

University of Konstanz
Faculty of Computer and Information Science
Chair, Prof. Dr. Scholl
WS 2003/04

Stefan Alexander Hohenadel

**Subtyping for Regular Tree Types:
A Java-based Implementation**

Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Information Engineering

University of Konstanz
October 2003

Contents

1	Preliminaries	5
1.1	Organization of Thesis	5
1.2	The Subtyping Concept	6
1.3	Type Representation by Regular Expressions	6
1.4	Subtyping Using Finite Automata	9
1.5	Subtyping Using Antimirov’s Calculus	11
2	The Derivation of Regular Expressions	13
2.1	Definition of the Term Algebra	13
2.2	Partial Derivatives of Regular Expressions	14
2.3	Examples for the Derivation of Regular Expressions	15
2.4	Partial Derivatives of Regular Inequalities	17
2.5	Rewriting Calculus for Regular Expressions	18
3	Extending Antimirov’s Calculus to a Subtyping Algorithm	21
3.1	The Check for ε -Inclusion	21
3.2	Leading Names	22
3.3	Two Wellformedness Constraints for Types	24
3.4	Checking Wellformedness of Types	27
3.5	Partial Derivatives of Regular Expressions Redefined	30
3.6	A Simplification for Partial Derivatives of Regular Inequalities	31
3.7	Trivial Cases	33
3.8	From a Calculus to an Algorithm	34
3.9	Completion of the Running Example	36
4	JAVA Class Design	39
4.1	The Hierarchy of Types	39
4.2	Names	42
4.3	Sets	42
4.4	Type Pairs	44
4.5	Regular Inequalities	44
4.6	Exceptions	45
5	Implementation of Auxiliary Functions	47
5.1	Check for Wellformedness	47
5.2	Unfolding Recursive Types	48
5.3	Function <code>nullable()</code>	50

5.4	Function <code>leadingNames()</code>	51
5.5	Function <code>content()</code>	52
5.6	Concatenation of Linear Forms	52
5.7	Partial Derivatives of Types	54
6	Implementation of the Subtyping Algorithm	57
6.1	Partial Derivatives of Regular Inequalities	57
6.2	Power Set Computation	58
6.3	Type Construction	61
6.4	Trivial Case Checking	63
6.5	Implementation of Method <code>prove()</code>	64
7	Related and Future Work	67
7.1	XML Schema Type Import	67
7.2	Normalization of non-wellformed types	67
7.3	Non-empty Intersections of Types	68
A	Logfile for Running Example	69

Chapter 1

Preliminaries

1.1 Organization of Thesis

This thesis presents the description of a subtyping algorithm and its implementation using the JAVA programming language. Subtyping algorithms are typically used in the context of compiler construction. This work originated in the Pathfinder project, a working group for the construction of a compiler for the XQuery language at the University of Konstanz. This subtyping algorithm is designed to be applied to the type system of XQuery and XML Schema, as described in [XQL03], [XQFS03] and [XSD01].

Pathfinder makes use of the subtyping algorithm described in this thesis. An earlier version of the algorithm was first invented by Martin Kempa and Volker Linnemann in the XOBÉ project at Lübeck University (cf. [KeLi03]). The algorithm adapts a derivation calculus for regular expressions to the task of subtyping. This calculus was conceived by Valentin Antimirov (1961-1995) in [Ant94] and [AntMos95]. Therefore, we will also speak of “Antimirov’s Algorithm”. This presentation adds some refinements to the algorithm described in [KeLi03].

In this chapter, we will explain the concept of “subtyping” and describe how types are represented formally by the algorithm. We will also give a short description about the “classical” way of subtyping using finite automata. The chapter ends with a rough outline of the idea of Antimirov’s Algorithm as an alternative to the subtyping approach using automata.

In Chapter 2, we will describe Antimirov’s original derivation calculus.

Chapter 3 contains a detailed description of all extensions and modifications added to the calculus to extend it to a subtyping algorithm. This will include the definition of a number of auxiliary functions. The algorithm is described in pseudo-code at the end of the chapter.

