numbers, each node partitions the remaining nodes in the enumerated forest into four classes. This can be illustrated by the **pre/post plane**. Construe the pre/post values being x/y coordinates of the according nodes in a two dimensional plane. The nodes from the second example are plotted in Figure III on page 18.

For example, the nodes e and f are **descendants** of d — which is defined by having greater pre values and smaller post values than the related node — and thus can be found in the quadrant to the lower right of d.

A formal definition of the descendants of α is

$$\text{descendants } \alpha := \{\beta \mid \beta \grave{>} \alpha \wedge \beta \acute{<} \alpha\} \ .$$

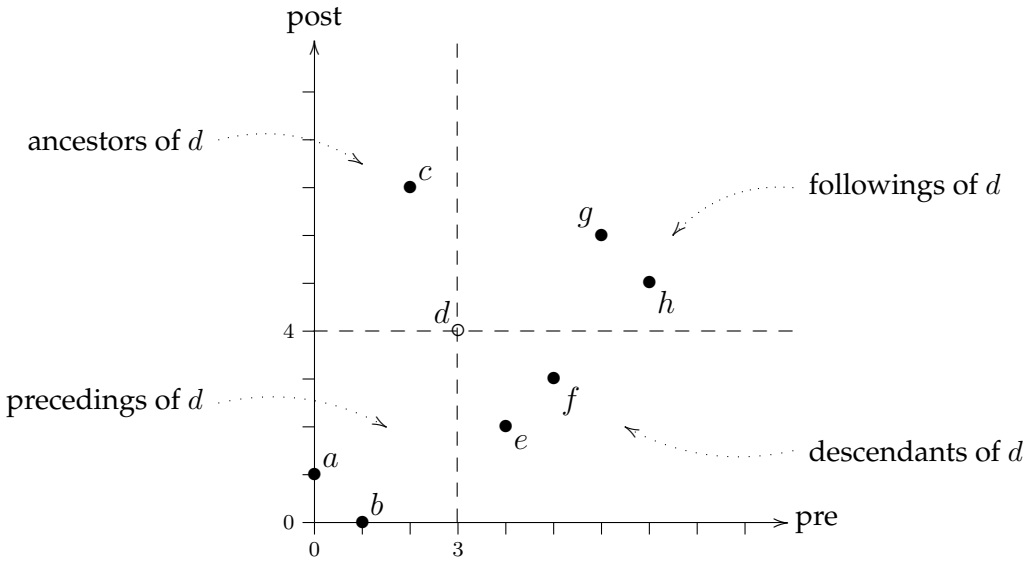


Figure III: The pre/post plane

The remaining classes **ancestors** , **precedings** and **followings** are defined analogous by the remaining three possibilities of having greater or smaller pre/post values, as can be seen in the picture.

2.5.4 Corollary Due to construction, we have

$$\forall p, q \in \mathbb{N} : \forall f \in \mathcal{F} : \#\varphi_{p,q} f = \sigma f \ .$$

That is, the size of the set created by φ for a given forest f is independent from its parameters p and q, but solely depends on the size of the forest.

2.5.5 Definition The sets

$$\mathcal{N} := \{M \mid \exists p, q \in \mathbb{N} : \exists f \in \mathcal{F} : \varphi_{p,q} f = M\} \ ,$$
$$\mathcal{N}_0 := \{M \mid \exists f \in \mathcal{F} : \varphi f = M\}$$

are called the **set of (normalized) pre/post enumerations** over V.

2.5.6 Remark For the construction of the pre/post enumeration (Definition 2.5.1), we obviously have for $\alpha = (p, a, q)$:

$$\forall \beta \in \varphi_{p+1,q} f : \beta \grave{>} \alpha \wedge \beta \acute{<} \alpha$$
$$\forall \beta \in \varphi_{y,x+1} g : \beta \grave{>} \alpha \wedge \beta \acute{>} \alpha \ .$$

This is simply another way to see how the pre/post enumeration first processes the descendants of a node and next the followings, hence being a DFS.

2.5.7 Definition For $M \in \mathcal{N}$ let us define

$$M_< := \{\beta \mid \beta \in M, \beta \acute{<} \tau_M\}$$
$$M_> := \{\beta \mid \beta \in M, \beta \acute{>} \tau_M\} \ .$$

Note that τ_M is the smallest element in M according to its *pre*order index. However, the sets $M_</M_>$ are built from elements smaller/greater than τ_M according to their *post*order index. So $M_<$ is the set representing the descendant forest, $M_>$ the following forest, with respect to the node $\check{\tau}_M$.

2.5.8 Remark Obviously,

$$\forall M \in \mathcal{N} : M = M_< \dot\cup \{\tau_M\} \dot\cup M_>$$

holds. From Definition 2.5.1 we have

$$\forall M = \varphi_{p,q} a\langle g\rangle h \in \mathcal{N} : \exists i,j,k,l \in \mathbb{N} : \varphi_{i,j} g = M_< \wedge \varphi_{k,l} h = M_> \ .$$

2.5.9 Definition The **similarity** of enumerations "$\sim$" $\subset \mathcal{N} \times \mathcal{N}$ is defined by

$$\begin{aligned} \forall M,N \in \mathcal{N} : M \sim N \quad :\Longleftrightarrow \quad & M = N = \emptyset \\ \vee \quad & M_< \sim N_< \wedge M_> \sim N_> \wedge \check{\tau}_M = \check{\tau}_N \ . \end{aligned}$$

This definition formalizes the "structural equivalence" of forests, independent of the start indices used for their enumeration.

2.5.10 Remark The two implications

$$M = N \Rightarrow M \sim N \qquad\qquad M \sim N \Rightarrow \#M = \#N$$

are trivially true.

2.5.11 Theorem Enumerations are similar, iff the corresponding forests are equal, i.e.,

$$\forall f,f' \in \mathcal{F} : \forall p,q,p',q' \in N : \big(\varphi_{p,q} f \sim \varphi_{p',q'} f' \Longleftrightarrow f = f'\big) \ .$$

The proof is deferred to 2.5.13 below.

2.5.12 Corollary

$$\mathcal{N}_0 \approx \mathcal{F}$$

since $\forall f,f' \in \mathcal{F} : \big(\varphi f = \varphi f' \Longleftrightarrow f = f'\big)$.

2.5.13 Proof of Theorem 2.5.11.
Due to Corollary 2.5.4 and Definition 2.2.6 we have

$$\begin{aligned} &\forall f \in \mathcal{F} : \forall p,q \in \mathbb{N} : \sigma f = \#\varphi_{p,q} f \\ &\forall f,g \in \mathcal{F} : f = g \Rightarrow \sigma f = \sigma g \ . \end{aligned}$$

So we can use induction over σf. Let $M := \varphi_{p,q} f$ and $M' := \varphi_{p',q'} f'$ for some $p,q,p',q' \in \mathbb{N}$.

▷ **Case I** $\sigma f = 0$.
There is nothing to show in this case, since

$$\begin{aligned} f = f' &\iff f = \epsilon \wedge f' = \epsilon \\ &\iff M = \emptyset \wedge M' = \emptyset \ . \end{aligned}$$

▷ **Case II** $\sigma f > 0$.
Let $f = a\langle g \rangle h$, $f' = a'\langle g' \rangle h'$. Obviously $M \neq \emptyset$. Note that $\sigma g, \sigma h < \sigma f$. Induction yields

$$\begin{aligned} M \sim M' &\iff \check{\tau}_M = \check{\tau}_{M'} \wedge M_< \sim M'_< \wedge M_> \sim M'_> \\ &\iff a = a' \wedge g = g' \wedge h = h' \\ &\iff f = f' \ . \end{aligned}$$

□

2.5.14 Notation When referring to an enumeration $N \in \mathcal{N}, N = \{\nu_1, \ldots, \nu_k\}$ as being **pre-sorted** , this means that w.l.o.g. $\forall i \in \mathbb{N}, 1 \leq i < k : \nu_i \grave{<} \nu_{i+1}$. This is not a property of the set. It rather specifies how its elements are named.

2.6 Summary

We have introduced the basic definitions to be used throughout this paper:

The Forests $\mathcal{F}$ over a set of Vertices V have been defined, as well as regular expressions $\mathfrak{F}$ and context free grammars $\mathcal{G}$ to restrict their shape. The matching relation "$\vDash$" describes which forests fulfill the restrictions of certain grammars.

The pre/post enumeration was introduced as a way to transform a forest f into a relational encoding $\varphi f \subset \mathbb{N} \times V \times \mathbb{N}$, which allows storage in a RDBMS. The pre/post plane illustrates how the relationship of nodes coincides with the order of their enumeration.

Chapter 3

The Derivation

The proposed algorithm does not use a finite state automaton derived from the grammar for validation. Instead the concept of "deriving a regular expression" is used, i.e., the algorithm looks at the first node of the document, and transforms the given regular expression to describe the potential remains of a document beginning with the node found. This is repeated until all nodes have been processed in document order. The original document matches the original regular expression, if and only if the expression generated this way is matched by the empty forest.

3.1 Guards

Since we want to operate on the enumeration of the forest directly, without "materializing" the trees, we need to incorporate information about the descendant- or following-relationship between nodes into the derivation process.

The idea is to surround a regular expression that must be matched by — for example — the children of a node a with *guards*. These guards protect an expression from being matched by a node that does not satisfy the demanded constraints. Informally this looks like:

$$b_{\text{from } a\langle b\rangle} \vDash [b]_{\text{children of } a} \quad , \qquad\qquad b_{\text{from } ab} \nvDash [b]_{\text{children of } a} \quad .$$

Of course, these guards utilize the pre/post enumeration to constrain matchings. Therefore, we extend the alphabet used for regular expressions once more:

3.1.1 Definition The set

$$\mathbf{L} := \{(\underline{p}, \overline{p}, \underline{q}, \overline{q}) \;\big|\; \underline{p}, \overline{p}, \underline{q}, \overline{q} \in \mathbb{N} \cup \{-\infty, \infty\}, \underline{p} < \overline{p}, \underline{q} < \overline{q}\}$$

is called the **set of limits**. The alphabet $\mathcal{B}$ used so far is extended to

$$\begin{aligned}\mathcal{B} := \mathcal{A}_V \,\dot\cup\, \{|, \cdot, (,), \oslash, ^{\mathbf{m},\mathbf{n}}, \langle, \rangle \;\big|\; (m,n) \in \mathbf{R}\} \\ \dot\cup\, \{{}_{\underline{\mathbf{p}}}^{\overline{\mathbf{p}}}[,]_{\underline{\mathbf{q}}}^{\overline{\mathbf{q}}} \;\big|\; (\underline{p}, \overline{p}, \underline{q}, \overline{q}) \in \mathbf{L}\} \qquad \leftarrow \boxed{\text{this is new}}\end{aligned}$$

by a set of symbols called **guards**. From now on, $\mathcal{B}$ always refers to this extended version. The impact of these guards is that an expression surrounded by $\overset{\overline{\mathbf{p}}}{\underset{\underline{\mathbf{p}}}{[}}$ and $\overset{\overline{\mathbf{q}}}{\underset{\underline{\mathbf{q}}}{]}}$ can only be derived successfully according to nodes (p, a, q) which satisfy $\underline{p} < p < \overline{p}$ and $\underline{q} < q < \overline{q}$.

Therefore, $\mathfrak{F}$ is extended to

$$\begin{aligned}
\mathfrak{X} := \mathfrak{X}_V := \min_{\subseteq} \quad \{F \big| \quad & F \in \mathcal{L}_{\mathcal{B}}, \\
& \epsilon, \oslash \in F, \\
& \forall a \in V : \forall x, y \in F : xy, x|y, x^{\mathbf{m},\mathbf{n}}, a\langle x \rangle \in F \\
& \forall \mathfrak{t} \in T : \mathfrak{t} \in F \\
\boxed{\text{new}} \rightarrow \quad & \forall (\underline{p}, \overline{p}, \underline{q}, \overline{q}) \in \mathbf{L} : \forall x \in F : \overset{\overline{\mathbf{p}}\ \overline{\mathbf{q}}}{\underset{\underline{\mathbf{p}}\ \underline{\mathbf{q}}}{[}} x \overset{\overline{\mathbf{p}}\ \overline{\mathbf{q}}}{\underset{\underline{\mathbf{p}}\ \underline{\mathbf{q}}}{]}} \in F \\
\} \ .
\end{aligned}$$

The functions defined on $\mathfrak{F}$ used so far shall be continued to functions on $\mathfrak{X}$. Additionally , we add the function

$$\begin{array}{rccccc}
[] : & \mathbf{L} & \times & \mathfrak{X} & \longrightarrow & \mathfrak{X} \\
& (\underline{p}, \overline{p}, \underline{q}, \overline{q}) & , & x & \longmapsto & \overset{\overline{p}\ \overline{q}}{\underset{\underline{p}\ \underline{q}}{[}} x \overset{\overline{p}\ \overline{q}}{\underset{\underline{p}\ \underline{q}}{]}} := \overset{\overline{\mathbf{p}}\ \overline{\mathbf{q}}}{\underset{\underline{\mathbf{p}}\ \underline{\mathbf{q}}}{[}} x \overset{\overline{\mathbf{p}}\ \overline{\mathbf{q}}}{\underset{\underline{\mathbf{p}}\ \underline{\mathbf{q}}}{]}} \ .
\end{array}$$

3.1.2 Notation Surrounding an expression with guards alters precedence just like usual parenthesis do.

Note that a new letter $\mathfrak{X}$ is introduced to distinguish between the regular expressions $\mathfrak{F}$ that occur in grammars in $\mathcal{G}$ and the extended regular expressions $\mathfrak{X}$ that are created during the derivation process. Also, note that $\mathfrak{F} \subset \mathfrak{X}$.

3.1.3 Notation For convenience, we omit the limits if they are extremal, e.g.,

$$\text{we write } \underset{2}{[} x \overset{15}{]} \text{ for } \overset{\infty}{\underset{2}{[}} x \overset{15}{\underset{-\infty}{]}} \ .$$

To enhance readability, for a node $\alpha = (p, a, q) \in N \in \mathcal{N}$, we omit the accents used to access the values p and q, when used in conjunction with a guard, e.g.,

$$\text{we write } \underset{\alpha}{[} x \overset{\alpha}{]} \text{ for } \underset{\grave{\alpha}}{[} x \overset{\acute{\alpha}}{]} \ .$$

This is safe, since we never use the pre value with a post guard, or vice versa.

We need a way to say *"Nothing is left that must be matched"*, but we cannot decide if the empty forest matches an expression in $\mathfrak{X}$, since we have not defined a matching on these[1]. So we introduce the idea of *nullable* expressions:

[1] In fact, a direct definition of matching between $\mathcal{F}$ and $\mathfrak{X}$ is difficult, since it is not obvious how to handle the guards.

3.1.4 Definition The **set of nullable expressions** is defined to be the minimum set $\mathcal{E} \subseteq \mathfrak{X}$ that satisfies

$$\begin{array}{l}\forall x, y \in \mathfrak{X} : \forall (m,n) \in \mathbf{R} : \forall (\underline{p}, \overline{p}, \underline{q}, \overline{q}) \in \mathbf{L} : \\ \quad \begin{array}{rcll} \epsilon & \in & \mathcal{E} & \\ xy & \in & \mathcal{E} & :\Longleftrightarrow \quad x \in \mathcal{E} \,\wedge\, y \in \mathcal{E} \\ x|y & \in & \mathcal{E} & :\Longleftrightarrow \quad x \in \mathcal{E} \,\vee\, y \in \mathcal{E} \\ x^{m,n} & \in & \mathcal{E} & :\Longleftrightarrow \quad m = 0 \,\vee\, x \in \mathcal{E} \\ \overset{\overline{p}\ \overline{q}}{\underset{\underline{p}\ \underline{q}}{[x]}} & \in & \mathcal{E} & :\Longleftrightarrow \quad x \in \mathcal{E} \ . \end{array}\end{array}$$

Apart from the last line, this is similar to checking for acceptance of ϵ for $x, y \in \mathcal{F}$.

3.2 The Derivation

The derivation function is applied to the nodes representing the forest in document order, i.e., ordered strictly increasing according to the pre value. We define the derivation for single nodes, and apply it once for each of them.

3.2.1 Definition For any $N \in \mathcal{N}$, the **derivation of a regular expression according to a single node** $(p, a, q) \in N$ is defined as

$$\begin{array}{rcl} \partial_{p,a,q} : \ \mathfrak{X} & \longrightarrow & \mathfrak{X} \\ \oslash,\ \epsilon & \longmapsto & \oslash \\ \overset{\overline{p}\ \overline{q}}{\underset{\underline{p}\ \underline{q}}{[x]}} & \longmapsto & \begin{cases} \overset{\overline{p}}{\underset{\underline{p}}{[}}\partial_{p,a,q}x\overset{\overline{q}}{\underset{\underline{q}}{]}} & \text{if } \underline{p} < p < \overline{p} \,\wedge\, \underline{q} < q < \overline{q} \\ \oslash & \text{otherwise} \end{cases} \\ b\langle x \rangle & \longmapsto & \begin{cases} \overset{q}{\underset{p}{[x]}} & \text{if } a = b \\ \oslash & \text{otherwise} \end{cases} \\ xy & \longmapsto & \begin{cases} \partial_{p,a,q}x\underset{p\ \ q}{[y]}|\partial_{p,a,q}y & \text{if } x \in \mathcal{E} \\ \partial_{p,a,q}x\underset{p\ \ q}{[y]} & \text{otherwise} \end{cases} \\ x|y & \longmapsto & \partial_{p,a,q}x|\partial_{p,a,q}y \\ x^{m,n} & \longmapsto & \begin{cases} \partial_{p,a,q}x & \text{if } n = 1 \\ \partial_{p,a,q}x\underset{p\qquad q}{[x^{0,n-1}]} & \text{if } n > 1 \,\wedge\, m = 0 \\ \partial_{p,a,q}x\underset{p\qquad\qquad q}{[x^{m-1,n-1}]} & \text{otherwise, i.e., } n > 1 \,\wedge\, m > 0 \end{cases} \\ \mathfrak{t} & \longmapsto & \partial_{p,a,q}(\lambda \mathfrak{t}) \end{array}$$

where $x, y \in \mathfrak{X}$, $(m,n) \in \mathbf{R}$, $b \in V$, $\mathfrak{t} \in T$, and $\infty - 1 := \infty$.

3.2.2 Notation Application of ∂ has precedence higher than $\cdot$, but lower than ^{m,n}. So the list of operations on regular expressions, with strictly decreasing precedence is:

$$\langle\rangle \quad , \quad ^{m,n} \quad , \quad \partial \quad , \quad \cdot \quad , \quad |$$

First, let us have a look at the rules given in this definition. Remember that we want to transform the given regular expression into a regular expression which describes the potential remains of a document beginning with the node found.

It is evident that a regular expression that is not matched by any forest will not turn into an expression that accepts something. Hence $\oslash \longmapsto \oslash$.

An expression that accepts only the empty forest (ϵ) cannot be matched by a forest that is not empty. So the derivation according to a node must yield $\oslash$ to signal that nothing will be able to validate against the remaining expression.

The next rule constitutes the semantics of guards. If the guards surrounding an expression are satisfied by the pre and post values of the node, the derivation is forwarded to their inside. The guards are not removed, since they might be necessary to constrain more than one node, e.g., if a node must have several children. If the limits are violated, $\oslash$ is returned, since the derivation was fed with a node that did not satisfy the imposed structure.

The subtree construction $b\langle x\rangle$ is the first one that introduces guards. A forest matching this expression must begin with a b node. The remaining forest must match the expression x, and must consist of *children* of b, hence the guards imposing a lower limit on the pre value, and an upper limit on the post value.

The sequence construct xy requires distinction of two cases. If $x \notin \mathcal{E}$ we know that the x expression must be derived according to the node (p, a, q), and that the y expression must be derived according to *followings* of a. If, however, $x \in \mathcal{E}$, we cannot determine immediately if the a node is meant to match in the x or in the y expression. To consider both matches, a $|$ construct is used in the latter case.

The choice construct $x|y$ represents the two alternative branches the derivation might follow. So the application is forwarded to the two operands equally.

The so called *occurrence constraint* construct $x^{m,n}$ is not treated as $x^m|x^{m+1}|\ldots|x^n$, since this would lead to severe difficulties, if x was nullable, or $n = \infty$. Note that the derivation is applied to x only, followed by a guarded expression that keeps the "remaining $x$es" to be matched.

Finally, a nonterminal t is resolved and the derivation goes on with the defining expression.

3.2.3 Definition The derivation according to single nodes is extended to the **derivation of a regular expression according to an enumeration**

$$N = \{\nu_1, \nu_2, \ldots, \nu_n\} \in \mathcal{N} \quad \text{, pre-sorted}$$

by

$$\partial_N := \partial_{\nu_n} \circ \partial_{\nu_{n-1}} \circ \ldots \circ \partial_{\nu_1} \quad .$$

This implements the processing of the pre/post table in pre order. Note that $\partial_\emptyset = \mathrm{id}$. This makes sense, since $\emptyset \in \mathcal{N}$ represents the empty forest.

3.2.4 Definition For convenience, let us also define the **derivation of a regular expression according to a forest** $f \in \mathcal{F}$ by

$$\partial_f := \partial_{\varphi f} \ .$$

3.3 Validation

3.3.1 Theorem This main theorem states that a forest f matches a regular expression x, iff the derivation of x according to f is nullable:

$$\forall x \in \mathfrak{F} : \forall f \in \mathcal{F} : (f \vDash x \iff \partial_f x \in \mathcal{E}) \ .$$

The proof is deferred to 3.3.4 below.

Note that the matching relation on $\mathcal{F} \times \mathfrak{F}$ is used. ∂ is well defined on $\mathfrak{F}$, since $\mathfrak{F} \subset \mathfrak{X}$. The introduction of $\mathfrak{X}$ above is necessary, since the application of ∂ can yield expressions that contain guards, however, $\mathfrak{X}$ is intended for "internal use" by the algorithm only. From an exterior view, we do not mind if any of the $\mathfrak{X}$ expressions created during derivation might "match" anything.

First, we show that the nullable test of a derived expression is independent from the enumeration used, i.e., that we only rely on the structural properties of a given forest, not its representation.

3.3.2 Lemma

$$\forall M, N \in \mathcal{N},\ M \sim N : \forall x \in \mathfrak{F} : \left(\partial_M x \in \mathcal{E} \iff \partial_N x \in \mathcal{E}\right) \ .$$

3.3.3 Proof of Lemma 3.3.2.
$M \sim N \Rightarrow \#M = \#N =: k$. Let $M = \{\mu_1, \ldots, \mu_k\}$ and $N = \{\nu_1, \ldots, \nu_k\}$, both pre-sorted. The similarity yields that $\forall i \in \mathbb{N}, 1 \leq i \leq k : \check{\mu}_i = \check{\nu}_i$.

Furthermore, with respect to any given node in the enumerated forest, both enumerations partition the remaining nodes into the same pre/post classes, i.e., $\forall \prec \in \{\grave{<}, \grave{>}, \acute{<}, \acute{>}\}$:

$$\forall i \in \mathbb{N}, 1 \leq i \leq k : \{j \mid \mu_i \prec \mu_j\} = \{j \mid \nu_i \prec \nu_j\} \ .$$

So all parameters driving the derivation process — name and order of the nodes, as well as structural relationships between the nodes, such as following and descendant — remain invariant. □

3.3.4 Proof of Theorem 3.3.1.
The equivalence stated in the main theorem can be proven by induction over the structure of expressions in $\mathfrak{F}$. Let z be such an expression.

▷ **Case I** $z = \oslash$.
It is evident that

$$\forall f \in \mathcal{F} : f \nvDash z \qquad\qquad \forall N \in \mathcal{N} : \partial_N z = \oslash \notin \mathcal{E}$$

hold. So there is nothing to prove.

▷ **Case II** $z = \epsilon$.
There is only one forest that matches ϵ — the empty forest. This yields the equivalence

$$f \vDash z \iff f = \epsilon \iff \varphi f = \emptyset \iff \partial_f = \mathrm{id} \quad .$$

With this, the two directions of the theorem can be verified by

$$\begin{array}{ll} \partial_f = \mathrm{id} \Rightarrow \partial_f z \in \mathcal{E} & \text{since } z = \epsilon \\ \partial_f \neq \mathrm{id} \Rightarrow \partial_f z = \oslash \notin \mathcal{E} & \text{since } \forall \alpha \neq \epsilon : \partial_\alpha \epsilon = \oslash \end{array}$$

which together yield the desired equivalence.

▷ **Case III** $z = x|y$.
By Definition 2.3.4, we have

$$f \vDash z \iff f \vDash x \mathbin{\vee} f \vDash y \quad .$$

So we can infer as follows:

$$\begin{array}{lll} f \vDash x \mathbin{\vee} f \vDash y & \iff \partial_f x \in \mathcal{E} \mathbin{\vee} \partial_f y \in \mathcal{E} & \text{due to induction} \\ & \iff \partial_f x | \partial_f y \in \mathcal{E} & \text{by definition of } \mathcal{E} \\ & \iff \partial_f (x|y) \in \mathcal{E} & \text{by definition of } \partial \\ & \iff \partial_f z \in \mathcal{E} \quad . & \end{array}$$

▷ **Case IV** $z = xy$.
We handle both directions of the theorem individually:

▷ **Part IV.a** "$\Longrightarrow$".
Assume $f \vDash z$. We need to show that $\partial_f z \in \mathcal{E}$.

$$f \vDash z \Rightarrow \exists g, h \in \mathcal{F}, f = gh : g \vDash x \mathbin{\wedge} h \vDash y \quad .$$

For those g and h we have, due to induction, that

$$\partial_g x \in \mathcal{E} \mathbin{\wedge} \partial_h y \in \mathcal{E} \quad .$$

Remark 2.5.8 and Theorem 2.5.11 yield

$$\exists M, N \in \mathcal{N}, \varphi f = M \dot\cup N : M \sim \varphi g \mathbin{\wedge} N \sim \varphi h$$

and for those M and N, Lemma 3.3.2 yields

$$\partial_M x \in \mathcal{E} \mathbin{\wedge} \partial_N y \in \mathcal{E} \quad .$$

Trivially (or see Remark 2.5.6),

$$\forall \mu \in M : \forall \nu \in N : \mu \dot{<} \nu \wedge \mu \acute{<} \nu \tag{1}$$

holds. Let $M = \{\mu_1, \ldots \mu_m\}$, $N = \{\nu_1, \ldots \nu_n\}$, both pre-sorted. It is sufficient to prove that $\partial_N \partial_M(xy) \in \mathcal{E}$, because this yields $\partial_f \in \mathcal{E}$. Let us apply the derivation in single steps: $\partial_M(xy) = \partial_{\mu_m} \ldots \partial_{\mu_2} \partial_{\mu_1}(xy)$. Application of $\partial_{\mu_1}(xy)$ yields

$$\partial_{\mu_1} x \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \underbrace{|\partial_{\mu_1} y}_{x \in \mathcal{E}}$$

where the under-bracketed expression is not present (i.e., the alternative term equals $\oslash$), if the subscripted statement does not hold. Then, application of ∂_{μ_2} yields

$$\partial_{\mu_2}\partial_{\mu_1} x \underset{\mu_2\mu_1}{[}\ \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \underset{\mu_1\mu_2}{]}\ \underbrace{|\partial_{\mu_2} \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}}_{\partial_{\mu_1} x \in \mathcal{E}}\ \underbrace{|\partial_{\mu_2}\partial_{\mu_1} y}_{x \in \mathcal{E}} \quad .$$

Likewise, the application of ∂_{μ_3} yields

$$\partial_{\mu_3} \ldots \partial_{\mu_1} x \underset{\mu_3}{[}\ \cdots\ \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \cdots\ \underset{\mu_3}{]}\ \underbrace{|\partial_{\mu_3} \underset{\mu_2\mu_1}{[}\ \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \underset{\mu_1\mu_2}{]}}_{\partial_{\mu_2}\partial_{\mu_1} x \in \mathcal{E}}\ \underbrace{|\partial_{\mu_3}\partial_{\mu_2} \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}}_{\partial_{\mu_1} x \in \mathcal{E}}\ \underbrace{|\partial_{\mu_3} \ldots \partial_{\mu_1} y}_{x \in \mathcal{E}}$$

and so on, until finally the application of ∂_{μ_m} yields

$$\begin{aligned} \partial_M(xy) \quad = \quad & \partial_{\mu_m} \ldots \partial_{\mu_1} x \underset{\mu_m}{[}\ \cdots\ \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \cdots\ \underset{\mu_m}{]} \\ & \underbrace{|\partial_{\mu_m} \underset{\mu_{m-1}}{[}\ \cdots\ \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \cdots\ \underset{\mu_{m-1}}{]}}_{\partial_{\mu_{m-1}} \ldots \partial_{\mu_1} x \in \mathcal{E}}\ \underbrace{|\partial_{\mu_m}\partial_{\mu_{m-1}} \underset{\mu_{m-2}}{[}\ \cdots\ \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \cdots\ \underset{\mu_{m-2}}{]}}_{\partial_{\mu_{m-2}} \ldots \partial_{\mu_1} x \in \mathcal{E}} \\ & |\ldots \underbrace{|\partial_{\mu_m} \ldots \partial_{\mu_2} \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}}_{\partial_{\mu_1} x \in \mathcal{E}}\ \underbrace{|\partial_{\mu_m} \ldots \partial_{\mu_1} y}_{x \in \mathcal{E}} \quad . \end{aligned}$$

So, for

$$\begin{aligned} A &:= \underset{\mu_m}{[}\ \cdots\ \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \cdots\ \underset{\mu_m}{]} \\ R &:= \oslash \underbrace{|\partial_{\mu_m} \underset{\mu_{m-1}}{[}\ \cdots\ \underset{\mu_1}{[}\, y \,\underset{\mu_1}{]}\ \cdots\ \underset{\mu_{m-1}}{]}}_{\partial_{\mu_{m-1}} \ldots \partial_{\mu_1} x \in \mathcal{E}} |\ldots \underbrace{|\partial_{\mu_m} \ldots \partial_{\mu_1} y}_{x \in \mathcal{E}} \end{aligned}$$

we have

$$\begin{aligned} \partial_N \partial_M(xy) \quad &= \quad \partial_N(\partial_M x A | R) \\ &= \quad \partial_N(\partial_M x A) \quad | \quad \partial_N R \end{aligned}$$

which is nullable, if $\partial_N(\partial_M x A) \in \mathcal{E}$.

$$\begin{aligned}\partial_N(\partial_M x A) &= \partial_{\nu_n} \dots \partial_{\nu_2} \partial_{\nu_1}(\partial_M x A) \\ &= \partial_{\nu_n} \dots \partial_{\nu_2}(\partial_{\nu_1}\partial_M x \underset{\nu_1}{[}\, A \,\underset{\nu_1}{]} \,|\partial_{\nu_1} A) \quad \text{since } \partial_M x \in \mathcal{E} \\ &= \partial_{\nu_n} \dots \partial_{\nu_2}(\partial_{\nu_1}\partial_M x \underset{\nu_1}{[}\, A \,\underset{\nu_1}{]}\,) \quad | \quad \partial_{\nu_n} \dots \partial_{\nu_2}(\partial_{\nu_1} A)\end{aligned}$$

which is nullable, if $\partial_N A \in \mathcal{E}$.

$$\begin{aligned}\partial_N A &= \underset{\mu_m}{[}\, \cdots \underset{\mu_1}{[}\, \partial_N y \,\underset{\mu_1}{]} \cdots \,\underset{\mu_m}{]} \quad \text{due to } (1) \\ &\in \mathcal{E} \qquad\qquad\qquad\qquad\quad \text{since } \partial_N y \in \mathcal{E}\ .\end{aligned}$$

▷ **Part IV.b** "$\Longleftarrow$".
Assume $\partial_f z \in \mathcal{E}$. We need to show that $f \vDash z$. Let $\varphi f = \{\nu_1, \dots, \nu_n\}$, pre-sorted.

$$\partial_f(xy) = \partial_{\nu_n} \dots \partial_{\nu_1}(xy)$$

Like above, we get

$$\begin{aligned}\partial_f(xy) &= \partial_{\nu_n} \dots \partial_{\nu_1} x \underset{\nu_n}{[}\, \cdots \underset{\nu_1}{[}\, y \,\underset{\nu_1}{]} \cdots \,\underset{\nu_n}{]} \\ &\quad \underbrace{|\partial_{\nu_n} \underset{\nu_{n-1}}{[}\, \cdots \underset{\nu_1}{[}\, y \,\underset{\nu_1}{]} \cdots \,\underset{\nu_{n-1}}{]}}_{\partial_{\nu_{n-1}} \dots \partial_{\nu_1} x \in \mathcal{E}} \underbrace{|\partial_{\nu_n}\partial_{\nu_{n-1}} \underset{\nu_{n-2}}{[}\, \cdots \underset{\nu_1}{[}\, y \,\underset{\nu_1}{]} \cdots \,\underset{\nu_{n-2}}{]}}_{\partial_{\nu_{n-2}} \dots \partial_{\nu_1} x \in \mathcal{E}} \\ &\quad | \dots \underbrace{|\partial_{\nu_n} \dots \partial_{\nu_2} \underset{\nu_1}{[}\, y \,\underset{\nu_1}{]}}_{\partial_{\nu_1} x \in \mathcal{E}} \underbrace{|\partial_{\nu_n} \dots \partial_{\nu_1} y}_{x \in \mathcal{E}}\ .\end{aligned}$$

So one of the alternative terms must exist and be nullable, due to the assumption $\partial_f(xy) \in \mathcal{E}$.

▷ **Case IV.b.1** Let $\partial_{\nu_n} \dots \partial_{\nu_1} x \underset{\nu_n}{[}\, \cdots \underset{\nu_1}{[}\, y \,\underset{\nu_1}{]} \cdots \,\underset{\nu_n}{]} \in \mathcal{E}$.
This implies, that

$$\partial_f x \in \mathcal{E} \wedge \underset{\nu_n}{[}\, \cdots \underset{\nu_1}{[}\, y \,\underset{\nu_1}{]} \cdots \,\underset{\nu_n}{]} \in \mathcal{E}$$

which implies $f \vDash x$. Choosing $g := f$ and $h := \epsilon$ yields

$$\begin{aligned}&g \vDash x \wedge h \vDash y \wedge gh = f \\ &\Rightarrow f \vDash z\ .\end{aligned}$$

▷ **Case IV.b.2** Assume, for an $i \in \mathbb{N}$ with $1 \leq i \leq n$, that

$$\partial_{\nu_n} \dots \partial_{\nu_i} \underset{\nu_{i-1}}{[}\, \cdots \underset{\nu_1}{[}\, y \,\underset{\nu_1}{]} \cdots \,\underset{\nu_{i-1}}{]} \in \mathcal{E} \tag{2}$$

and that this term exists in the term depicting $\partial_f(xy)$ above, i.e.,

$$\partial_{\nu_{i-1}} \dots \partial_{\nu_1} x \in \mathcal{E}\ . \tag{3}$$

So, we know that

$$\forall k, l \in \mathbb{N} : \big(n \geq k \geq i \wedge i - 1 \geq l \geq 1 \Rightarrow \nu_k \grave{>} \nu_l \wedge \mu_k \acute{>} \nu_l\big)$$

holds, due to the guards in (2), and that $\partial_{\nu_n} \dots \partial_{\nu_i} y \in \mathcal{E}$. So

$$\exists h \in \mathcal{F} : \varphi h \sim \{\nu_i, \dots, \nu_n\} \wedge \partial_h y \in \mathcal{E} \quad \text{because of (2)}$$
$$\exists g \in \mathcal{F} : \varphi g \sim \{\nu_1, \dots, \nu_{i-1}\} \wedge \partial_g x \in \mathcal{E} \quad \text{because of (3)} \ .$$

For those g and h, induction yields

$$g \vDash x \wedge h \vDash y$$

which implies that $f \vDash z$.

▷ **Case V** $z = x^{m,n}$.
If $f = \epsilon$, nothing needs to be shown, since then $\partial_f = \mathrm{id}$, and

$$\forall (m, n) \in \mathbf{R} : \big(\epsilon \vDash x^{m,n} \Longleftrightarrow x^{m,n} \in \mathcal{E}\big)$$

is obvious, because of Definition 3.1.4. So let us assume $\varphi f = \{\mu_1, \dots, \mu_k\} \neq \emptyset$, pre-sorted.

▷ **Case V.a** $n = 1$.
For $m \in \{0, 1\}$, we have

$$\begin{aligned} \partial_f x^{m,1} \in \mathcal{E} &\Longleftrightarrow \partial_{\mu_k} \dots \partial_{\mu_1} x^{m,1} \in \mathcal{E} \\ &\Longleftrightarrow \partial_{\mu_k} \dots \partial_{\mu_1} x \in \mathcal{E} \quad \text{due to Definition 3.2.1} \\ &\Longleftrightarrow \partial_f x \in \mathcal{E} \end{aligned}$$

which is equivalent to $f \vDash x$, due to induction.

▷ **Case V.b** $n > 1$.
We use induction over n, assuming that

$$\forall i, j \in \mathbb{N}, 1 \leq j < n, 0 \leq i \leq j : (\partial_f x^{i,j} \in \mathcal{E} \Longleftrightarrow f \vDash x^{i,j})$$

holds. Again, we apply the derivation step by step. So $\partial_{\mu_1} x^{m,n}$ yields

$$\partial_{\mu_1} x \underset{\mu_1}{[}\ x^{m',n-1}\ \underset{\mu_1}{]}$$

where, for the remainder of this case,

$$m' := \begin{cases} 0 & \text{if } m = 0 \\ m - 1 & \text{otherwise} \ . \end{cases}$$

Then, the application of ∂_{μ_2} yields

$$\partial_{\mu_2}\partial_{\mu_1} x \underset{\mu_2\mu_1}{[}\ \underset{\mu_1}{[}\ x^{m',n-1}\ \underset{\mu_1}{]}\ \underset{\mu_1\mu_2}{]}\ \underbrace{|\partial_{\mu_2} \underset{\mu_1}{[}\ x^{m',n-1}\ \underset{\mu_1}{]}}_{\partial_{\mu_1} x \in \mathcal{E}}$$

and so on, until the application of ∂_{μ_k} finally yields

$$\begin{aligned}\partial_f x^{m,n} \;=\; & \partial_{\mu_k}\ldots\partial_{\mu_1} x \underset{\mu_k}{[}\cdots\underset{\mu_1}{[}x^{m',n-1}\underset{\mu_1}{]}\cdots\underset{\mu_k}{]} \underbrace{|\partial_{\mu_k}\underset{\mu_{k-1}}{[}\cdots\underset{\mu_1}{[}x^{m',n-1}\underset{\mu_1}{]}\cdots\underset{\mu_{k-1}}{]}}_{\partial_{\mu_{k-1}}\ldots\partial_{\mu_1}x\in\mathcal{E}} \ldots \\ & \ldots \underbrace{|\partial_{\mu_k}\ldots\partial_{\mu_{j+1}}\underset{\mu_j}{[}\cdots\underset{\mu_1}{[}x^{m',n-1}\underset{\mu_1}{]}\cdots\underset{\mu_j}{]}}_{\partial_{\mu_j}\ldots\partial_{\mu_1}x\in\mathcal{E}}\ldots \\ & \ldots \underbrace{|\partial_{\mu_k}\ldots\partial_{\mu_2}\underset{\mu_1}{[}x^{m',n-1}\underset{\mu_1}{]}}_{\partial_{\mu_1}x\in\mathcal{E}} \quad .\end{aligned} \tag{4}$$

Now let us have a look at the both directions of the theorem. For the following two steps (Part V.b.1 and Part V.b.2) first assume that $m = 0$. After that, using exactly the same reasoning for $m > 0$ leads to an inductive proof for all $m \in \mathbb{N}, m \leq n$.