In Chapter 4, we describe all JAVA classes involved in the implementation of the algorithm. We will also provide a description of the interactions between classes and describe the mapping from functions and entities to classes and methods.

Chapter 5 explains in detail the implementation of all auxiliary functions.

Chapter 6 contains a description of the implementation of the subtyping al-

gorithm. The most important aspect is how the computation of *partial derivatives of a regular inequality* works.

In Chapter 7, we will address some aspects, that at this time cannot be discussed within the context of this thesis, or which will become subject to optimization in future works.

Appendix A contains a logfile of one run of the subtyping algorithm. It is enclosed to give an example for the logging facility and the logfile design.

1.2 The Subtyping Concept

A typed language assigns types to all expression forms or constructs it provides. A language like XQuery defines built-in types and allows the definition of user defined types. The compilation of a piece of program code written in a typed language therefore requires some kind of check as to whether the use of the expressions obeys the typing rules of this language.

The two tasks connected with this requirement are *type inference* and *type checking*. Type inference means deriving the type of a simple or composite expression e , which requires the application of typing rules to e . Type checking means to check whether the use of the declared instances is coherent with the typing rules in any explicit or implicit assignment. In the following, we will concentrate on a single – but central – task concerning type checking, the so-called “subtyping”. In the context of this thesis subtyping means to check for two types r and s whether the set of instances matched by r is a subset of the set of instances matched by s . We will add a more precise version of this definition in Section 1.3. The definition of the formal presuppositions is made in Section 2.1.

The subtyping facility is an important feature because it is a central part of the type checking. A typed language cannot be compiled without type-checking. The translation process from XQuery to the XQuery core language (cf. [XQFS03]) makes excessive use of `typeswitch` expressions. Without any subtyping mechanism, we would not be able to efficiently compile expressions like `typeswitch`. Furthermore, in any language runtime system, where explicit type checking is implemented (e.g., represented by the `typeswitch` command), an implementation of subtyping must be available, too.

1.3 Type Representation by Regular Expressions

To talk precisely about subtyping we first have to make precise, what *types* are. In the context of this work, types are represented by regular expressions.

The alphabetic entities in regular expressions are based on an *alphabet* \mathcal{A} . An alphabet is a set of symbols $\alpha \in \mathcal{A}$. These symbols are called *letters*.

Regular expressions define *languages*. A language is a set of sequences of letters. We denote the language of a regular expression e as $L(e)$. In the context of this work, languages denote sets of types. The regular expression

$$int \cdot (str \{2, 5\})$$

denotes all types formed of an occurrence of the symbol *int* followed by at least 2 and at most 5 occurrences of the symbol *str*. A detailed description of regular expressions will follow in Section 2.1.

If a sequence of letters σ is element of the language of a regular expression e , we say that e *matches* σ .

Specific letters in \mathcal{A} we will consider are the type called *none* \emptyset , whose language is the empty set \emptyset and the so-called empty type ε that matches the empty sequence.

In our context, all XML Schema built-in data types in accordance to the XML Schema definition part 2 like specified in [XSD01] are letters $\alpha \in \mathcal{A}$. So we have two kinds of letters: XML node types and primitive types.

XML node types include element nodes, comment nodes, text nodes etc. Within regular expressions and definitions of functions, we will denote node types as:

$\mathbf{n}[e]$ for Element node with tag \mathbf{n} and content e
 \mathbf{n} for Element node with tag \mathbf{n} and no content

What kind of content a node can have is determined by the type of the node. The symbol \mathbf{n} denotes a name, e is a regular expression, including ε .

Following the W3C definition, also primitive types like *boolean* for boolean values and *str* for string values are contained in the definition of a letter. Primitive types are denoted like:

p for Primitive type with name p

These primitive types are the XML Schema primitive types like `xs:string`, `xs:boolean`, etc.