▷ **Part V.b.1** "$\Longrightarrow$".
Assume $f \vDash x^{m,n}$. This means that $\exists i \in \mathbb{N},\ m \leq i \leq n : f \vDash x^i$. Since we have agreed on $f \neq \epsilon$, we can assume $1 \leq i$, which is relevant only if $m = 0$. We already know that

$$\begin{aligned}& f \vDash x^i \Longleftrightarrow f \vDash x(x^{i-1}) \Longleftrightarrow \exists g, h \in \mathcal{F}, f = gh : g \vDash x \wedge h \vDash x^{i-1} \\ & f = gh \Rightarrow \exists j \in \mathbb{N}, 1 \leq j \leq k : \varphi g \sim \{\mu_1, \ldots, \mu_j\} \wedge \varphi h \sim \{\mu_{j+1}, \ldots, \mu_k\} \ .\end{aligned}$$

Assuming such g and h, this j indicates which of the alternative terms in (4) to look at. If $j = k$, the term is nullable. This can be seen as follows:

$$\begin{array}{rcll} j = k & \Rightarrow & h = \epsilon & \text{because } \varphi g \sim \{\mu_1, \ldots, \mu_k\} \Rightarrow \varphi h = \emptyset \\ & \Rightarrow & \epsilon \vDash x^{i-1} & \text{because of } h \vDash x^{i-1} \\ & \Rightarrow & x \in \mathcal{E} \vee i = 1 & \\ & \Rightarrow & x \in \mathcal{E} \vee m \leq 1 & \text{because of } m \leq i \\ & \Rightarrow & x^{m',n-1} \in \mathcal{E} & \text{because } m \leq 1 \Rightarrow m' = 0 \ . \end{array}$$

And with $g \vDash x$ we have $\partial_f x \in \mathcal{E}$ due to induction over the structure of z, hence both parts of the first term are nullable.
Otherwise, i.e., if $j < k$, we have $\partial_{\mu_j}\ldots\partial_{\mu_1} x \in \mathcal{E}$, since $g \vDash x$. So the according term exists (it is the one in the middle line of (4)). Since $f = gh$, the constraints imposed by the guards in that term are all fulfilled:

$$\forall \alpha \in \{\mu_1, \ldots, \mu_j\} : \forall \beta \in \{\mu_{j+1}, \ldots, \mu_k\} : \alpha \dot{\leq} \beta \wedge \alpha \dot{<} \beta \ .$$

And because of $h \vDash x^{i-1}$ and $i \leq n$ we get $\partial_{\mu_k}\ldots\partial_{\mu_{j+1}} x^{m',n-1} \in \mathcal{E}$ by induction over n. So we have $\partial_f x^{m,n} \in \mathcal{E}$.

▷ **Part V.b.2** "$\Longleftarrow$".
Assume $\partial_f x^{m,n} \in \mathcal{E}$. So at least one of the alternative terms in (4) must be nullable. This situation can be handled as in Case IV.b, which leads to $f \vDash x^{m,n}$.

▷ **Case VI** $z = a\langle x\rangle$.
We prove both directions of the theorem individually:

▷ **Part VI.a** "$\Longrightarrow$".
Assume $f \vDash z$. Let $\alpha := \tau_{\varphi f}$, and $N := (\varphi f)_<$, so that $\varphi f = N \dot\cup \{\alpha\}$, and therefore $\partial_f = \partial_N \circ \partial_\alpha$. We have

$$\exists g \in \mathcal{F} : f = a\langle g\rangle \wedge g \vDash x$$
$$\partial_\alpha z = \underset{\alpha}{[}\overset{\alpha}{x]} \quad \text{since } a = \breve\alpha \ .$$

Then, for such g, we have $\partial_N x \in \mathcal{E}$ since $N \sim \varphi g$ and $\forall \nu \in N : \nu \dot< \alpha$. Together, they yield

$$\partial_f z = \underset{\alpha}{[}\overset{\alpha}{\partial_N x]} \in \mathcal{E} \ .$$

▷ **Part VI.b** "$\Longleftarrow$".
Now let $\partial_f z \in \mathcal{E}$. Indirect, by assuming that $f \nvDash z$, we have to show that then $\partial_f z \notin \mathcal{E}$. Let $f = b\langle g\rangle h,\ b \in V,\ g, h \in \mathcal{F}$. So

$$f \nvDash z \Rightarrow a \neq b \vee g \nvDash x \vee h \nvDash \epsilon \ .$$

▷ **Case VI.b.1** $a \neq b$.
Let $\beta := \tau_{\varphi f}, N = \varphi f \setminus \beta$. Then

$$\begin{aligned}\partial_f z &= \partial_N \partial_\beta a\langle x\rangle \\ &= \partial_N \oslash \quad \text{since } \breve\beta = b \neq a \\ &= \oslash \notin \mathcal{E} \ .\end{aligned}$$

▷ **Case VI.b.2** $a = b \wedge h \neq \epsilon$.
In this case, at least one non-empty tree is following the tree which has b as its root. Again, let $M \sim \varphi g, N \sim \varphi h, \alpha := \tau_{\varphi f}$.

$$\begin{aligned}\partial_f z &= \partial_N \partial_M \partial_\alpha a\langle x\rangle \\ &= \partial_N \partial_M \overset{\alpha}{[x]} && \text{since } a = b \\ &= \partial_N \underset{\alpha}{[}\overset{\alpha}{\partial_M x]} \\ &= \underset{\alpha}{\oslash} && \text{since } \forall \nu \in N : \nu \dot> \alpha \\ &\notin \mathcal{E} \ .\end{aligned}$$

▷ **Case VI.b.3** $a = b \wedge h = \epsilon \wedge g \nvDash x$.
We know $\partial_g x \notin \mathcal{E}$ due to induction. Let $M \sim \varphi g, N \sim \varphi h, \alpha := \tau_{\varphi f}$.

$$\begin{aligned}\partial_f z &= \partial_M \partial_\alpha a\langle x\rangle && \text{since } \partial_N = \mathrm{id} \\ &= \partial_M \overset{\alpha}{[x]} && \text{since } a = b \\ &= \underset{\alpha}{[}\overset{\alpha}{\partial_M x]} \\ &\notin \underset{\alpha}{\mathcal{E}} && \text{since } g \nvDash a\end{aligned}$$

The case of $z = \mathfrak{t} \in T$ is not handled here, since it can be explained by derivation of the expression defining $\mathfrak{t}$. Here, Appendix A.1 comes in handy, insofar as it assures that lookup will not run into an infinite loop chasing nonterminal definitions. □

3.4 Optimizing

Nothing As stated in Remark 2.1.18, we can remove any expression that is concatenated to $\oslash$ and any alternative branch that contains only $\oslash$. For example, if we run into $\oslash xy|z$ we can save memory and processing time by first purging xy, and then removing $\oslash|$, leaving only z alone.

Collapse Guards We can safely strip away guards that surround ϵ or $\oslash$, since the nullable test ignores them, and application of ∂ yields $\oslash$ in any case.

Also, consider the derivation of an expression that is directly surrounded by multiple guards, e.g.,

$$\begin{aligned}
\partial_\alpha \underset{ac}{\overset{bd}{[}}\underset{eg}{\overset{fh}{[}} x \underset{eg}{\overset{fh}{]}}\underset{ac}{\overset{bd}{]}} &= \begin{cases} \underset{a}{\overset{b}{[}}\partial_\alpha \underset{c}{\overset{d}{[}} x \underset{eg}{\overset{fh}{]}}\underset{a}{\overset{b}{]}} & \text{if } a < \grave{\alpha} < b \wedge g < \acute{\alpha} < h \\ \oslash & \text{otherwise} \end{cases} \\
&= \begin{cases} \underset{ac}{\overset{bd}{[}}[\partial_\alpha x] \underset{eg}{\overset{fh}{]}}\underset{ac}{\overset{bd}{]}} & \text{if } a < \grave{\alpha} < b \wedge g < \acute{\alpha} < h \wedge c < \grave{\alpha} < d \wedge e < \acute{\alpha} < f \\ \oslash & \text{otherwise} \end{cases} \\
&= \underset{\max\{a,c\}}{\overset{\min\{b,d\}}{[}} x \underset{\max\{e,g\}}{\overset{\min\{f,h\}}{]}} .
\end{aligned}$$

So instead of introducing new guards we can modify guards that already exist. This saves memory and speeds up further processing.

Only post order constraints Proof 3.3.4 shows that the upper guard limit of pre values is never used. This is evident, since the only constraints used are *following* and *descendant*. None of them introduces an upper limit on the pre values.

Moreover, the lower limit on pre values will always be satisfied, since the enumerated nodes are processed with strictly increasing pre values (Definition 3.2.3).

So it is safe to ignore all the pre value limits given in guards.

Other enumerations This algorithm can be applied to any enumeration that allows to determine the structural relationships (ancestor, following) between the nodes.

Without proof, e.g., the pre/size enumeration — annotating each node with its pre value and the size of its child forest — offers the properties required by this

algorithm and additionally gives the opportunity to easily skip parts of the document that shall not be validated. This might be desired, if parts (i.e., subtrees) of the document have been validated already (e.g., if they own a `xsi:type` attribute which explicitly sets their type), or when using the XQuery validation modes `lax` or `skip`.

Further research Maybe there is an efficient way to replace guarded expressions by looking at their pre/post values (or whatever enumeration is used). A very simple example is an expression guarded with consecutive values, which always yields $\oslash$ when derived, and thus could be safely replaced by ϵ, if it is nullable, or $\oslash$ otherwise:

$$\partial_\alpha[x\,]_p^{p+1} = \oslash \quad \text{since the guards impose exclusive limits} \ .$$

Regarding the structure of the grammar or the forest already processed, it may be possible to gather information about the pre/post values that shall be assigned to the remaining nodes of a valid document, or about the size a certain subtree must have. This could yield information that allows detection of expressions whose guards will never be satisfied. Maybe other enumerations offer better opportunities for such an optimization.

3.5 Summary

Now we have closed the gap between the relational forest encoding and the context free grammars restricting them to a certain shape.

The regular expressions have been enriched by guards to supply the derivation process with information about where in the tree a node is situated, relative to others. The derivation was introduced, and it turned out that a forest f matches a regular expression x if, and only if, derivation of that expression according to the enumerated forest yields a nullable expression:

$$f \vDash x \iff \partial_{\varphi f} x \in \mathcal{E} \ .$$

Some approaches for optimization have been proposed in the remainder of the chapter.

Chapter 4

XML Schema Type Information

4.1 Fitting XML

Most of the XML Schema constructs, namely those that can be expressed by context free grammars, can be translated straightforward into the calculus developed in this thesis: *complex types* are constructed using the $a\langle b\rangle$ construct, *named complex types* and *named groups* are mapped to nonterminals. *Choice* and *sequence* are translated straightforward into the $\cdot$ and $|$ operators, *occurrence constraints* are translated into the $x^{m,n}$ construct. The *All group*, which provides a simplified version of the SGML & operator, is handled below. With this, we can handle *attributes* as children of the node they belong to. Their order does not play a role in XML.

The All Group It is obvious that an All group can be translated into an expression built by means of sequence and choice operators only. However, the size requirements of such an expression, which describes an All group with k elements, is in $\mathcal{O}(k!)$. For example,

$$\{a, b, c\} \equiv abc|acb|bac|bca|cab|cba$$

where the set notation with curly braces { and } is used to express the All group.

But fortunately, since derivation is applied to the All group, we are not interested in the whole resulting expression. It is only relevant how the All group changes when derived. If we implement derivation of the All group by

$$\partial_{p,a,q} : \{a_1, a_2, \ldots, a_k\} \longmapsto \partial_{p,a,q} \begin{array}{ll} (& a_1\{a_2, \ldots, a_k\} \\ | & a_2\{a_1, a_3, \ldots, a_k\} \\ \vdots & \\ | & a_k\{a_1, \ldots, a_{k-1}\} \quad) \end{array}$$

we only need space in $\mathcal{O}(k^2)$, since after the following derivation step all but (at most) one of the alternative terms will disappear, due to the strong limitations

the XML Schema standard imposes on the All group (see Appendix A.3 and Appendix A.4). Time complexity will remain in $\mathcal{O}(k^2)$, however.

The mentioned restrictions on the All group also save us an extra discussion of ambiguity for such "unfolded" All groups.

Implementation using sets might be even faster (e.g., using bit vectors): Understand the All group as a set, and remove its elements one by one during derivation. Derivation according to an element that is not in the set yields $\oslash$, the empty set is nullable, as well as a set that contains only elements with a minimum occurrence constraint of zero. This way, the time and space requirements depend on the implementation of the set only. However, the digression from the calculus of purely regular expressions imposes a lack of elegance.

4.2 Purchase Order Schema Example

As an example of how to translate an XML Schema description into this calculus, recall the *Purchase Order* example taken from [15][1], with minor modifications to circumvent XML Schema features that are currently not implemented by the algorithm developed in this thesis.

Again, XML and XML Schema code snippets are marked by a vertical line on their left hand side.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:choice>
        <xsd:group   ref="shipAndBill"/>
        <xsd:element name="singleUSAddress" type="USAddress"/>
      </xsd:choice>
      <xsd:element name="comment" type="xsd:string"
                                      minOccurs="0"/>
      <xsd:element name="items"  type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:group name="shipAndBill">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
    </xsd:sequence>
  </xsd:group>
```

[1] In [15], Section 2.1 gives the basic example, and Section 2.7 introduces named groups.

```
  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name"   type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city"   type="xsd:string"/>
      <xsd:element name="state"  type="xsd:string"/>
      <xsd:element name="zip"    type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN"
                                        fixed="US"/>
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0"
        maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName"
                          type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="USPrice"  type="xsd:decimal"/>
            <xsd:element name="comment" type="xsd:string"
                                           minOccurs="0"/>
            <xsd:element name="shipDate" type="xsd:date"
                                           minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="partNum" type="SKU"
                                      use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

Translating the Schema Let us construct the grammar (START, λ) implementing the Purchase Order schema. For readability, mnemonic names like ITEMS for nonterminals, and *zip* for terminals are used, instead of symbols like $\mathfrak{t}$ and a.

Note that the three XML Schema concepts

- named complex type (<xsd:element name="a" type="$\mathfrak{t}$"/>),
- named group (<xsd:group ref="$\mathfrak{t}$"/>), and
- anonymous complex type (<xsd:complexType name="$\mathfrak{t}$">...)

are mapped to only two concepts in this calculus, namely *nonterminals* and *subtree construction*. The only difference between named complex types and named groups is that the former are associated with a node and yield type information — while the latter stand on their own and do not yield type information. In the implementation, this distinction is realized by two different namespaces the names reside in (see *The Regular Expressions* on page 69). Here, we use the suggestive notation

$$\textit{element}{::}\text{NAME} \quad := \quad \textit{element}\langle\text{NAME}\rangle$$

iff NAME refers to a named type. This denotes that resolving the name yields type information for the associated node. The set of nonterminals thus is

$$\begin{aligned} T := \{ \;& \text{START}, \text{ITEMS}, \text{PURCHASEORDERTYPE}, \text{SKU}, \text{USADDRESS}, \\ & \text{SHIPANDBILL}, \text{XSD:NMTOKEN1}, \text{XSD:DATE}, \text{XSD:DECIMAL}, \\ & \text{XSD:POSITIVEINTEGER}, \text{XSD:STRING} \qquad \} \; . \end{aligned}$$

Since attributes are handled as children of the respective nodes, they are marked with an @ to indicate their kind. The set of terminals thus is

$$\begin{aligned} V := \{ \;& \textit{@country}, \textit{@orderDate}, \textit{@partNum}, \textit{USPrice}, \textit{billTo}, \textit{city}, \\ & \textit{comment}, \textit{items}, \textit{item}, \textit{name}, \textit{productName}, \textit{purchaseOrder}, \\ & \textit{quantity}, \textit{shipDate}, \textit{shipTo}, \textit{singleUSAddress}, \textit{state}, \textit{street}, \textit{zip}, \\ & /.*/, /[0-9][0-9]*/, /[0-9][0-9][0-9]-[A-Z][A-Z]/ \quad \} \end{aligned}$$

where each of the regular expressions given at the end of V is meant to extend the set by a set of valid text nodes in the XML document. Clearly, this is not a technical clean solution, however, it permits deferring validation of text nodes to later work. See also *Text Nodes* on page 40.

The regular expressions for strings, as defined for XML Schema's pattern facet in [17] in Appendix F, are embraced by a pair of slashes (/), a well known notation borrowed from Perl [11].

Lookup Function In addition to the sets of (non)terminals, the lookup function must be defined. Processing of the document starts with derivation of the START group:

$$\begin{aligned} \lambda(\text{START}) \quad := \quad & \textit{purchaseOrder}{::}\text{PURCHASEORDERTYPE} \\ \mid \quad & \textit{comment}{::}\text{XSD:STRING} \; . \end{aligned}$$

The definition of a type like

$$\begin{array}{rcl}\lambda(\text{PURCHASEORDERTYPE}) & := & (@\mathit{orderDate}::\text{XSD:DATE})^{0,1} \\ & \cdot & (\\ & & \quad \text{SHIPANDBILL} \\ & & \quad \mid \mathit{singleUSAddress}::\text{USADDRESS} \\ & &) \\ & \cdot & (\mathit{comment}::\text{XSD:STRING})^{0,1} \\ & \cdot & \mathit{items}::\text{ITEMS}\end{array}$$

looks identical to the definition of a group like

$$\lambda(\text{SHIPANDBILL}) \;:=\; \mathit{shipTo}::\text{USADDRESS} \cdot \mathit{billTo}::\text{USADDRESS}\ .$$

The difference between named groups and types — different namespaces — is not visible here.

$$\begin{array}{rcl}\lambda(\text{USADDRESS}) & := & @\mathit{country}::\text{XSD:NMTOKEN1} \cdot \mathit{name}::\text{XSD:STRING} \\ & & \cdot\, \mathit{street}::\text{XSD:STRING} \cdot \mathit{city}::\text{XSD:STRING} \\ & & \cdot\, \mathit{state}::\text{XSD:STRING} \cdot \mathit{zip}::\text{XSD:DECIMAL} \\ \lambda(\text{XSD:NMTOKEN1}) & := & /US/\end{array}$$

Note that *country* was prefixed with an @ sign to indicate its origin as an attribute in the XML document. The position of the attribute at the beginning of the sequence conforms with the requirements of the XQuery document order ([18] in Section 2.4). Also note that a new type XSD:NMTOKEN1 has been introduced to represent the restriction of the XSD:NMTOKEN node to a certain value.

$$\begin{array}{rcl}\lambda(\text{ITEMS}) & := & \big(\mathit{item}\langle \\ & & \quad @\mathit{partNum}::\text{SKU} \cdot \mathit{productName}::\text{XSD:STRING} \\ & & \quad \cdot\, \mathit{quantity}::\text{XSD:POSITIVEINTEGER} \cdot \mathit{USPrice}::\text{XSD:DECIMAL} \\ & & \quad \cdot\, (\mathit{comment}::\text{XSD:STRING})^{0,1} \cdot (\mathit{shipDate}::\text{XSD:DATE})^{0,1} \\ & & \langle\big)^{0,\infty}\end{array}$$

The definition of *item*'s content model is "inlined" here, since *item* is of anonymous type.

$$\lambda(\text{SKU}) \;:=\; /\backslash d\{3\} - [A-Z]\{2\}/$$

Here, the use of plain regular expressions that do not express trees — as known from XML's pattern facet — is shown, assuming that they define a choice of valid terminals, like "000-AA|000-AB| . . . |999-ZZ" in the example above.

$$\begin{array}{rcl}\lambda(\text{XSD:STRING}) & := & /.*/ \\ \lambda(\text{XSD:DECIMAL}) & := & /[0-9][0-9]*/\end{array}$$

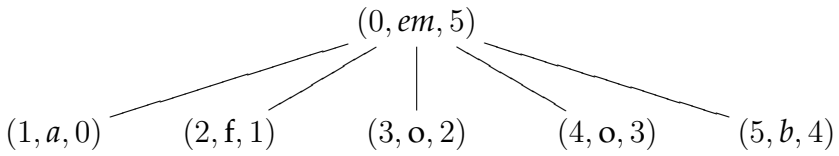
These are two examples of how the predefined types XSD:STRING and XSD:DECIMAL might be implemented using this calculus.

Text Nodes To validate text nodes constrained by XML Schema's pattern facet, an obvious approach would be to extend the process of derivation to regular expressions for strings. Therefore, consider a string of characters being a forest, each character forming one node. Note that a string is flat, i.e., there is no subtree construction. The set of terminals must be extended to contain the Unicode character set [12]. In fact, it will then be a union of nodes induced by XML element nodes, and the character set.

Following this idea, which is also suggested in [14] in Section 2.6, the XML fragment

```
<em><a/>foo<b/></em>
```

would be understood and encoded as

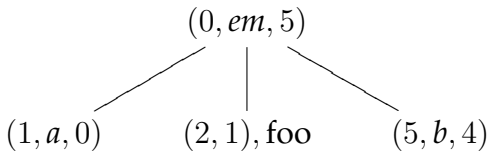


clearly validating against a regular expression like

$$em\langle a \cdot \mathrm{f} \cdot \mathrm{o} \cdot \mathrm{o} \cdot b \rangle \ .$$

One drawback is the memory consumption, which is roughly trebled for documents containing mainly unstructured textual data. However, it is evident that a string has strictly sequential structure. According to Definition 2.5.1, we know that all characters inside the same string will have strictly sequential pre/post values assigned (looking at that definition, one finds that $x = q - 1$ and $y = p$, since $f = \epsilon$). So during encoding (i.e., enumerating), it is possible to annotate the string with the pre/post values for its first character only, and increase the pre/post counters according to the string's length.

The above example would be encoded as



with the pair $(2, 1)$ denoting the start values for enumerating the string, which is then done during validation. Note that the concept of the pre/post plane is not violated by this change in the enumeration.

Though regular expressions might offer an easy approach to cover some of XML Schema's facets ([15] in Appendix B), it seems unwise to use this mechanism, e.g., for a *maxInclusive* facet, restricting a value of type *dateTime*. An elegant solution for validating all possible facets is deferred to further research.

4.3 Collecting Type Information

The purpose of validation is not only to determine whether a given document is valid. We also want to *annotate each node with its type*. This means collecting type names $\mathfrak{t}$ whenever the derivation triggers a lookup $\lambda\mathfrak{t}$ that does not belong to a named group (XML Schema Groups do not generate type information. Consider them being a "macro facility" that allows reuse of code).

Potentially, we must save the type information of each name resolved during derivation, whether it will be part of the final type information, or not. Consider the (not XML conforming) grammar $(\mathfrak{s}, \lambda)$ with

$$\begin{aligned} V &:= \{a, b, c\} & T &:= \{\mathfrak{s}, \mathfrak{t}_1, \mathfrak{t}_2\} \\ \lambda\mathfrak{s} &:= a\langle\mathfrak{t}_1\rangle | a\langle\mathfrak{t}_2\rangle \\ \lambda\mathfrak{t}_1 &:= b & \lambda\mathfrak{t}_2 &:= c \end{aligned}$$

implementing the intention "the document consists of a node a which is either of type $\mathfrak{t}_1$ and contains a b node, or of type $\mathfrak{t}_2$ and contains a c node".

Now there are two forests matching this grammar: $a\langle b\rangle$ and $a\langle c\rangle$. Derivation of both forests succeeds, yielding different type annotations. We must annotate each alternative branch of the produced $\mathfrak{X}$ expression with the possible types the node could have:

$$\partial_{0,a,1}\mathfrak{s} = \partial_{0,a,1}(a\langle\mathfrak{t}_1\rangle | a\langle\mathfrak{t}_2\rangle) = \partial_{0,a,1}a\langle\mathfrak{t}_1\rangle | \partial_{0,a,1}a\langle\mathfrak{t}_2\rangle = \underbrace{\overset{1}{[\mathfrak{t}_1]}}_{\substack{0\\ a::\mathfrak{t}_1}} \mid \underbrace{\overset{1}{[\mathfrak{t}_2]}}_{\substack{0\\ a::\mathfrak{t}_2}} .$$

After successful derivation according to the whole enumeration, one of the alternative branches makes the expression nullable:

$$\begin{aligned} \partial_{1,b,0}(\underbrace{\overset{1}{[\mathfrak{t}_1]}}_{\substack{0\\ a::\mathfrak{t}_1}} \mid \underbrace{\overset{1}{[\mathfrak{t}_2]}}_{\substack{0\\ a::\mathfrak{t}_2}}) &= \ldots = \underbrace{\overset{1}{[\partial_{1,b,0}b]}}_{\substack{0\\ a::\mathfrak{t}_1}} \mid \underbrace{\overset{1}{[\partial_{1,b,0}c]}}_{\substack{0\\ a::\mathfrak{t}_2}} = \underbrace{\overset{1}{[\epsilon]}}_{\substack{0\\ a::\mathfrak{t}_1}} \mid \underbrace{\overset{1}{[\oslash]}}_{\substack{0\\ a::\mathfrak{t}_2}} \\ \partial_{1,c,0}(\underbrace{\overset{1}{[\mathfrak{t}_1]}}_{\substack{0\\ a::\mathfrak{t}_1}} \mid \underbrace{\overset{1}{[\mathfrak{t}_2]}}_{\substack{0\\ a::\mathfrak{t}_2}}) &= \ldots = \underbrace{\overset{1}{[\partial_{1,c,0}b]}}_{\substack{0\\ a::\mathfrak{t}_1}} \mid \underbrace{\overset{1}{[\partial_{1,c,0}c]}}_{\substack{0\\ a::\mathfrak{t}_2}} = \underbrace{\overset{1}{[\oslash]}}_{\substack{0\\ a::\mathfrak{t}_1}} \mid \underbrace{\overset{1}{[\epsilon]}}_{\substack{0\\ a::\mathfrak{t}_2}} . \end{aligned}$$

Type annotation associated with the nullable branch is used, while annotation associated with non-nullable expressions is discarded. This way we would have to collect the correct type annotations during the final nullable test.

Polymorphism The potential concomitance of multiple nullable alternative branches would even lead to documents of polymorphic type. Consider the following (not valid, see below) XML Schema description:

```
<xsd:element name="a">
  <xsd:complexType>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="b" type="xsd:boolean">
      <xsd:element name="b" type="xsd:integer">
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

This is translated into

$$
\begin{aligned}
V &:= \{a, b, \mathit{true}, \mathit{false}, /[0-9][0-9]*/\} \\
T &:= \{\text{XSD:BOOLEAN}, \text{XSD:INTEGER}\} \\
\lambda(\text{XSD:BOOLEAN}) &:= \mathit{true}|\mathit{false}|\mathit{1}|\mathit{0} \\
\lambda(\text{XSD:INTEGER}) &:= /[0-9][0-9]*/ \ .
\end{aligned}
$$

With this, node b has a polymorphic type. For example, consider the following XML document:

```
<a>
  <b>23</b>
  <b>false</b>
  <b>1</b>
</a>
```

Clearly, the first and second occurrence of b would have assigned XSD:INTEGER and XSD:BOOLEAN. The third occurrence, however, might validate as XSD:INTEGER as well as as XSD:BOOLEAN.

Though multiple views of a document might look eligible, polymorphism introduces difficulties during processing such documents, as discussed in Chapter 5. Therefore, the XML Schema standard employs the Unique Particle Attribution Constraint (see Appendix A.2). This restriction, also known as one-unambiguity, enforces grammars to be of a form that always allows to determine the "correct" alternative branch immediately, i.e., without further inspection of the input stream. So the grammars given above do not represent a valid XML Schema description.

Thus, we do not need to store the type annotations inside the $\mathfrak{X}$ expression and invest the effort of collecting them later. Instead, it is possible to directly store the mapping of a node to its type into a simple lookup table, since the node either has one certain type which is known immediately, or is not part of a valid document. If the final nullable test succeeds, this type annotation table is declared valid, otherwise it is discarded, because the document is not valid.

Forecast In Chapter 5 it will turn out that one-unambiguity (which is formalized there) may be lost during the derivation process (see the example on page 54). However, this will not introduce polymorphism, as explained in Section 5.4.

4.4 A Complete Validation Example

As a more comprehensive example of XML Schema validation than the one given above, have a look at the Purchase Order, taken from [15] in Section 2. The according schema description was already discussed and translated into our calculus in Section 4.2.

The accompanying software contains the file `purchaseorder.hs`, source code of a Purchase Order schema validator which implements the according schema description. There you will also find the file `purchaseorder.xml`, which contains an XML document validating against the schema:

```
<?xml version="1.0"?>

<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

Compilation of `purchaseorder.hs` and the associated source code files produces the `purchaseorder` schema validator. All examples in this chapter concerning the Purchase Order example, can be reproduced by this software. See

pre	node	post	pre	node	post
0	*purchaseOrder*	53	27	*zip*	26
1	*@orderDate*	1	28	"95819"	25
2	"1999-10-20"	0	29	*comment*	29
3	*shipTo*	14	30	"Hurry, my lawn is going wild!"	28
4	*@country*	3	31	*items*	52
5	"US"	2	32	*item*	40
6	*name*	5	33	*@partNum*	31
7	"Alice Smith"	4	34	"872-AA"	30
8	*street*	7	35	*productName*	33
9	"123 Maple Street"	6	36	"Lawnmower"	32
10	*city*	9	37	*quantity*	35
11	"Mill Valley"	8	38	"1"	34
12	*state*	11	39	*USPrice*	37
13	"CA"	10	40	"148.95"	36
14	*zip*	13	41	*comment*	39
15	"90952"	12	42	"Confirm this is electric"	38
16	*billTo*	27	43	*item*	51
17	*@country*	16	44	*@partNum*	42
18	"US"	15	45	"926-AA"	41
19	*name*	18	46	*productName*	44
20	"Robert Smith"	17	47	"Baby Monitor"	43
21	*street*	20	48	*quantity*	46
22	"8 Oak Avenue"	19	49	"1"	45
23	*city*	22	50	*USPrice*	48
24	"Old Town"	21	51	"39.98"	47
25	*state*	24	52	*shipDate*	50
26	"PA"	23	53	"1999-05-21"	49

Table I: Encoding of `purchaseorder.xml`

Appendix B for more details about the supplied software.

A complete enumeration of the Purchase Order document tree is given in Table I. Note that the pre column is strictly ordered, hence can be omitted in a RDBMS that preserves the order of the stored entries.

A closeup view Let us have a look at some derivation steps performed during validation.

The first step is derivation of

$$(\textit{purchaseOrder}::\textsc{PurchaseOrderType}|\textit{comment}::\textsc{xsd}:\textsc{String})$$

according to the first node $(0, \textit{purchaseOrder}, 53)$. This yields, after performing a

lookup $\lambda(\text{PURCHASEORDERTYPE})$, the expression

$$[(@\textit{orderDate}::\text{XSD:DATE}^{0,1} \cdot (\text{SHIPANDBILL}|\textit{singleUSAddress}::\text{USADDRESS}) \cdot$$

$$\textit{comment}::\text{XSD:STRING})^{0,1} \cdot \textit{items}::\text{ITEMS}\,{}^{53}]$$

which is not nullable. It states, that the remainder of the document must consist of

- an optional *orderDate* attribute of type XSD:DATE,
- followed by
  - either a SHIPANDBILL group, or
  - a *singleUSAddress* element of type USADDRESS,
- followed by an optional *comment* element of type XSD:STRING, and finally
- followed by an *items* node of type ITEMS.

Moreover, all these elements must have a post order value smaller than 53 associated, which marks them being descendants of the $(0, \textit{purchaseOrder}, 53)$ node. Recall that the pre order constraints can be ignored, as discussed in *Only post order constraints* on page 32.

The lookup $\lambda(\text{PURCHASEORDERTYPE})$ performed above, also yields type information since the nonterminal PURCHASEORDERTYPE is associated to a type definition (in contrast to a group definition, which does not generate type information). Hence, the table implementing the pre $\mapsto$ type mapping (Table II on page 48) is populated with a $(0, \text{PurchaseOrderType})$ tuple.

For brevity, only one more derivation step will be discussed:

Just before derivation according to $(41, \textit{comment}, 39)$, the regular expression describing the potential remainder of the document is nullable:

$$\begin{array}{l}
[\quad[\qquad[\epsilon\,{}^{37}_{35}] \\
\qquad\qquad[(\textit{comment}::\text{XSD:STRING})^{0,1}\,{}_{37}] \\
\qquad\qquad[(\textit{shipDate}::\text{XSD:DATE})^{0,1}\,{}_{37}] \\
\quad{}^{40}] \\
\quad[(\textit{item}\langle \\
\qquad\qquad @\textit{partNum}::\text{SKU} \cdot \textit{productName}::\text{XSD:STRING} \cdot \\
\qquad\qquad \textit{quantity}::\text{XSD:POSITIVEINTEGER}\textit{USPrice}::\text{XSD:DECIMAL} \cdot \\
\qquad\qquad (\textit{comment}::\text{XSD:STRING})^{0,1} \cdot (\textit{shipDate}::\text{XSD:DATE})^{0,1} \\
\quad\rangle)^{0,\infty}\,{}_{40}] \\
{}^{52}] \quad . \\
{}_{29}
\end{array}$$

This is, because a document that is equivalent to the subtree of the Purchase Order example processed so far (i.e., the first 41 nodes), also validates against the Purchase Order schema. As one can see from the regular expression depicted above, the remaining two elements *comment* and *shipDate*, which might appear as children of the $(32, \textit{item}, 40)$ node, succeeding the $(39, \textit{USPrice}, 37)$ node, are both optional due to their occurrence constraint of ^{0,1}. Also, *item*s following $(32, \textit{item}, 40)$ are not required due to their occurrence constraint of $^{0,\infty}$. The first ϵ expression is a relict of previous successful derivations.

However, immediately after derivation according to the now following node $(41, \textit{comment}, 39)$, the regular expression is not nullable any more, since it requires a string as child of the *comment* node:

$$
\begin{array}{l}
[\quad[\qquad(\quad\oslash[(\textit{comment}{::}\text{XSD:STRING})^{0,1}\,]^{39}_{39}\quad|\quad[/.^{0,\infty}/\,]^{39}_{37}\quad) \\
\qquad\qquad[(\textit{shipDate}{::}\text{XSD:DATE})^{0,1}\,]^{39}_{39}\,|\oslash \\
\quad]^{40} \\
\quad[(\textit{item}\langle \\
\qquad\qquad @\textit{partNum}{::}\text{SKU}\cdot \textit{productName}{::}\text{XSD:STRING}\cdot \\
\qquad\qquad \textit{quantity}{::}\text{XSD:POSITIVEINTEGER}\textit{USPrice}{::}\text{XSD:DECIMAL}\cdot \\
\qquad\qquad (\textit{comment}{::}\text{XSD:STRING})^{0,1}\cdot(\textit{shipDate}{::}\text{XSD:DATE})^{0,1} \\
\quad\rangle)^{0,\infty}\,]^{40} \\
]^{52}_{29}\quad.
\end{array}
$$

Note that the *comment* is still in the expression, a relict from being preceded by the $[\epsilon]^{37}_{35}$ expression above. It does does not play a role in further derivations, since it is concatenated to a $\oslash$, hence can be removed by an optimizer (see *Nothing* on page 32). The relevant part is the expression $[/.^{0,\infty}/\,]^{39}_{37}$, which will match a string that is enumerated exactly with post value 38.

Also note that the guards protecting the *shipDate* expression have changed from $q > 37$ to $q > 39$ denoting "following $(41, \textit{comment}, 39)$", but remain $q < 40$, still denoting "child of $(32, \textit{item}, 40)$".

Table II on page 48, finally, displays the type annotations collected for the Purchase Order document. Note that the node names are displayed only for convenience. It is sufficient to have a pre number $\mapsto$ type annotation mapping.

Also note that there are two minor differences between type annotation as suggested by the XML Schema standard and as produced by the current implementation of the proposed algorithm.

- Some nodes (namely the *item*s in the example) do not generate type annotations. This is, because they are of anonymous type, as can be seen in the

definition of the grammar. In XML Schema parlance, their type annotation is “xs:anyType”.

This behavior is not implemented, since the suggested annotation is rather a convention than being derived from the supplied schema or document.

- The annotation of text nodes with the matched regular expression, embraced by slashes /.../, is used in this thesis only. The XML Schema specification does not assign a type to text nodes. XQuery suggests “xsd:untypedAtomic”.

  Here, we decided to annotate text nodes with the regular expression, which is associated with the least loss of information.

Of course, an implementation not only used for testing must fulfill the standards. However, the mentioned shortcomings are subject to only trivial changes in the implementation, and — for development purposes — well justified.

pre	node	type annotation
0	*purchaseOrder*	PurchaseOrderType
1	*@orderDate*	xsd:Date
2	1999-10-20	$/.... - .. - ../$
3	*shipTo*	USAddress
4	*@country*	xsd:Nmtoken1
5	US	$/US/$
6	*name*	xsd:String
7	Alice Smith	$/.^{0,\infty}/$
8	*street*	xsd:String
9	123 Maple Street	$/.^{0,\infty}/$
10	*city*	xsd:String
11	Mill Valley	$/.^{0,\infty}/$
12	*state*	xsd:String
13	CA	$/.^{0,\infty}/$
14	*zip*	xsd:Decimal
15	90952	$/[0-9]^{1,\infty}/$
16	*billTo*	USAddress
17	*@country*	xsd:Nmtoken1
18	US	$/US/$
19	*name*	xsd:String
20	Robert Smith	$/.^{0,\infty}/$
21	*street*	xsd:String
22	8 Oak Avenue	$/.^{0,\infty}/$
23	*city*	xsd:String
24	Old Town	$/.^{0,\infty}/$
25	*state*	xsd:String
26	PA	$/.^{0,\infty}/$
27	*zip*	xsd:Decimal
28	95819	$/[0-9]^{1,\infty}/$
29	*comment*	xsd:String
30	Hurry, my lawn is going wild!	$/.^{0,\infty}/$
31	*items*	Items
33	*@partNum*	SKU
34	872-AA	$/[0-9]^{3,3} - [A-Z]^{2,2}/$
35	*productName*	xsd:String
36	Lawnmower	$/.^{0,\infty}/$
37	*quantity*	xsd:PositiveInteger
38	1	$/[0-9]^{1,\infty}/$
39	*USPrice*	xsd:Decimal
40	148.95	$/[0-9]^{1,\infty}/$
41	*comment*	xsd:String
42	Confirm this is electric	$/.^{0,\infty}/$
44	*@partNum*	SKU
45	926-AA	$/[0-9]^{3,3} - [A-Z]^{2,2}/$
46	*productName*	xsd:String
47	Baby Monitor	$/.^{0,\infty}/$
48	*quantity*	xsd:PositiveInteger
49	1	$/[0-9]^{1,\infty}/$
50	*USPrice*	xsd:Decimal
51	39.98	$/[0-9]^{1,\infty}/$
52	*shipDate*	xsd:Date
53	1999-05-21	$/.... - .. - ../$

Note: The node column is redundant, since the pre $\mapsto$ node mapping is already present in the pre/post enumeration.