Regular expressions are formed by application of three regular expression constructors:

alternation: $e_1 \mid e_2$
concatenation: $e_1 \cdot e_2$
iteration: $e\{n, m\}$ with $n, m \in \mathbb{N}_0$, $n < m$

The regular expression defining a type can have a name. This is called a named type. To denote the occurrence of named types within regular expressions, we will use the colon and upper case letters:

$N : def(N)$ for Named type with name N and definition $def(N)$.

The name of a named type is used to denote its definition. Wherever the name occurs, this occurrence will be treated as if the type's definition has been seen. If the name of a named type is part of its own definition it is called a recursive type.

Below, we list the valid regular expressions and the languages they define.

$$\begin{aligned}
L(\emptyset) &\equiv \emptyset \\
L(\varepsilon) &\equiv \{\varepsilon\} \\
L(\mathbf{n}[e]) &\equiv \langle \mathbf{n} \rangle L(e) \langle /\mathbf{n} \rangle \\
L(p) &\equiv \{p\} \\
L(N : \mathit{def}(N)) &\equiv L(\mathit{def}(N)) \\
L(e_1 \cdot e_2) &\equiv \{s_1 s_2 \mid s_1 \in L(e_1) \wedge s_2 \in L(e_2)\} \\
L(e_1 \mid e_2) &\equiv L(e_1) \cup L(e_2) \\
L(e\{n, m\}) &\equiv \bigcup_{i=n}^m L(e^i)
\end{aligned}$$

Note that $e^0 = \varepsilon$ and $e^i = e \cdot e^{i-1}$.

The notation this thesis uses for iteration expressions denotes an occurrence of at least n and at most m occurrences of e connected by a concatenation. Hence the usual symbols for some iteration expressions can be redefined as follows:

$$\begin{aligned}
e^* &= e\{0, \infty\} \\
e^+ &= e\{1, \infty\} \\
e? &= e\{0, 1\}
\end{aligned}$$

Sometimes these usual symbols will be used, but the reader should keep in mind that they all instantiate iteration expressions with just different lower and upper bounds for repetition.

A type describes a set of simple values or XML tree-shaped data. Thus the regular expressions denoting the types of our type system also denote either sets of simple values or XML tree structures.

Example: Some examples for the representation of XML node types by regular expressions. The right-hand side denotes the language defined by the expression on the left-hand side.

$$\begin{aligned}
L(\mathbf{a}[\mathbf{b}]) &\equiv \{\langle \mathbf{a} \rangle \langle \mathbf{b} / \rangle \langle / \mathbf{a} \rangle\} \\
L(\mathbf{a}[\mathbf{b}] \cdot \mathbf{c}) &\equiv \{\langle \mathbf{a} \rangle \langle \mathbf{b} / \rangle \langle / \mathbf{a} \rangle \langle \mathbf{c} / \rangle\} \\
L(\mathbf{a} \cdot (\mathbf{b} \mid \mathbf{c})) &\equiv \{\langle \mathbf{a} / \rangle \langle \mathbf{b} / \rangle, \langle \mathbf{a} / \rangle \langle \mathbf{c} / \rangle\} \\
L(\mathbf{a} \cdot \mathbf{b}[\mathbf{c}?]) &\equiv \{\langle \mathbf{a} / \rangle \langle \mathbf{b} \rangle \langle / \mathbf{b} \rangle, \langle \mathbf{a} / \rangle \langle \mathbf{b} \rangle \langle \mathbf{c} / \rangle \langle / \mathbf{b} \rangle\} \\
L(\mathbf{a} \cdot \mathbf{b}[\mathbf{c}\{2, 3\}]) &\equiv \{\langle \mathbf{a} / \rangle \langle \mathbf{b} \rangle \langle \mathbf{c} / \rangle \langle \mathbf{c} / \rangle \langle / \mathbf{b} \rangle, \langle \mathbf{a} / \rangle \langle \mathbf{b} \rangle \langle \mathbf{c} / \rangle \langle \mathbf{c} / \rangle \langle \mathbf{c} / \rangle \langle / \mathbf{b} \rangle\}
\end{aligned}$$

Example: The following example shows how XML Schema types are represented by regular expressions.