Table II: Type annotation for `purchaseorder.xml`

Chapter 5

Complexity

5.1 Runtime Behavior

For a given forest f, the size of the pre/post table obviously is in $\mathcal{O}(\sigma f)$.

The space requirements for maintaining only the document to be validated are in $\mathcal{O}(\sigma f)$: The encoded document is not kept in memory, and each node is used exactly once, consumed in pre order. However, the post values have to be remembered inside the produced guards. Since each node produces at most one pair of guards, this yields the given upper bound.

However, the regular expression mangled by the derivation process might become quite big.

5.1.1 Definition The definition of $\mathfrak{X}$ induces a **tree structure** which is formalized as follows:

$$\operatorname{tree}\epsilon := \epsilon \qquad \operatorname{tree}\oslash := \oslash$$

$$\operatorname{tree}(xy) := \begin{array}{ccc} & \cdot & \\ \operatorname{tree} x & & \operatorname{tree} y \end{array} \qquad \operatorname{tree} x^{m,n} := \begin{array}{c} m,n \\ | \\ \operatorname{tree} x \end{array}$$

$$\operatorname{tree}(x|y) := \begin{array}{ccc} & | & \\ \operatorname{tree} x & & \operatorname{tree} y \end{array} \qquad \operatorname{tree}\overset{\overline{p}\ \overline{q}}{\underset{\underline{p}\ \underline{q}}{[x]}} := \begin{array}{c} \underline{p},\overline{p},\underline{q},\overline{q} \\ | \\ \operatorname{tree} x \end{array}$$

$$\operatorname{tree} a\langle x\rangle := \begin{array}{c} a \\ | \\ \operatorname{tree} x \end{array} \qquad \operatorname{tree}\mathfrak{t} := \mathfrak{t}\ .$$

With this, a **subexpression** is simply defined as a subtree. We write $x \prec y$, iff x is a subexpression of y.

5.1.2 Definition The **size** σx of a regular expression $x \in \mathfrak{X}$ is defined by $\forall a \in V,\ \forall x, y \in \mathfrak{X},\ \forall \mathfrak{t} \in T$:

$$\begin{aligned} \sigma\epsilon\ ,\ \sigma\oslash\ ,\ \sigma a\ ,\ \sigma\mathfrak{t} &= 1 \\ \sigma x^{m,n}\ ,\ \sigma[x]_{\underline{q}}^{\overline{q}} &= 1 + \sigma x \\ \sigma(xy)\ ,\ \sigma(x|y) &= 1 + \sigma x + \sigma y \\ \sigma a\langle x\rangle &= 1 + \sigma a + \sigma x\ . \end{aligned}$$

This is a straightforward definition according to the tree structure of the expression, assuming consumption of one "memory unit" for each operand and each operator.

5.1.3 Definition For a grammar $G = (\mathfrak{s}, \lambda) \in \mathcal{G}$, let us define its **size**

$$\sigma G := \sum_{\mathfrak{t} \in T} \sigma\lambda\mathfrak{t} + \#T$$

by simply summing up the size of its components plus the amount of defined names.

Some of the derivation rules do not impose growth of the derived expression — i.e., $\sigma x = \sigma(\partial x)$ — like $\oslash$ or ϵ. Even guards do not increase the size, however the expression therein might do so.

For subtree construction we have $\sigma(\partial a\langle x\rangle) \leq \sigma(a\langle x\rangle) - 1$, which is covered by maintaining the document as depicted above. Also, if x is not nullable, derivation of xy does not enlarge the expression beyond what the derivation of x alone does.

Nonterminals, however, do increase the size of the expression when they are resolved. See *Translating the Schema* on page 38 for a translation of XML Schema's named groups and types into this calculus by means of the nonterminals in T.

For named groups, the depth d of group nesting is limited to the amount of groups defined in the grammar, since no recursion is allowed (Appendix A.1). With l being the length of the longest regular expression that occurs in a given grammar, and $dl < \sigma G$, this leads to space requirements in $\mathcal{O}(\sigma G)$ for deriving one group reference. After derivation of the group, its description has vanished from the regular expression, potentially leaving some guards which are already accounted for by maintenance of the document above. So the space requirements for all group derivations can be satisfied in $\mathcal{O}(\sigma G)$.

Note that in our case space requirements coincide with time requirements. A referenced nonterminal must be copied to prevent the original from suffering the transformation imposed by the derivation process.

For named types, at most σf lookups will be performed, since a type lookup implies the consumption of a node from the forest. Analogous to the reasoning for groups, the references to named types can be handled in $\mathcal{O}(\sigma f)$.

Difficulties start with the alternation construct. Derivation of such an expression doubles processing time, since each branch needs to be processed distinct from the other: $\partial(x|y) = \partial x|\partial y$.

The same problem occurs for an expression xy with $x \in \mathcal{E}$ which introduces an alternation on derivation. For a nullable x, $\partial(xy)$ yields $\partial x[y]|\partial y$. And since the derivation of $x^{m,n}$ may introduce a concatenation like $\partial x[x^{m-1,n-1}]$, this construct is also affected if $\partial x \in \mathcal{E}$.

Worst case scenarios describe a regular expression that yields an alternation construct with each of its branches giving rise to more alternation constructs repeatedly. For example, the expression $(a^{0,\infty})^{0,\infty}$ derived ("$\rightsquigarrow$") according to a sequence of as shows this exponential growth: (Note that in the following calculation the guard limits are omitted for brevity; they are trivially fulfilled)

$$
\begin{array}{rcll}
(a^{0,\infty})^{0,\infty} & \rightsquigarrow & \partial_a a^{0,\infty}[(a^{0,\infty})^{0,\infty}] & \\
& = & \partial_a a[a^{0,\infty}][(a^{0,\infty})^{0,\infty}] & \\
& = & [a^{0,\infty}][(a^{0,\infty})^{0,\infty}] & (\star) \\
& & & \\
& \rightsquigarrow & \partial_a[a^{0,\infty}][(a^{0,\infty})^{0,\infty}] \mid \partial_a[(a^{0,\infty})^{0,\infty}] & \\
& = & \partial_a a[a^{0,\infty}][(a^{0,\infty})^{0,\infty}] \mid [[a^{0,\infty}][(a^{0,\infty})^{0,\infty}]] & \\
& = & [a^{0,\infty}][(a^{0,\infty})^{0,\infty}] \mid [[a^{0,\infty}][(a^{0,\infty})^{0,\infty}]] & 2 \text{ times } (\star) \\
& & & \\
& \rightsquigarrow & [a^{0,\infty}][(a^{0,\infty})^{0,\infty}] \mid [[a^{0,\infty}][(a^{0,\infty})^{0,\infty}]] & \\
& & \quad \mid [[a^{0,\infty}][(a^{0,\infty})^{0,\infty}] \mid [[a^{0,\infty}][(a^{0,\infty})^{0,\infty}]]] \ . & 4 \text{ times } (\star)
\end{array}
$$

The problem is that the derivation process has no knowledge about which one of the alternative branches (if any) will finally make the expression nullable, so all of them need to be kept.

XML Schema restricts (see Appendix A.2) its grammars to be *one-unambiguous* (see Section 5.2). A one-unambiguous regular expression allows to determine exactly the one branch — if any — of a | expression, which can be derived to a nullable expression.

5.1.4 Corollary If the algorithm works on one-unambiguous regular expressions only, time and space constraints are in $\mathcal{O}(\sigma G + \sigma f)$, i.e., linear in the size of the grammar G and the forest f.

However, it turns out that one-unambiguity can be lost during derivation, which potentially yields exponential time and space requirements.

5.2 One-unambiguous Regular Expressions

This section discusses the one-unambiguity for regular expressions, a condition which is required — but, as it will turn out, is not sufficient — to ensure a good runtime behavior.

The only way derivation copes with nonterminals, is replacing the nonterminal by its definition and deriving the replacement (Definition 3.2.1). The XML Schema standard ensures that no infinite loops will occur in doing so (see Appendix A.1). So, *for a given document,* it is safe to consider the derivation process as working on one static regular expression instead of a context free grammar. Simply imagine all the group references being resolved, and the type references being resolved to a depth that will never be reached for the given document forest. In fact, this is exactly what the algorithm does "on the fly".

Relating to XML Schema, one also talks about one-unambiguous grammars. With one-unambiguous regular expressions, which are to be defined in Definition 5.2.9, we note the following definition:

5.2.1 Definition A grammar is called **one-unambiguous** , iff any such expansion of a context free grammar to a regular expression will yield a one-unambiguous regular expression.

Now it is sufficient to focus the discussion on the one-unambiguity of regular expressions. Nonterminals and the lookup function λ will not occur within this section any more.

5.2.2 Definition The set of **marked nodes** is defined as the nodes used so far, indexed with natural numbers:

$$V' := V \times \mathbb{N} \qquad\qquad a_i := (a, i) \in V' \ .$$

5.2.3 Definition The set of **marked expressions** is defined by

$$\mathfrak{X}' := \{x \mid x \in \mathfrak{X}_{V'} \wedge \forall a', b' \in V',\ a', b' \prec x : a' \neq b'\}$$

which simply states that each occurrence of a node in the expression can be identified uniquely by its index. Such a **marking** x' of an expression x can be achieved, e.g., by reading the expression from left to right and annotating each node in V with strictly increasing numbers.

Without loss of generality, assume the markings of subexpressions to be consistent with the marking of the overall expression, i.e., $x \prec z \Rightarrow x' \prec z'$.

This marking is a widely used approach to explain one-unambiguity, e.g., used in [2].

Theorem 3.3.1 justifies the definition of a matching between an enumeration and a regular expression with guards as follows:

5.2.4 Definition The **matching** relation "$\vDash$" $\subset \mathcal{N} \times \mathfrak{X}$ is defined by

$$\forall N \in \mathcal{N} : \forall x \in \mathfrak{X} : (N \vDash x :\Longleftrightarrow \partial_N x \in \mathcal{E}) \ .$$

Note that this matching relation is defined not only on the normalized enumerations. This is useful for a discussion of subexpressions and subtrees. Remark 2.5.8 implies that the enumeration of a tree always yields enumerations of its subtrees as subsets of the overall enumeration.

Also note that one enumeration might match an expression, even though another enumeration of the same forest may not match that expression.

The following discussion observes in which order nodes in a matching enumeration may appear. The values of the guards are ignored, since the most general case must be considered.

5.2.5 Definition The **first set** of a regular expression is defined by the function

$$
\begin{array}{rrcl}
\mathrm{first}: & \mathfrak{X} & \longrightarrow & \mathcal{P}V \\
& \oslash & \longmapsto & \emptyset \\
& \epsilon & \longmapsto & \emptyset \\
& [x] & \longmapsto & \mathrm{first}\, x \\
& a\langle x\rangle & \longmapsto & \{a\} \\
& xy & \longmapsto & \begin{cases} \emptyset & \text{if } \mathrm{first}\, y = \emptyset \wedge y \notin \mathcal{E} \\ \mathrm{first}\, x & \text{otherwise, if } x \notin \mathcal{E} \\ \mathrm{first}\, x \cup \mathrm{first}\, y & \text{otherwise} \end{cases} \\
& x|y & \longmapsto & \mathrm{first}\, x \cup \mathrm{first}\, y \\
& x^{m,n} & \longmapsto & \mathrm{first}\, x \ .
\end{array}
$$

Hence, $\mathrm{first}\, x$ is the set that contains all those nodes an enumeration matching x might begin with. The first case of xy is necessary to handle $y = \oslash$ correctly.

5.2.6 Definition For any $b \in V$, the **follow set** of a regular expression is defined by the function

$$
\begin{array}{rrcl}
\mathrm{follow}_b: & \mathfrak{X} & \longrightarrow & \mathcal{P}V \\
& \oslash & \longmapsto & \emptyset \\
& \epsilon & \longmapsto & \emptyset \\
& [x] & \longmapsto & \mathrm{follow}_b\, x \\
& a\langle x\rangle & \longmapsto & \begin{cases} \mathrm{follow}_b\, x \cup \mathrm{first}\, x & \text{if } a = b \\ \mathrm{follow}_b\, x & \text{otherwise} \end{cases} \\
& xy & \longmapsto & \begin{cases} \mathrm{follow}_b\, x \cup \mathrm{follow}_b\, y \cup \mathrm{first}\, y & \text{if } b \in \mathrm{last}\, x \\ \mathrm{follow}_b\, x \cup \mathrm{follow}_b\, y & \text{otherwise} \end{cases} \\
& x|y & \longmapsto & \mathrm{follow}_b\, x \cup \mathrm{follow}_b\, y \\
& x^{m,n} & \longmapsto & \begin{cases} \mathrm{follow}_a\, x \cup \mathrm{first}\, x & \text{if } b \in \mathrm{last}\, x \wedge n > 1 \\ \mathrm{follow}_a\, x & \text{otherwise} \ . \end{cases}
\end{array}
$$

$\mathrm{follow}_b\, x$ is the "intermediate" variant of first, denoting all those nodes that might follow $b \in V$ in an enumeration matching x.

5.2.7 Definition The **last set** of a regular expression is defined by the function

$$\begin{array}{rrcl}
\text{last}: & \mathfrak{X} & \longrightarrow & \mathcal{P}V \\
& \oslash & \longmapsto & \emptyset \\
& \epsilon & \longmapsto & \emptyset \\
& [x] & \longmapsto & \text{last}\, x \\
& a\langle x\rangle & \longmapsto & \begin{cases} \{a\} \cup \text{last}\, x & \text{if } x \in \mathcal{E} \\ \text{last}\, x & \text{otherwise} \end{cases} \\
& xy & \longmapsto & \begin{cases} \emptyset & \text{if } \text{last}\, x = \emptyset \wedge x \notin \mathcal{E} \\ \text{last}\, y & \text{otherwise, if } y \notin \mathcal{E} \\ \text{last}\, x \cup \text{last}\, y & \text{otherwise} \end{cases} \\
& x|y & \longmapsto & \text{last}\, x \cup \text{last}\, y \\
& x^{m,n} & \longmapsto & \text{last}\, x \ .
\end{array}$$

The last function is the counterpart of first, denoting the nodes that might appear at the end of a matching enumeration.

5.2.8 Definition The **follow last set** of a regular expression is defined by the function

$$\begin{array}{rrcl}
\text{followlast}: & \mathfrak{X} & \longrightarrow & \mathcal{P}V \\
& x & \longmapsto & \{a \mid \exists b \in \text{last}\, x : a \in \text{follow}_b\, x\} \ .
\end{array}$$

So followlast denotes those nodes that may continue an already valid enumeration. Let first, last, follow, and followlast be defined likewise for $\mathfrak{X}'$ and V' instead of $\mathfrak{X}$ and V.

5.2.9 Definition The **one-unambiguous** regular expressions are defined by

$$\begin{array}{rl}
\mathfrak{X}_1 := \{x \mid & \forall a_i, a_j \in \text{first}\, x' : i = j \\
\wedge & \forall c' \in V' : \forall a_i, a_j \in \text{follow}_{c'}\, x' : i = j \quad \} \ .
\end{array}$$

If a regular expression is one-unambiguous, we are able to select at most one alternative branch during derivation according to a single node that will become nullable. Fortunately, the XML Schema standard restricts its grammars to yield one-unambiguous regular expressions only, i.e., expressions like $ab|ac$ are not allowed, they will appear not even when resolving nonterminals (Appendix A.2).

Unfortunately, however, a one-unambiguous regular expression does not necessarily derive to a one-unambiguous expression. As an example (which is taken from [2]), have a look at $x := ((ab)^{0,\infty}|c)^{0,\infty}$. This clearly is one-unambiguous, since, for $x' = ((a_1b_2)^{0,\infty}|c_3)^{0,\infty}$, there are only the tree symbols a_1, b_2, and c_3.

Deriving x according to a, however, yields

$$\begin{array}{rcll}
\partial_a x & = & \partial_a((ab)^{0,\infty}|c)^{0,\infty} & \\
& = & \partial_a((ab)^{0,\infty}|c) & \cdot\ [((ab)^{0,\infty}|c)^{0,\infty}] \\
& = & \left(\partial_a(ab)^{0,\infty}\middle|\partial_a c\right) & \cdot\ [((ab)^{0,\infty}|c)^{0,\infty}] \\
& = & \left(\partial_a(ab)[(ab)^{0,\infty}]\middle|\oslash\right) & \cdot\ [((ab)^{0,\infty}|c)^{0,\infty}] \\
& = & \left(b[(ab)^{0,\infty}]\middle|\oslash\right) & \cdot\ [((ab)^{0,\infty}|c)^{0,\infty}]
\end{array}$$

which, marked as

$$\big(b_1[(a_2b_3)^{0,\infty}]\big|\oslash\big)\ [((a_4b_5)^{0,\infty}|c_6)^{0,\infty}]$$

yields

$$\mathrm{follow}_{b_3}(\partial_a x)' = \{a_2, a_4\}$$

violating the one-unambiguity condition.

So, having a one-unambiguous regular expression alone, does not guarantee that derivation cannot generate an ambiguous expression which potentially yields bad runtime behavior.

5.3 The Star Normal Form

To solve this problem, [2] introduces a *star normal form* for regular expressions that use the Kleene closure * instead of the ^{m,n} operator ($a^* = a^{0,\infty}$), and are restricted to strings instead of trees. An adoption to the context of this thesis is the Definition 5.3.4 given below.

Therefore let us introduce another definition of one-unambiguity, which eases proof and formalization of the according theorems.

5.3.1 Theorem An equivalent definition of one-unambiguity is

$$\begin{aligned} z \in \mathfrak{X}_1 \iff\ & \forall xy \prec z : x \notin \mathcal{E} \ ^\vee \mathrm{first}\, x \cap \mathrm{first}\, y = \emptyset && ① \\ & ^\wedge \forall xy \prec z : \mathrm{followlast}\, x \cap \mathrm{first}\, y = \emptyset && ② \\ & ^\wedge \forall x|y \prec z : \mathrm{first}\, x \cap \mathrm{first}\, y = \emptyset && ③ \end{aligned}$$

omitting the marking of the alphabet V.

The proof is deferred to 5.3.3 below.

As an example, the expression $\big(b[(ab)^{0,\infty}]\big|\oslash\big) \cdot [((ab)^{0,\infty}|c)^{0,\infty}]$ from above is not one-unambiguous, since

$$a \in \mathrm{followlast}\big(b[(ab)^{0,\infty}]\big|\oslash\big) \cap \mathrm{first}[((ab)^{0,\infty}|c)^{0,\infty}]$$

holds, violating ②.

5.3.2 Corollary Theorem 5.3.1 directly shows that

$$\forall z \in \mathfrak{X}_1 : \forall x \prec z : x \in \mathfrak{X}_1\ .$$

That is, each subexpression of a one-unambiguous expression must be one-unambiguous in turn.

5.3.3 Proof of Theorem 5.3.1.
We prove both directions of the theorem individually:

▷ **Part I** "$\Longleftarrow$".
Assuming that $z \in \mathfrak{X}$ and that ①, ② and ③ hold, we need to show that $z \in \mathfrak{X}_1$. Therefore we look at a', b' either in $\mathrm{first}\, z'$ (Case I.a) or in $\mathrm{follow}_{c'}\, z'$ for a $c' \in V'$ (Case I.b), and prove that either $a \neq b$ or $a' = b'$ holds. Induction on the structure of z is used.

▷ **Case I.a** $a', b' \in \mathrm{first}\, z'$.
The cases $z = \oslash$ and $z = \epsilon$ will not occur, since then $\mathrm{first}\, z' = \emptyset$.

▷ **Case I.a.1** $z = [x]$.
This case is covered by induction.

▷ **Case I.a.2** $z = d\langle x \rangle$.
Obviously, $a' = b' = d'$ holds.

▷ **Case I.a.3** $z = xy$.
This implies

$$\begin{aligned}&\big(a', b' \in \mathrm{first}\, x'\big) \vee \big(x \in \mathcal{E} \wedge a', b' \in \mathrm{first}\, y'\big)\\ &\vee\big(x \in \mathcal{E} \wedge a' \in \mathrm{first}\, x' \wedge b' \in \mathrm{first}\, y'\big) \vee \big(x \in \mathcal{E} \wedge b' \in \mathrm{first}\, x' \wedge a' \in \mathrm{first}\, y'\big)\ .\end{aligned}$$

The alternatives from the first line are covered by induction. For the alternatives from the second line, ① implies $a \neq b$.

▷ **Case I.a.4** $z = x|y$.
This implies

$$\begin{aligned}&\big(a', b' \in \mathrm{first}\, x'\big) \vee \big(a', b' \in \mathrm{first}\, y'\big)\\ &\vee\big(a' \in \mathrm{first}\, x' \wedge b' \in \mathrm{first}\, y'\big) \vee \big(b' \in \mathrm{first}\, x' \wedge a' \in \mathrm{first}\, y'\big)\ .\end{aligned}$$

Again, the alternatives from the first line are covered by induction. For the alternatives from the second line, ③ implies $a \neq b$.

▷ **Case I.a.5** $z = x^{m,n}$.
This implies $a', b' \in \mathrm{first}\, x'$, which is covered by induction.

▷ **Case I.b** $a', b' \in \mathrm{follow}_{c'}\, z'$ for a $c \in V$.

▷ **Case I.b.1** $z = xy$.
This implies

$$\begin{aligned}&a', b' \in \mathrm{follow}_{c'}\, x' \vee a', b' \in \mathrm{follow}_{c'}\, y' \vee a', b' \in \mathrm{first}\, y'\\ &\vee\big(a' \in \mathrm{followlast}\, x' \wedge b' \in \mathrm{first}\, y'\big) \vee \big(b' \in \mathrm{followlast}\, x' \wedge a' \in \mathrm{first}\, y'\big)\ .\end{aligned}$$

The first line is covered by induction, for the second one, ② yields $a \neq b$.

▷ **Case I.b.2** $z = x^{m,n}$.
This implies

$$a', b' \in \mathrm{follow}_{c'}\, x' \vee a', b' \in \mathrm{first}\, x'$$

which is covered by induction. Note that the case $a' \in \mathrm{followlast}\, x' \wedge b' \in \mathrm{first}\, x'$ is already covered by $a', b' \in \mathrm{follow}_{c'}\, x'$.

▷ **Case I.b.3** All the remaining cases are either trivial, or covered by induction directly:

$$\begin{aligned} z = [x] &\Rightarrow a', b' \in \mathrm{follow}_{c'}\, x' \\ z = d\langle x \rangle &\Rightarrow a', b' \in \mathrm{follow}_{c'}\, x' \vee a', b' \in \mathrm{first}\, x' \\ z = x|y &\Rightarrow a', b' \in \mathrm{follow}_{c'}\, x' \vee a', b' \in \mathrm{follow}_{c'}\, y' \\ z = x^{m,n} &\Rightarrow a', b' \in \mathrm{follow}_{c'}\, x' \vee \left(c' \in \mathrm{last}\, x' \wedge a', b' \in \mathrm{first}\, x'\right) \ . \end{aligned}$$

Again, $z = \oslash$ and $z = \epsilon$ will not occur, since then $\mathrm{follow}_c\, z = \emptyset$.

▷ **Part II** "$\Longrightarrow$".
Assuming that $z \in \mathfrak{X}_1$, we need to show that ①, ②, and ③ hold. Otherwise, let there be a subexpression xy or $x|y$ of z, violating one of the conditions.
Consistency of the marking implies $x'y' = (xy)' \prec z'$ and similar for $x|y$. For $a_i \prec x'$ and $a_j \prec y'$, this already implies $i \neq j$.

▷ **Case II.a** Violation of ①.
Assume $x \in \mathcal{E}$, $xy \prec z$, $a_i \in \mathrm{first}\, x'$ and $a_j \in \mathrm{first}\, y'$. With $x \in \mathcal{E}$ we find that

$$a_i, a_j \in \mathrm{first}(xy)' \tag{5}$$

i.e., a_i and a_j are members of the first set of the same subexpression xy in z.

▷ **Case II.a.1** $a_i \in \mathrm{first}\, z'$.
(5) implies $a_j \in \mathrm{first}\, z'$, contradicting $z \in \mathfrak{X}_1$.

▷ **Case II.a.2** $a_i \notin \mathrm{first}\, z'$.
Again, (5) implies

$$\exists c \in V : a_i \in \mathrm{follow}_{c'}\, z' \wedge a_j \in \mathrm{follow}_{c'}\, z'$$

which contradicts $z \in \mathfrak{X}_1$.

▷ **Case II.b** Violation of ②.
Assume $xy \prec z$, $a_i \in \mathrm{followlast}\, x'$ and $a_j \in \mathrm{followlast}\, y'$. This implies

$$\exists c' \in \mathrm{last}\, x' : a_i \in \mathrm{follow}_{c'}\, z' \wedge a_j \in \mathrm{follow}_{c'}\, z'$$

which contradicts $z \in \mathfrak{X}_1$.

▷ **Case II.c** Violation of ③.
The reasoning is the same as in Case II.a, however $x \in \mathcal{E}$ is not needed, since $\operatorname{first} y \subseteq \operatorname{first}(x|y)$ trivially holds for all $x, y \in \mathfrak{X}$.

□

5.3.4 Definition An expression $z \in \mathfrak{X}$ is said to be in **star normal form** (**SNF**), iff

$$\forall (m,n) \in \mathbf{R} : \forall x^{m,n} \prec z : \operatorname{followlast} x \cap \operatorname{first} x = \emptyset \ . \qquad ④$$

Let $\mathfrak{X}_*$ denote the regular expressions in star normal form.

5.3.5 Remark The definition implies that each subexpression of an expression in SNF must in turn be in SNF.

5.3.6 Lemma Expressions in SNF derive to expressions in SNF, i.e.,

$$\forall z \in \mathfrak{X}_* : \forall a \in V : \partial_a z \in \mathfrak{X}_* \ .$$

The proof is deferred to 5.3.9 below.

5.3.7 Theorem The derivative of a one-unambiguous expression in SNF is one-unambiguous, i.e.,

$$\forall z \in \mathfrak{X}_1 \cap \mathfrak{X}_* : \forall a \in V : \partial_a z \in \mathfrak{X}_1 \ .$$

The proof is deferred to 5.3.12 below.

5.3.8 Corollary Lemma 5.3.6 and Theorem 5.3.7 directly yield that $\mathfrak{X}_1 \cap \mathfrak{X}_*$ is closed under derivation.

This is an adoption of what [2] has stated for regular expressions on strings.

5.3.9 Proof of Lemma 5.3.6.
Simple induction on the structure of an expression $z \in \mathfrak{X}_*$ yields that the only case where derivation of z might create an expression violating ④, is the derivation of a subexpression $x^{m,n} \prec z$.

For $n = 1$, the derivation yields $\partial_a x$, which is in SNF, since the subexpression in question looses its ^{m,n} operator. For $n > 1$ we will get $\partial_a x[x^{m',n'}]$, which is in SNF, since $x^{m,n}$ already satisfied ④. □

5.3.10 Lemma The following implications are used during the proof of Theorem 5.3.7. For $z \in \mathfrak{X}$ and $a, b \in V$

$$\partial_a z \in \mathcal{E} \Rightarrow a \in \operatorname{last} z \wedge a \in \operatorname{first} z \qquad (6)$$
$$\operatorname{first} \partial_a z \neq \emptyset \Rightarrow a \in \operatorname{first} z \qquad (7)$$
$$b \in \operatorname{first} \partial_a z \Rightarrow b \in \operatorname{follow}_a z \qquad (8)$$
$$b \in \operatorname{followlast}(\partial_a z) \Rightarrow b \in \operatorname{followlast} z \qquad (9)$$

hold.

5.3.11 Proof of Lemma 5.3.10.
For all three implications, induction on the structure of z is used.

▷ **Part I** Proof of (*6*).
Assume $\partial_a z \in \mathcal{E}$. The cases $z = \oslash$ and $z = \epsilon$ cannot occur, since then $\partial_a z = \oslash \notin \mathcal{E}$.

▷ **Case I.a** $z = [x]$.
The assumption implies $\partial_a x \in \mathcal{E}$. So induction yields the claim.

▷ **Case I.b** $z = b\langle x \rangle$.
The assumption implies $a = b \wedge x \in \mathcal{E}$ which in turn yields the claim.

▷ **Case I.c** $z = xy$.
The assumption implies

$$\begin{array}{rll} & \partial_a x[y] \in \mathcal{E} & \vee \ \big(x \in \mathcal{E} \wedge \partial_a y \in \mathcal{E}\big) \\ \Rightarrow & \big(\partial_a x \in \mathcal{E} \wedge y \in \mathcal{E}\big) & \vee \ \big(x \in \mathcal{E} \wedge \partial_a y \in \mathcal{E}\big) \\ \Rightarrow & \big(a \in \mathrm{first}\, x \wedge a \in \mathrm{last}\, x \wedge y \in \mathcal{E}\big) & \vee \ \big(a \in \mathrm{first}\, y \wedge a \in \mathrm{last}\, y \wedge x \in \mathcal{E}\big) \end{array}$$

which implies $a \in \mathrm{first}\, z \wedge a \in \mathrm{last}\, z$.

▷ **Case I.d** $z = x|y$.
The assumption implies $\partial_a x \in \mathcal{E} \vee \partial_a y \in \mathcal{E}$ which yields the claim by induction.

▷ **Case I.e** $z = x^{m,n}$.
The assumption implies $\partial_a x \in \mathcal{E}$. So induction yields $a \in \mathrm{first}\, x \wedge a \in \mathrm{last}\, x$ which yields $a \in \mathrm{first}\, z \wedge a \in \mathrm{last}\, z$.

▷ **Part II** Proof of (*7*).
Assume $\partial_a z \neq \emptyset$. The cases $z = \oslash$ and $z = \epsilon$ cannot occur, since then $\mathrm{first}\, \partial_a z = \emptyset$.

▷ **Case II.a** $z = [x]$.
The assumption yields $\mathrm{first}\, \partial_a x \neq \emptyset$, hence induction yields the claim.

▷ **Case II.b** $z = b\langle x \rangle$.
The assumption yields $\partial_a z \neq \oslash \Rightarrow a = b \in \mathrm{first}\, z$.

▷ **Case II.c** $z = xy$.
Here, the assumption yields

$$\mathrm{first}(\partial_a x[y]) \neq \emptyset \vee \big(x \in \mathcal{E} \wedge \mathrm{first}\, \partial_a y \neq \emptyset\big) \ .$$

Both cases imply $a \in \mathrm{first}\, z$, due to induction.

▷ **Case II.d** $z = x|y$.
With $\mathrm{first}\, \partial_a z = \mathrm{first}\, \partial_a x \cup \mathrm{first}\, \partial_a y$, induction yields the claim.

▷ **Case II.e** $z = x^{m,n}$.
The assumption implies

$$\text{first}\,\partial_a x \neq \emptyset \,\vee\, \big(n \geq 1 \,\wedge\, \partial_a x \in \mathcal{E}\big) \ .$$

Both cases — the first one due to induction, the second one due to (*6*) — yield $a \in \text{first}\, z$.

▷ **Part III** Proof of (*8*).
Assume $b \in \text{first}\,\partial_a z$. The cases $z = \oslash$ and $z = \epsilon$ cannot occur, since then $\text{first}\,\partial_a z = \emptyset$.

▷ **Case III.a** $z = [x]$.
The assumption implies $b \in \text{first}\,\partial_a x$. Due to induction, this yields the claim.

▷ **Case III.b** $z = c\langle x\rangle$.
The assumption yields $a = c \,\wedge\, b \in \text{first}\, x$, hence $b \in \text{follow}_a\, z$.

▷ **Case III.c** $z = xy$ The assumption yields

$$\begin{array}{rll} & b \in \text{first}\,\partial_a x[y] & \vee \quad \big(x \in \mathcal{E} \,\wedge\, b \in \text{first}\,\partial_a y\big) \\ \Rightarrow & \underbrace{b \in \text{first}\,\partial_a x}_{(\star)} \vee \big(\partial_a x \in \mathcal{E} \,\wedge\, b \in \text{first}\, y\big) & \vee \quad \underbrace{\big(x \in \mathcal{E} \,\wedge\, b \in \text{first}\,\partial_a y\big)}_{(\star)} \ . \end{array}$$

The first and last case — marked with a $(\star)$ — imply the claim due to induction. The remaining case implies $a \in \text{last}\, x$ due to (*6*), which in turn implies $b \in \text{follow}_a\, z$, since $b \in \text{first}\, y$.

▷ **Case III.d** $z = x|y$.
The assumption implies $b \in \text{first}\,\partial_a x \,\vee\, b \in \text{first}\,\partial_a y$, which yields the claim due to induction.

▷ **Case III.e** $z = x^{m,n}$.
The assumption implies

$$\begin{array}{rll} & b \in \text{first}\,\partial_a x & \vee \quad \big(\partial_a x \in \mathcal{E} \,\wedge\, n > 1 \,\wedge\, b \in \text{first}\, x\big) \\ \Rightarrow & \underbrace{b \in \text{follow}_a\, x}_{\text{due to induction}} & \vee \quad \big(\underbrace{a \in \text{last}\, x}_{\text{due to (6)}} \wedge n > 1 \,\wedge\, b \in \text{first}\, x\big) \end{array}$$

which both yield the claim.

▷ **Part IV** Proof of (*9*).
Assume $b \in \text{followlast}\,\partial_a z$. The cases $z = \oslash$ and $z = \epsilon$ cannot occur, since then $\text{followlast}\,\partial_a z = \emptyset$.

▷ **Case IV.a** $z = [x]$.
The assumption implies $b \in \text{followlast}\,\partial_a x$, hence induction yields the claim.

▷ **Case IV.b** $z = c\langle x\rangle$.
The assumption yields $a = c \,\wedge\, b \in \text{followlast}\, x$. Then, $\text{followlast}\, x \subseteq \text{followlast}\, z$ yields the claim.

▷ **Case IV.c** $z = xy$.
The assumption yields

$$b \in \text{followlast}(\partial_a x[y]) \vee \big(x \in \mathcal{E} \wedge b \in \text{followlast}\, \partial_a y\big) \ .$$

The second case implies the claim due to induction. The first one implies

$$b \in \text{followlast}[y] \vee \big(y \in \mathcal{E} \wedge b \in \text{followlast}\, \partial_a x\big) \vee \big(y \in \mathcal{E} \wedge b \in \text{first}\, y\big) \ .$$

Again, the first two cases yield the claim due to induction. For the remaining case, we know that

$$y \in \mathcal{E} \Rightarrow \text{last}\, x \subseteq \text{last}\, z$$
$$b \in \text{first}\, y \Rightarrow \forall c \in \text{last}\, x : b \in \text{follow}_c\, z$$

hold. Together, this implies $b \in \text{followlast}\, z$.

▷ **Case IV.d** $z = x|y$.
Since $\text{followlast}\, z = \text{followlast}\, x \cup \text{followlast}\, y$, induction yields the claim.

▷ **Case IV.e** $z = x^{m,n}$.
For $n = 1$, the assumption yields $b \in \text{followlast}\, \partial_a x$, hence induction proves the claim. Let $n > 1$. Then $b \in \text{followlast}\, \partial_a x[x^{m',n'}]$ implies

$$\underbrace{b \in \text{followlast}\, x^{m',n'}}_{\Rightarrow b \in \text{followlast}\, x} \vee \big(x^{m',n'} \in \mathcal{E} \wedge \big(\underbrace{b \in \text{followlast}\, \partial_a x}_{\text{induction proves the claim}} \vee \underbrace{b \in \text{first}\, x^{m',n'}}_{\star}\big)\big) \ .$$

With $x^{m',n'} \in \mathcal{E}$, the term $\star$ implies that

$$\forall c \in \text{last}\, \partial_a x : b \in \text{follow}_c\, z$$
$$\text{last}\, \partial_a x \subseteq \text{last}\, \partial_a z$$

hold. Together, this yields the claim.

□

5.3.12 Proof of Theorem 5.3.7.
Assume $z \in \mathfrak{X}_1 \cap \mathfrak{X}_*$. If, for any $a \in V$, we had $\partial_a z \notin \mathfrak{X}_1$, then one of the conditions ①, ② or ③ must have been violated for $\partial_a z$. We show that this implies $z \notin \mathfrak{X}_1$ or $z \notin \mathfrak{X}_*$. Induction on the structure of z is used.

▷ **Case I** $z = \oslash$ or $z = \epsilon$ always yields $\partial_a z \in \mathfrak{X}_1$.

▷ **Case II** $z = [x]$.
$\partial_a z \notin \mathfrak{X}_1 \Rightarrow \partial_a x \notin \mathfrak{X}_1$, which is covered by induction.

▷ **Case III** $z = b\langle x \rangle$.
$\partial_a b\langle x \rangle \notin \mathfrak{X}_1 \Rightarrow x \notin \mathfrak{X}_1$ contradicting the assumption.

▷ **Case IV** $z = xy$.

▷ **Case IV.a** $x \in \mathcal{E}$.
Here $\partial_a z \notin \mathfrak{X}_1$ implies that $\partial_a xy | \partial_a y$ violates one of the one-unambiguity conditions. However, $z \in \mathfrak{X}_1$ ensures that x and y are one-unambiguous, and induction yields $\partial_a x$ and $\partial_a y$ being one-unambiguous, leaving only the following cases:

▷ **Case IV.a.1** Violation of ①

$$\begin{aligned}&\partial_a x \in \mathcal{E} \wedge \exists b \in \operatorname{first}(\partial_a x) \cap \operatorname{first} y \\ &\Rightarrow a \in \operatorname{last} x \wedge b \in \operatorname{follow}_a x \wedge b \in \operatorname{first} y \quad \text{due to (6), (8)} \\ &\Rightarrow b \in \operatorname{followlast} x \wedge b \in \operatorname{first} y\end{aligned}$$

contradicting ① for z.

▷ **Case IV.a.2** Violation of ②

$$\begin{aligned}&\exists b \in \operatorname{followlast} \partial_a x \cap \operatorname{first} y \\ &\Rightarrow b \in \operatorname{followlast} x \wedge b \in \operatorname{first} y \quad \text{due to (9)}\end{aligned}$$

contradicting ② for z.

▷ **Case IV.a.3** Violation of ③

$$\begin{aligned}&\exists b \in \operatorname{first}(\partial_a xy) \cap \operatorname{first} \partial_a y \\ &\Rightarrow \partial_a x \neq \oslash \wedge \partial_a y \neq \oslash \\ &\Rightarrow a \in \operatorname{first} x \wedge a \in \operatorname{first} y \quad \text{due to (7)} \ .\end{aligned}$$

Since $x \in \mathcal{E}$, this violates ① for z.

▷ **Case IV.b** $x \notin \mathcal{E}$.
The reasoning is just like the one used for $x \in \mathcal{E}$, however, the term $\partial_a y$ does not exist in the derivation, so the last case will not appear.

▷ **Case V** $z = x|y$.
Again, $z \in \mathfrak{X}_1$ asserts x and y being one-unambiguous. Induction yields one-unambiguity for $\partial_a x$ and $\partial_a y$. So the outermost | construct must violate condition ③.

$$\begin{aligned}&\exists b \in \operatorname{first} \partial_a x \cap \operatorname{first} \partial_a y \\ &\Rightarrow a \in \operatorname{first} x \wedge a \in \operatorname{first} y \quad \text{due to (7)}\end{aligned}$$

contradicting ③ for z.