Consider the following XML Schema type definition:

```

<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

<element name="title" type="xs:string"/>
<element name="author" type="xs:string"/>

<element name="paper" type="paperType"/>

<complexType name="paperType"/>
  <sequence>
    <element ref="title"/>
    <element ref="author" minOccurs="1" maxOccurs="unbounded"/>
  </sequence>
</complexType>

</schema>

```

We represent elements `title` and `author` as follows:

$$\begin{aligned} \text{title} &= \text{title}[str] \\ \text{author} &= \text{author}[str] \end{aligned}$$

Element `paper` is also an XML element node containing a single occurrence of type `paperType` where `paperType` is a non-recursive named type that consists of a concatenation of a single occurrence of `title` and a non-empty iteration of `author`.

$$\begin{aligned} \text{paper} &= \text{paper}[PaperType] \\ \text{paperType} &= PaperType : \text{title} \cdot \text{author} \{1, \infty\} \end{aligned}$$

This technique of representing XML Schema types by regular expressions allows us to operate on regular expressions and not on an XML Schema definition.

Having completed these descriptions, we can make the concept of subtyping more precise for regular expressions: The subtyping check for two regular expressions r and s means deciding whether $L(r)$ is a subset of $L(s)$. The subtype relationship between r and s is denoted by the subtyping operator $<: .$

$$r <: s \iff L(r) \subseteq L(s)$$

If $r <: s$ holds, we say “ r is a *subtype* of s ”. A clause $r <: s$ consisting of regular expressions r and s is called a *regular inequality*.

1.4 Subtyping Using Finite Automata

As types are defined by regular languages, the classical way for comparing these languages is to construct the corresponding finite automata for r and s and to compare the automata.

This technique causes a high level of complexity because it requires many manipulations of automata. An important and influential approach to language recognition by finite automata can be found in [BW98] where so-called Glushkov automata are explained and used.

A Glushkov automaton of a regular expression e with n occurrences of letters in it, is – roughly – a non-deterministic finite automaton (NFA) without ε -transitions and with n states and an additional initial state¹. The concept of Glushkov automata is based on so-called *marked* regular expressions. Approaches of generating automata from marked regular expressions were first made in [Glu61] and [MY60]. For details see there or [BW98].

A classical strategy for deciding if $r <: s$ holds can informally be stated as follows (we express the language accepted by an automaton A by $L(A)$):

1. Construct Glushkov automata G_r accepting $L(r)$ and G_s accepting $L(s)$, respectively. (The resulting automata for XQuery types will be deterministic.)
2. Construct the complementary automaton $\gamma(G_s)$ of G_s , accepting the complementary language of $L(G_s)$. (This is a quite expensive operation.)
3. Construct the automaton $\omega(G_r, \gamma(G_s))$ accepting the intersection $L(G_r) \cap L(\gamma(G_s))$ of the languages accepted by G_r and $\gamma(G_s)$. (This is a highly expensive and complex operation because the number of states of $\omega(G_r, \gamma(G_s))$ is exponential in the number of states of G_s in general.)
4. If $\omega(G_1, \gamma(G_s))$ accepts only ε , i.e., $L(\omega(G_1, \gamma(G_s))) \equiv \{\varepsilon\}$ which means that no final state of $\omega(G_1, \gamma(G_s))$ is reachable, $r <: s$ evaluates to *TRUE* otherwise $r <: s$ evaluates to *FALSE*.

Computing a complement of a deterministic finite automaton (DFA) includes making it total (if it is partial) and swap final and non-final states.

The computation of an automaton that accepts exactly the intersection of two DFAs starts with constructing the complements of the initial DFAs. Next, the union of these resulting automata is constructed. Then one has to remove all unreachable and useless states and minimise the resulting automaton (the algorithms for removing unreachable and useless states are simple but increase complexity). The final step is to construct an automaton that accepts the complement of the language accepted by the minimised automaton.