▷ **Case VI** $z = x^{m,n}$.
The case $n = 1$ implies that $\partial_a x \notin \mathfrak{X}_1$, so induction proves the claim. Let $n > 1$, and $\partial_a z = \partial_a x[x^{m',n'}]$, where m' and n' shall be defined as induced by Definition 3.2.1. Again, we know that x and $\partial_a x$ are one-unambiguous, leaving the following two cases:

▷ **Case VI.a** Violation of ①

$$\begin{aligned}&\partial_a x \in \mathcal{E} \wedge \exists b \in \operatorname{first} \partial_a x \cap \operatorname{first} x^{m',n'}\\ &\Rightarrow a \in \operatorname{last} x \wedge b \in \operatorname{follow}_a x \wedge b \in \operatorname{first} x \quad \text{due to (6), (8)}\\ &\Rightarrow b \in \operatorname{followlast} x \wedge b \in \operatorname{first} x\end{aligned}$$

contradicting the SNF condition ④.

▷ **Case VI.b** Violation of ②

$$\begin{aligned}&\exists b \in \operatorname{followlast} \partial_a x \cap \operatorname{first} x^{m',n'}\\ &\Rightarrow b \in \operatorname{followlast} x \wedge b \in \operatorname{first} x \quad \text{due to (9)}\end{aligned}$$

again, contradicting the SNF condition ④.

□

5.4 Conclusion

5.4.1 Corollary Corollary 5.1.4 and Corollary 5.3.8 imply that the proposed algorithm validates a forest f against a regular expression $x \in \mathfrak{X}_1 \cap \mathfrak{X}_*$ with linear time and space requirements, i.e., in $\mathcal{O}(\sigma x + \sigma f)$.

Even though a violation of the star normal form may lead to ambiguity, the "indistinguishable" nodes will have the same type annotations associated. This is quite evident, since they arise from the same x in the expression $x^{m,n}$ which violates ④. However, a naive implementation may generate multiple (identical) type annotations for the same node, once for each of its appearances in the regular expression. *Missing SNF Example* on page 82 demonstrates this.

5.4.2 Corollary Ambiguities introduced during the derivation due to a violation of the star normal form do not introduce polymorphism as described in *Polymorphism* on page 41.

5.5 Obtaining SNF

Since XML only restricts its grammars to be one-unambiguous but does not enforce them to the star normal form, one would like to have an algorithm to transform each one-unambiguous grammar into a one-unambiguous grammar in star normal form.

Brüggemann-Klein [3] showed how to transform one-unambiguous regular expressions into one-unambiguous regular expressions in star normal form in linear time. However, that algorithm is not applicable to this situation, since XML's

regular expressions employ the ^{m,n} operation, which is stronger (i.e., more expressive) than Kleene's closure * used there:

$$\forall (m,n) \in \mathbf{R} : \{f \mid f \vDash x^{m,n}\} \subseteq \{f \mid f \vDash x^*\} \subseteq \{f \mid f \vDash x^{0,\infty}\} \ .$$

5.5.1 Definition The transformation given in [3] employs an intermediate form of the expression to be mangled. A straightforward adaption that introduces subtree construction and replaces the Kleene * by ^{m,n}, is

$$\begin{array}{rcll}
\text{inter}: & \mathfrak{X} & \longrightarrow & \mathfrak{X} \\
& \oslash, \epsilon & \longmapsto & \oslash \\
& a\langle x \rangle & \longmapsto & a\langle x \rangle \\
& x|y & \longmapsto & \text{inter}\, x | \,\text{inter}\, y \\
& xy & \longmapsto & \begin{cases} xy & \text{if } x, y \notin \mathcal{E} \\ (\text{inter}\, x)y & \text{if } x \notin \mathcal{E} \wedge y \in \mathcal{E} \\ x \cdot \text{inter}\, y & \text{if } x \in \mathcal{E} \wedge y \notin \mathcal{E} \\ \text{inter}\, x | \,\text{inter}\, y & \text{if } x, y \in \mathcal{E} \end{cases} \\
& x^{m,n} & \longmapsto & \text{inter}\, x \ . \qquad (\star)
\end{array}$$

With this, the following function is intended to transform a regular expression into an equivalent regular expression in star normal form. Again, this is a smooth adaption that introduces subtree construction and replaces the Kleene * by ^{m,n}:

$$\begin{array}{rcll}
\text{snf}: & \mathfrak{X} & \longrightarrow & \mathfrak{X} \\
& \oslash & \longmapsto & \oslash \\
& \epsilon & \longmapsto & \epsilon \\
& a\langle x \rangle & \longmapsto & a\langle \text{snf}\, x \rangle \\
& x|y & \longmapsto & \text{snf}\, x | \,\text{snf}\, y \\
& xy & \longmapsto & \text{snf}\, x \cdot \text{snf}\, y \\
& x^{m,n} & \longmapsto & (\text{inter}(\text{snf}\, x))^{m,n} \ . \qquad (\star)
\end{array}$$

As one can see, the lines marked with $(\star)$ remove "inner" repetition operators to be handled at an "outer" level of the expression. A basic assumption in [3] is that

$$x^{0,\infty} = (\text{inter}\, x)^{0,\infty}$$

holds. However the more general version — which is required in this context — namely that

$$x^{m,n} = (\text{inter}\, x)^{m,n} \qquad \text{(false)}$$

holds, is not true any more. Simply reconstruct that $(a^{1,2})^{3,4} \neq a^{3,4}$.

The problem is only due to the "counting" capability of the ^{m,n} operation. Recalling the example on page 54, we get

$$\text{snf}((ab)^{0,\infty}|c)^{0,\infty} \quad = \quad (ab|c)^{0,\infty}$$

which is an equivalent regular expression in star normal form. The inner repetition of ab is accounted for by the outer one. However, the concept fails when applied to expressions that make use of the ^{m,n} operators extra capability of expression. The proposed rules transform

$$\mathrm{snf}((ab)^{2,3}|c)^{5,7} \quad = \quad (ab|c)^{5,7}$$

which is in star normal form, but does not represent the initial expression any more, since the outer repetition construct cannot account for the inner one.

Of course, it is possible to *unfold* an expression $x^{m,n}$ into an equivalent expression employing Kleene's closure only:

$$x^{m,n} = \begin{cases} \underbrace{x \ldots x}_{m \text{ times}} \cdot \underbrace{(\epsilon|x) \ldots (\epsilon|x)}_{n-m \text{ times}} & \text{if } n < \infty \\ \underbrace{x \ldots x}_{m \text{ times}} \cdot x^* & \text{otherwise .} \end{cases}$$

In fact, each translation into a *finite state automaton* implementing $x^{m,n}$ is required to do such an unfolding first: It is well known that finite automata cannot count, i.e., are not able to determine how often a transition has been performed. So the only way to implement a^m is by introducing $m-1$ states, connected via a transitions. In general, implementation of x^m is done by introducing m sub-automata implementing x, connected by ϵ-transitions.

This unfolding, however, introduces ambiguity in several ways: Assume $z \in \mathfrak{X}_1 \setminus \mathfrak{X}_*$ with $x^{m,n} \prec z$ violating ④.

The term $x \ldots x$ violates ①, if x was nullable. Though this is not severe — since then $x^{m,n} = x^{0,n}$ and that term can be omitted — the fact that $x^{m,n}$ violates ④ in any case implies a violation of ② by $(\epsilon|x) \ldots (\epsilon|x)$.

Clearly $(\epsilon|x) \ldots (\epsilon|x)$ violates ①. Even a more sophisticated translation of $x^{0,n}$, for example into $\epsilon|x\big(\epsilon|x\big(\epsilon|x\big(\ldots\big)\big)\big)$ yields a violation of ①, if $x \in \mathcal{E}$.

Further research Try to determine if the unfolding of $x^{m,n}$ introduces only certain ambiguities, which might be removed more easily. Ambiguities in general cannot be removed [2].

5.6 Real Life Tests

The Pathfinder working group [10] has already built an implementation of this algorithm in C. The results of first benchmark tests [7] using XMark [13] are satisfying: The XMark DTD was translated into 74 regular expressions, and document instances with the number of nodes ranging from $3\,000$ to $3\,000\,000$ have been validated against it. A validation time, scaling linearly with the amount of nodes in

the document instance, was observed. The implementation hosted on a 2.4 Linux PC, equipped with a $2.2\,\mathrm{GHz}$ processor and $2\,\mathrm{GB}$ RAM was able to validate a $110\,\mathrm{MB}$ document within less than 5 seconds.

5.7 Summary

In this chapter we have seen that derivation can process a document in linear time, if validating against a "well behaved" grammar. It turned out, however, that the XML Schema standard, constraining its grammars to one-unambiguity, is to weak to ensure that property. The star normal form, introduced by [3], was adopted to the context of this work. It was proven that one-unambiguity and star normal form together ensure linear runtime for the proposed algorithm.

However, the question if, and how, one-unambiguous grammars that are not in star normal form, but employ the ^{m,n} operator, can be transformed to be in SNF, must be deferred to further research.

Chapter 6

Implementation

This diploma thesis comes with an implementation of the proposed algorithm, which, however, does not cover the whole subset of XML that can be expressed by context free grammars. For example, it still lacks the validation of All groups or text nodes. Appendix B explains how to retrieve, compile and use the implementation.

6.1 The Toolbox

The first implementation was done in Haskell [8], a purely functional programming language. The excellence of Haskell in prototyping concise formalized algorithms, as well as its algebraic datatypes and list processing capabilities, have been crucial for its choice. So it was almost a job of simply converting the mathematical definitions into Haskell syntax, and using the Glasgow Haskell Compiler [4] as an evaluation system.

Although it was easy to verify the process of derivation and type annotation using the Haskell approach, this had one severe drawback: The evaluation speed of the generated code was not adequate. So another implementation [7] in C has been written, however not within the scope of this diploma thesis. That second implementation shows the full strength of the algorithm when used with real world examples. See Section 5.6.

The following discussion describes the Haskell implementation of the algorithm. For a definition of the Haskell programming language, see [8].

6.2 Datatypes

The derivation algorithm aims at relationally encoded tree structures. However, for testing and evaluation, we start from forests which can be imported directly from an XML source.

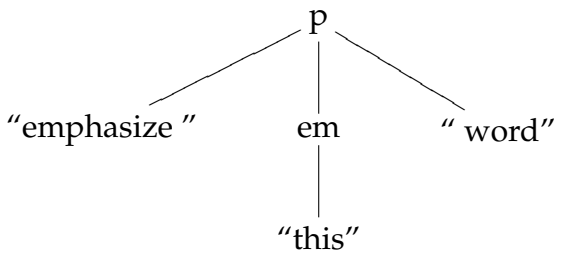
The Forests are defined straightforward. Note the two type constructors `Node` and `Text` used to implement a distinction between the XML nodes that represent element nodes and those that represent text nodes:

```
data Forest = Empty
            | Node String Forest Forest   -- element node
            | Text String Forest          -- text node
            deriving Eq
```

The first `Forest` passed to a `Node` constructor represents the descendants, the second one its followings. Note that text nodes cannot have children. Membership in the `Eq` class implements the recursive definition of equality between forests, as given in Definition 2.2.6.

As an example, let us have a look at the following XML snippet and its associated tree:

```
<p>
  emphasize
  <em>this</em>
  word
</p>
```



This is encoded in Haskell as follows:

```
Node "p"
     (Text "emphasize "          -- first child of p
           (Node "em"            -- em follows ”emphasize ”
                 (Text "this"    -- first child of em
                       Empty     -- last child ”this”
                 )
                 (Text " word"   -- ” word” follows em
                       Empty     -- last child ” word”
     )))
     Empty                       -- nothing follows p
```

The Pre/Post Enumeration again shows the distinction of element nodes and text nodes: The nodes of the forest are encoded as `Elements` by:

```
data Element = Tag String      -- element node
             | Value String    -- text node
             deriving Eq
```

With `Inti` being an extension of the `Int` datatype, supporting ∞ and $-\infty$, the enumeration for of a forest is typed `PpnSet`:

```
type PpnTupel = (Inti,Element,Inti)
type PpnSet = [PpnTupel]
```

The capability of representing ∞ is not necessary for the nodes themselves, however the guards will utilize infinity, and their values will be compared to the ones used in the enumeration.

The Regular Expressions as defined in $\mathfrak{X}$ are encoded in Haskell by the `FRex` datatype. The comments give hints about the constructors' intended purpose.

```
data FRex = NoForest                -- ∅
          | Epsilon                 -- ϵ
          | Con FRex FRex           -- xy
          | Alt FRex FRex           -- x|y
          | Rep Inti Inti FRex      -- x^{m,n}
          | Tree String FRex        -- foo⟨x⟩
          | Element String String   -- foo⟨t⟩
          | Group String            -- t
          | CData String            -- ".*"
                                    --  b
          | Guarded Limit FRex      -- [x]
                                    --  a
          deriving Eq
```

where the `Limit`s are simply pairs of `Inti` values. Nodes representing element nodes, named groups and types are referred to by `String` names. The *`CData`* constructor takes a string argument whose intention is to describe the requested text node by means of a regular expression (see *Text Nodes* on page 40).

Recall, that the three XML Schema concepts of named complex types, named groups, and anonymous complex types are mapped to nonterminals and subtree constructions (see *Translating the Schema* on page 38). This is why

- *`Element`* uses subtree construction as well as a nonterminal referring to the named type,
- *`Group`* uses a nonterminal referring to the named group only, and
- *`Tree`* simply creates a subtree structure for the definition of the content model.

Also note that the implementation of $\mathfrak{X}$ already provides an implementation of $\mathfrak{F}$ which is a subset of $\mathfrak{X}$.

For convenience, we will also define functions that simplify the use of the *`Con`* and *`Alt`* constructors. They will be used later in source code examples:

```
-- Con frontends
(<.>) :: FRex -> FRex -> FRex
a <.> b = Con a b

sequen :: [FRex] -> FRex
sequen = foldl Con Epsilon


-- Alt frontends
(<|>) :: FRex -> FRex -> FRex
a <|> b = Alt a b

choice :: [FRex] -> FRex
choice = foldl Alt NoForest
```

Grammars A grammar is implemented as lookup table which contains the rules of a CFG, i.e., the name of what is defined on the left hand side, and a regular expression on the right hand side:

```
type Table = [(Def,FRex)]
```

XML Schema provides two different kinds of objects that can be "looked up" by their string identifier: *Groups* implementing a simple macro facility which allows the reuse of schema descriptions, and *types* which describe the contents (i.e., children) of a node. This distinction is also made in the implementation by annotating the left hand side of the CFG productions according to their intended use:

```
data Def = GroupDef String
         | TypeDef String
         deriving Eq
```

Note that these reside in two different namespaces, i.e., they cannot be interchanged. Membership in the `Eq` class thus implements the correct equality.

A start group, whose name is purely convention and must not conflict with any group name used in the schema description, is used as the starting point for the derivation process.

The Type Annotations will be collected in a table of type `Annot`, defined by

```
type Annot = [(Inti,Element,String)]
```

which depicts a relation between the pre number of the node and the string referring to its annotated type. The `Element` is stored only to enhance readability; the pre number to node mapping is already present in the pre/post enumeration.

6.3 Translating XML Schema into Haskell

With the given datatypes it is now possible to describe most parts of an XML document, XML Schema, and the validation process. As an example, let us have a look at the *Purchase Order* example already used above.

Starting derivation with the CFG left hand side group START, the possible root nodes of the document are translated into:

```
(GroupDef "Start",
 choice [
         (Element "purchaseOrder" "PurchaseOrderType"),
         (Element "comment" "xsd:string")
        ]
)
```

A named complex type definition, utilizing sequence, choice, and named group constructs, looks as follows:

```
(TypeDef "PurchaseOrderType",
 sequen [
         (Element "@orderDate" "xsd:date")
         choice [
                 (Group "ShipAndBill"),
                 (Element "singleUSAddress" "USAddress")
                ],
         (Rep 0 1 (Element "comment" "xsd:string")),
         (Element "items" "Items"),
        ]
)

(GroupDef "ShipAndBill",
 sequen [ ... ]
)
```

Anonymous types are also supported, as can be seen in the definition of ITEMS.

```
(TypeDef "Items",
 (Rep 0 Inf (Tree "item" (
    sequen [
            (Element "@partNum" "SKU")
            (Element "productName" "xsd:string"),
            (Element "quantity" "xsd:positiveInteger"),
            ...
           ]
                       )))
)
```

Note that attributes are simply considered being children of a node, prefixed with an @ character (see *The All Group* on page 35 for a discussion of the order of attributes).

6.4 Importing XML Data

For testing, XML documents are imported into the Haskell runtime system using the XML parser facilities provided by the HaXml modules, that come with the Glasgow Haskell Compiler [4]. Conversion routines map the gathered data structures to the `Forest` structure depicted above, stripping away some XML concepts that are not supported by this implementation, e.g., processing instructions. Recall that the import of XML documents and their enumeration are not to be performed in the intended environment of this algorithm, since the idea is to work on relationally encoded data without building the trees.

6.5 Enumerate

The pre/post enumeration is not used by the actual algorithm. It is required only for testing, to encode the XML forests imported to the Haskell runtime system. Its definition in Haskell is a bit different from Definition 2.5.1 to avoid searching the maximum in a set, but the recursive structure remains the same.

The `phi` function (i.e., φ from above) enumerates a given forest according to DFS.

```
phi :: Inti -> Inti -> Forest -> PpnSet -> (Inti,Inti,PpnSet)
```

It takes two `Inti` arguments which denote the smallest pre/post numbers to use, and a accumulating parameter `lst` which will collect the enumeration. It returns the pre/post numbers to use for further enumeration of successive trees, as well as the enumeration of the given forest.

The recursive structure of the enumeration is anchored at *`Empty`*:

```
phi p q Empty lst          = (p,q,lst)
```

The empty forest does not add nodes to the enumeration. Also, it does not increase the pre/post counters.

The enumeration of a node with potential children and successors is as follows:

```
phi p q (Node a cf sf) lst = (sp,sq,(p,Tag a,cq):se) -- current
  where (cp,cq,ce) = (phi (p+1) q cf lst)            -- children
        (sp,sq,se) = (phi cp (cq+1) sf ce)           -- successives
```

First, the child forest `cf` is processed, beginning with a pre value of `p+1` (since `p` is the pre number of the node itself), and a post value of `q` (since the current node has not been processed completely by DFS). This results in the `(cp,cq,ce)` tuple, where `ce` is the enumeration of the child forest, added to the enumeration given in `lst`, `cq` is the post value of the current node, since now all of its children have been processed, and `cp` is the new pre number to be used for enumeration of successive forests. Then the enumeration `se` of the successive forest `sf` is calculated, beginning with the pre values `cp`, the post value `cq+1`, and adding its

results to the enumeration `ce`. Finally the current node `a` itself is added. The whole enumeration as well as the two indices `sp` and `sq` are returned.

The enumeration of text nodes is similar, differing only in the absence of possible children:

```
phi p q (Text a sf) lst    = (sp,sq,(p,Value a,q):se)
  where (sp,sq,se) = (phi (p+1) (q+1) sf lst)
```

With this, the normalized pre/post enumeration `ppn` of a forest `f` is defined as

```
ppn :: Forest -> PpnSet
ppn f = x
    where (_,_,x) = phi 0 0 f []
```

where the pre/post values to use for further enumeration are discarded.

6.6 Validate

The nullable test is straightforward from Definition 3.1.4. For a given grammar, it checks if a `FRex` expression is in $\mathcal{E}$. Therefore it requires access to the lookup table induced by the grammar.

```
nlb :: FRex -> Table -> Bool
nlb (Epsilon)       _ = True
nlb (Con x y)     tab = nlb x tab && nlb y tab
nlb (Alt x y)     tab = nlb x tab || nlb y tab
nlb (Rep m n x)   tab | m==0        = True
                      | otherwise   = nlb x tab
nlb (Guarded _ x) tab = nlb x tab
nlb (Group n)     tab = nlb (resolve (GroupDef n) tab)
                            (remRecursion (GroupDef n) tab)
nlb _ _ = False
```

Where the `remRecursion` function alters the given definition in the table to `NoForest` to avoid infinite loops that might occur within group definitions like

```
(GroupDef "A", Group "A" <|> Group "B")
```

and a nullable test of `nlb (Group "A")` which is assumed to fail if group `B` is not nullable. This is relevant only for testing, since XML Schema prohibits such grammars (see Appendix A.2).

The core function is the derivation `der`, implementing the derivation according to a single node.

It consumes one node, a regular expression describing the potential remains of the document, a table of CFG rules, as well as a table of already collected type

annotations. It returns the transformed regular expression and the potentially enriched table of type annotations.

$$\partial_{p,a,q} : \mathfrak{X} \longrightarrow \mathfrak{X}$$

```
der :: PpnTupel -> FRex -> Table -> Annot -> (FRex,Annot)
```

The implementation directly follows Definition 3.2.1 of the derivation according to a single node.

▷ **Case I** The both cases where nothing is left that might be matched:

$$\oslash \longmapsto \oslash \quad \text{and} \quad \epsilon \longmapsto \oslash$$

```
der _ (NoForest) _ ta = (NoForest,ta)
der _ (Epsilon)  _ ta = (NoForest,ta)
```

▷ **Case II** Deriving an expression protected by guards:

$$\begin{matrix}\overline{p} & \overline{q} \\ [x] \\ \underline{p} & \underline{q}\end{matrix} \longmapsto \begin{cases} \begin{matrix}\overline{p} \\ [\partial_{p,a,q} x] \\ \underline{p}\end{matrix}^{\overline{q}}_{\underline{q}} & \text{if } \underline{p} < p < \overline{p} \wedge \underline{q} < q < \overline{q} \\ \oslash & \text{otherwise} \end{cases}$$

```
der t@(p,a,q) (Guarded l x) tab ta
                    | x'==NoForest = (NoForest,tb)
                    | otherwise = if contains l q
                                  then (guard l x',tb)
                                  else (NoForest,tb)
    where (x',tb) = der t x tab ta
```

Here, the function `contains` checks whether the post value is within the allowed range. The `guard` function protects the given expression with the given guards. For optimization reasons, this means modification of guards — if present — rather than always generating a new guard structure (see *Collapse Guards* on page 32).

▷ **Case III** Derivation of a hierarchical structure which does not use type or group lookups:

$$b\langle x \rangle \longmapsto \begin{cases} [x]^{q}_{p} & \text{if } a = b \\ \oslash & \text{otherwise} \end{cases}$$

```
der t@(_,Tag a,_) (Tree b x) tab ta | a==b = (chldOf t x,ta)
                                    | otherwise = (NoForest,ta)
der t@(_,_,_) (Tree _ _) tab  ta = (NoForest,ta)
```

The *Tag* assures that the first rule is not applied to text nodes. The expression returned if $a = b$ is protected with guards by the `chldOf` function, enforcing all following nodes to be children of the passed node.

▷ **Case IV** Derivation of a sequence:

$$xy \longmapsto \begin{cases} \partial_{p,a,q}x\underset{p}{[}y\underset{q}{]}|\partial_{p,a,q}y & \text{if } x \in \mathcal{E} \\ \partial_{p,a,q}x\underset{p}{[}y\underset{q}{]} & \text{otherwise} \end{cases}$$

```
der t (Con x y) tab ta
              | x==NoForest || y==NoForest = (NoForest,ta)
              | nlb x tab = (
                             ( x' <.> (succOf t y) ) <|> y',
                             tc
                            )
              | otherwise = (
                             x' <.> (succOf t y),
                             tc
                            )
             where (x',tb) = der t x tab ta
                   (y',tc) = der t y tab tb
```

The function `succOf` protects the given regular expression with guards, enforcing all following nodes to be successives of the given one.

▷ **Case V** Derivation of a choice:

$$x|y \longmapsto \partial_{p,a,q}x|\partial_{p,a,q}y$$

```
der t (Alt x y) tab ta | x'==NoForest = (y',tc)
                       | y'==NoForest = (x',tc)
                       | otherwise = (x' <|> y' ,tc)
       where (x',tb) = der t x tab ta
             (y',tc) = der t y tab tb
```

▷ **Case VI** Derivation of an occurrence constraint:

$$x^{m,n} \longmapsto \begin{cases} \partial_{p,a,q}x & \text{if } n = 1 \\ \partial_{p,a,q}x\underset{p}{[}x^{0,n-1}\underset{q}{]} & \text{if } n > 1 \wedge m = 0 \\ \partial_{p,a,q}x\underset{p}{[}x^{m-1,n-1}\underset{q}{]} & \text{otherwise, i.e., } n > 1 \wedge m > 0 \end{cases}$$

```
der t (Rep m n x) tab ta
     | x'==NoForest = (NoForest,tb)
     | m>0 && n>1  = (x' <.> (succOf t (Rep (m-1) (n-1) x)),tb)
     | n==1        = (x',ta)
     | m==0 && n>1 = (x' <.> (succOf t (Rep 0 (n-1) x)),tb)
     | otherwise   = error ("unexpected repetition values")
     where (x',tb) = der t x tab ta
```

▷ **Case VII** Derivation of a nonterminal:

$$\mathfrak{t} \longmapsto \partial_{p,a,q}(\lambda\mathfrak{t})$$

In order to support XML Schema's named groups as well as its types, we need to distinguish two kinds of nonterminals, as shown in *Translating the Schema* on page 38.

▷ **Case VII.a** Resolving a named group:

```
der t (Group ty) tab ta =
      der t (resolve (GroupDef ty) tab) tab ta
```

If the nonterminal refers to a group, it is replaced by that group's definition by the `resolve` function, and derivation carries on with the replacement.

▷ **Case VII.b** Resolving a named type:

```
der t@(p,Tag a,_) (Element b ty) tab ta
      | a==b = (
                chldOf t (resolve (TypeDef ty) tab),
                (add ta (p,Tag a,ty))
               )
      | otherwise = (NoForest,ta)
```

If derivation is according to the expected XML element node, then the returned expression is the content model definition of the named type, returned by the `resolve` function. Again, the `chldOf` function surrounds the given expression with guards, enforcing the following nodes to be children of the node given in `t`. Additionally, the type annotation table `ta` is enriched with the gained type information by the `add` function.

The `add` function is used to avoid that type information is added more than once. This might happen for grammars that do not possess the star normal form (see Section 5.4).

▷ **Case VIII** Text nodes are not represented in the calculus. Instead, they are handled like leaf element nodes in the document forest. However, since validation of text nodes is not implemented yet, it must be handled specially by the implementation.

```
der (p,Value a,_) (CData regex) _ ta
          | regexMatch regex a = (Epsilon,
                                  (add ta (p,Value a,regex)))
          | otherwise = (NoForest,ta)
          where regexMatch _ _ = True
der (p,Value a,_) _ _ ta = (NoForest,ta)
der (_,_,_) (CData _) _ ta = (NoForest,ta)
```

The `regexMatch` function currently always succeeds, however should be replaced by a function that really validates the given string against the given regular expression.

Note that text node validation validation should not call for changes to the calculus. See *Text Nodes* on page 40 for a discussion.

Appendix A

XML Schema Constraints

The calculus of forests and languages developed in this thesis is quite powerful. Sometimes too powerful: It permits the generation of documents of polymorphic type (see *Polymorphism* on page 41), or the definition of grammars that might run into infinite application of the lookup function λ (see page 15). However XML Schema — which potentially suffers the same complexity — is restricted by a set of constraints. On the one hand, these strip away some of XML Schema's expressiveness, on the other, they make it significantly easier to handle the grammars and documents.

The following is a summary of the XML Schema Constraints referenced in this thesis:

A.1 Schema Component Constraint: Model Group Correct

> [...] *Circular groups are disallowed. That is, within the particles of a group there must not be at any depth a particle whose term is the group itself.*
>
> [16] Section 3.8.6

This forbids recursive groups like $\mathfrak{s} ::= a\langle\mathfrak{s}|\epsilon\rangle$ and thus omits infinite loops while resolving a named group.

A.2 Schema Component Constraint: Unique Particle Attribution

> *A content model must be formed such that during validation of an element information item sequence, the particle* [...] *therein with which to attempt to validate each item in the sequence in turn can be uniquely determined without examining the content or attributes of that item, and without any information about the items in the remainder of the sequence.*
>
> [16] Section 3.8.6

This constraint, which is also known as "one-unambiguity", forbids the declaration of content models as implied by $\mathfrak{s} ::= ab|ac$.

A.3 Schema Component Constraint: All Group Limited

> [...] *The max occurs of all the particles in the particles of the group must be 0 or 1.*
>
> [16] Section 3.8.6

A.4 More about the All Group

> *The All group* [...] *is limited to the top-level of any content model. Moreover, the group's children must all be individual elements (no groups), and no element in the content model may appear more than once* [...]
>
> [15] Section 2.7

The last two quotes state that the All group in fact offers some kind of set semantics: Derivation against an element in that group is almost equivalent to removing that element from the set.

Appendix B

The Accompanying Software

This diploma thesis comes with an implementation of the proposed algorithm, written in the functional language Haskell, as described in Chapter 6.

The implementation is contained on the enclosed CD, and can be retrieved online through the Konstanzer Online-Publikations-System (KOPS), using the permanent direct URL

http://www.ub.uni-konstanz.de/kops/volltexte/2004/1234/

For compilation, you will need the Glasgow Haskell Compiler [4]. The software has been developed with version 5.04, which already supplied the HaXml libraries required for XML parsing. With this and GNU *make* compilation is as easy as typing

```
$ make
```

where `$` denotes the shell prompt as usual. Three validators are built by make: `purchaseorder`, `exponential`, and `nosnf`.

The command line syntax is VALIDATOR (`-g` |(`-t` ^{0,1} FILENAME))^{0,1}. Without parameters, the validator will display a short help message. The `-g` option prints the implemented grammar and exits. The name of an XML file alone will cause the validator to validate the file, and return either a type annotation table, or a `not valid` message. If the `-t` option is given, a complete validation trace will be printed.

When tracing, for each step of the derivation the validator prints

1. the remaining enumeration of the document in reverse pre order,
2. the regular expression to be derived according to the first node in the remaining enumeration during the next step,
3. whether this expression is nullable, and
4. the type annotations gathered so far, also in reverse pre order.

The reverse order is used so that you can see the regular expression, the next node to be used, and the last type annotation generated, at the same time without scrolling through the potentially large document.

Purchase Order Example The `purchaseorder` validator implements the Purchase Order schema given in Section 4.2. Table I on page 44 and Table II on page 48 can be generated by

```
$ purchaseorder -t purchaseorder.xml
```

which validates the file `purchaseorder.xml`, also present in the source directory, against the Purchase Order schema.

Exponential Growth Example The `exponential` validator implements the (not XML conforming) regular expression given on page 51. The exponential growth of a regular expression that lacks one-unambiguity, can be observed by:

```
$ exponential -t exponential.xml
```

You may want to edit the file `exponential.xml`, which currently contains a sequence of only few `<a/>` elements.

Missing SNF Example The `nosnf` validator implements the XML conforming regular expression given on page 54 that lacks the star normal form. By running

```
$ nosnf nosnf.xml
```

it becomes clearly visible how the regular expression grows to cover the introduced ambiguities.

Currently, type annotations are created for a node only if not already present. This hides ambiguities as discussed in Section 5.4. You may want to edit the `add` function in file `FRex.hs`, where marked with a line

```
{- *** change below *** -}
```

to always add the type annotation. Recompilation and running

```
$ make nosnf
$ nosnf nosnf.xml
```

shows that multiple but identical type annotations are created.

Your own Validator To build your own validator `foo`, proceed as follows:

1. Translate the desired grammar into Haskell syntax, using the provided data structures as discussed in *Translating the Schema* on page 38 and Section 6.3.

2. Take a copy of one of the example implementations, e.g., `nosnf.hs`, and save it as `foo.hs`.

3. Edit the file `foo.hs`. Change the definition of `grammarName` to `foo`:

   ```
   grammarName = "foo"
   ```

   Change the definition of `grammar` to represent the grammar you have built:

   ```
   grammar = Grammar [
                       (GroupDef "START", ...),
                       ...
                     ]
   ```

   The grammar must contain a `GroupDef "START"` which is used as a starting point for validation.

4. Edit the `makefile` to compile your grammar. Add `foo` to the list of validators:

   ```
   validators = purchaseorder exponential nosnf foo
   ```

5. Build the validator.

   ```
   $ make foo
   ```

6. Enjoy!

   ```
   $ foo
   ```

   prints a short help message,

   ```
   $ foo -g
   ```

   displays the implemented grammar, and

   ```
   $ foo bar.xml
   ```

   validates the file `bar.xml` against the implemented grammar, returning a type annotation table or "`not valid`". If the `-t` option is also given, a complete derivation trace will be printed.

Symbol Reference List

The comments in this list are intended as a reminder only, rather than giving a mathematically correct definition.

Greek Letters

ϵ The empty word or forest. Definition 2.1.1, 2.2.1.

λ Lookup function for nonterminals in CFG. Definition 2.4.1.

$\varphi, \varphi_{p,q}$ (Normalized) pre/post enumerations. Definition 2.5.1.

σ Size of a forest, a regular expression or a grammar. Definition 2.2.8, 5.1.2, 5.1.3.

τ_M First node in an enumeration according to pre order values. Notation 2.5.2.

Latin Letters

$\mathbf{L}$ Set of valid values for the guard construct. Definition 3.1.1.

$M_<, M_>$ Enumerations representing the descendant and following forest with respect to τ_M. Definition 2.5.7.

$\underline{p}, \overline{p}$ Lower/upper limit for pre/post values, specified in a guard. Definition 3.1.1.

$\mathbf{R}$ Set of valid values for the repetition construct. Definition 2.1.7.

V, V' Set of nodes (aka. vertices) and marked nodes. Section 2.2, 5.2.2.

Script Letters

$\mathcal{A}, \mathcal{B}$ Alphabet of a language, and extended alphabet for regular expressions to build such a language. Definition 2.1.1, 2.1.10, 2.3.1

$\mathcal{E}$ Set of nullable expressions. Definition 3.1.4.

$\mathcal{F}$ The set of forests. Definition 2.2.3.

$\mathcal{L}_\mathcal{A}$ Set of languages over $\mathcal{A}$. Definition 2.1.3.

$\mathcal{N}, \mathcal{N}_0$ Set of (normalized) pre/post enumerations. Definition 2.5.5.

$\mathcal{O}$ Upper bound, according to Landau notation.

$\mathcal{P}$ The usual power set. For any set A, $\mathcal{P}A$ is also known as 2^A.

Gothic Letters

$\mathfrak{F}$ Regular expressions for forests. Definition 2.3.3.

$\mathfrak{R}$ Regular expressions for non-tree languages. Definition 2.1.13.

$\mathfrak{t}, \mathfrak{s}$ Names for nonterminals. Definition 2.4.1.

$\mathfrak{X}, \mathfrak{X}'$ Regular expressions for forests, enriched with guards, and marked regular expressions. Definition 3.1.1, 5.2.3.

$\mathfrak{X}_1$ one-unambiguous expressions in $\mathfrak{X}$. Definition 5.2.9, Theorem 5.3.1.

$\mathfrak{X}_*$ Expressions in $\mathfrak{X}$ that possess the star normal form. Definition 5.3.4.

Symbols Ordered by arity.

$\oslash$ Nothing. Regular expression that cannot be matched. Definition 2.3.2.

$\mathbb{N}$ The usual natural numbers. Throughout this thesis, always assume $0 \in \mathbb{N}$.

* $\mathcal{A}^*$ denotes the set of words over $\mathcal{A}$, Definition 2.1.1. For a regular expression x, $x^* = x^{0,\infty}$ is Kleene's closure.

$\partial_\alpha, \partial_N, \partial_f$ Derivation according to a node, an enumeration, and a forest. Definition 3.2.3.

^{m,n} Repetition, an extension of Kleene's closure. Definition 2.1.7.

$\grave{}, \check{}, \acute{}$ Refer to pre order value, node, or post order value. Notation 2.5.2.

$<, >$ see $M_<, M_>$ in *Latin Letters* on page 85.

$|$ Alternation. Definition 2.1.4, 2.3.2.

$\cdot$ Concatenation. Definition 2.1.2, 2.1.4, 2.2.2, 2.3.2.

$\langle\rangle$ Subtree construction. Definition 2.2.2, 2.3.2.

$\vDash, \nvDash$ A word/forest either matches, or does not match a regular expression. Definition 2.1.14, 2.3.4.

$\sim$ Similarity between enumerations. Definition 2.5.9.

$\overset{b}{\underset{a}{[}}x\overset{d}{\underset{c}{]}}$ Guards restricting matching of a regular expression x with respect to the pre/post enumeration. Definition 3.1.1.

Special Typeset

name Name of a terminal imported from an XML Schema description. *Translating the Schema* on page 38.

NAME Name of a nonterminal imported from an XML Schema description. *Translating the Schema* on page 38.

Others

:: Used to distinguish named types from named groups. *Translating the Schema* on page 38.

@ Prefix to distinguish XML attribute nodes from element nodes. *Translating the Schema* on page 38.

first Nodes that may appear at first in a matching forest. Definition 5.2.5.

follow_c Nodes that may follow c in a matching forest. Definition 5.2.6.

$\mathrm{followlast}$ Nodes that may follow any of the last nodes in a matching forest. Definition 5.2.8.

id The usual identity ($\mathrm{id}\, x = x$).

inter Intermediate function, used for transformation of a regular expression into SNF. Definition 5.5.1.

last Nodes that may appear at last in a matching forest. Definition 5.2.7.

snf Function intended to transform a regular expression into SNF. Definition 5.5.1.

Index

Bold numbers refer to definitions. Small typeset numbers indicate the section, whereas normal sized numbers indicate the page.

Bibliography

[1] Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[2] Anne Brüggemann-Klein, D. Wood. One-Unambiguous Regular Languages. *Information and Computation*, 142(2):182–206, 1998.

[3] Anne Brüggemann-Klein. Regular Expressions into Finite Automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[4] The Glasgow Haskell Compiler. Website.
http://www.haskell.org/ghc/index.html

[5] Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the 21st International ACM SIGMOD Conference on Management of Data*, page 109–120, Madison, Wisconsin, USA, June 2002.

[6] Torsten Grust, Maurice van Keulen, Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proceedings of the 29th Conference on Very Large Databases (VLDB)*, Berlin, Germany, September 2003.

[7] Torsten Grust, Stefan Klinger. Schema Validation and Type Annotation for Encoded Trees. Submitted, March 2004.

[8] Haskell – A Purely Functional Language. Website, 22 Mar 2004.
http://www.haskell.org/

[9] MonetDB – Query Processing at Light-Speed. Website, 2004.
http://monetdb.cwi.nl/

[10] The Pathfinder Project. Website, Jan 2003.
http://www.inf.uni-konstanz.de/dbis/research/pathfinder/

[11] Comprehensive Perl Archive Network. Website, 25 Mar 2004.
http://www.cpan.org/

[12] Unicode Home Page. Website, 2004.
http://www.unicode.org/

[13] XMark – An XML Benchmark Project. Website, 28 Jun 2003.
http://www.xml-benchmark.org/

[14] John Cowan, Richard Tobin. XML Information Set. Recommendation, W3C, October 2001.
http://www.w3.org/TR/xml-infoset/

[15] David C. Fallside. XML Schema Part 0: Primer. Recommendation, W3C, May 2001.
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

[16] David Beech, et al. XML Schema Part 1: Structures. Recommendation, W3C, May 2001.
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[17] Paul V. Biron, Ashok Malhotra. XML Schema Part 2: Datatypes. Recommendation, W3C, May 2001.
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[18] Mary Fernández, et al. XQuery 1.0 and XPath 2.0 Data Model. Working draft, W3C, November 2003.
http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/

The compact disc included on this page of the printed version, contains the implementation of the proposed algorithm. See Chapter 6 and Appendix B for further information.