For a precise definition of Glushkov automata, marked regular expressions and the automata-based approach of subtyping see [BW98]. For general introduction to finite automata see [HU79] and [TATA].

Subtyping algorithms based on this classic technique can be found in [Hos00] and [HP02] for instance. The complexity of these operations has been our principal motivation to search for an easier subtyping technique.

¹The Glushkov automata corresponding to XQuery types are always deterministic. For details see [BW98].

1.5 Subtyping Using Antimirov’s Calculus

A more elegant and simpler way of comparing regular expressions is provided by Antimirov’s calculus: It reduces the expressions to be compared step by step using a term rewriting system. When the reduced expressions are simple enough for a trivial case check, it compares the reduced expressions by performing a cheap test. Hence, the use of automata and their associated complex manipulation operations is avoided.

Antimirov’s starting point is the observation, that for every invalid regular inequality there exists at least one reduced inequality which is *trivially inconsistent*. This basic observation is extended to a strategy for comparing regular expressions: from the initial regular inequality a set of simpler inequalities is derived. This derivation is performed until the resulting, simplified expressions match one of several trivial cases and are therefore proven or disproven.

An example for a trivial case of a regular inequality $r <: s$ is if ε is an element of $L(r)$ but not of $L(s)$. Hence, r obviously can not be a subtype of s . Such a case is called a “trivial inconsistency”: type r includes the empty type and type s does not.

To make further explanations more clear we will introduce a running example. Consider the two types:

$$\begin{aligned} r &\equiv \text{foo}[str] \cdot (\text{bar}[int] \cdot \text{foo})^* \\ s &\equiv (\text{foo}[str] \cdot \text{bar}[int])^* \cdot \text{foo}. \end{aligned}$$

In the following sections, we will use Antimirov’s technique to derive, step by step, if

$$r <: s$$

holds. Underway, we will introduce and perform all computations necessary for this.

Chapter 2

The Derivation of Regular Expressions

2.1 Definition of the Term Algebra

Most of the following definitions are taken from [Ant94]. This section describes the part of Antimirov's approach that is relevant for our context.

Given a set X , we denote its cardinality by $|X|$. The power set $\mathcal{P}(X)$ is the set of all subsets of X . The set of all *finite* subsets of X we denote by $\mathbf{Set}[X]$.

Given a finite alphabet \mathcal{A} , a word on \mathcal{A} is a sequence of letters from \mathcal{A} . We defined the concrete letters in \mathcal{A} in Section 1.3. We denote the set of all finite words on \mathcal{A} as \mathcal{A}^* .

A *regular term* t is a syntactical entity formed of letters from the alphabet \mathcal{A} and optionally structured by the standard regular operations as there are concatenation $t_1 \cdot t_2$, alternation $t_1 \mid t_2$ and iteration $t \{n, m\}$.

A *regular expression* e is a regular term that optionally contains variables denoting regular expressions. Thus the set of regular terms is a subset of the set of regular expressions. The representation of these variables in the type system are named types. Any regular expression e forms a regular language $L(e)$, which is a subset of \mathcal{A}^* .

The set $\mathbf{Reg}[\mathcal{A}]$ of regular expressions on \mathcal{A} is the least subset of the power set $\mathcal{P}(\mathcal{A}^*)$ which includes the empty set \emptyset , the empty word $\{\varepsilon\}$, the singletons $\{\alpha\}$ for all $\alpha \in \mathcal{A}$ and is closed under the standard regular operations. The set $\mathbf{Reg}[\mathcal{A}]$ together with the standard regular operations forms a *regular algebra*.

Let $\mathbf{Reg}_1[\mathcal{A}]$ be a subset of $\mathbf{Reg}[\mathcal{A}]$ consisting of all the regular expressions containing the empty word ε . Let $\mathbf{Reg}_0[\mathcal{A}]$ be the complement of $\mathbf{Reg}_1[\mathcal{A}]$ on $\mathbf{Reg}[\mathcal{A}]$, so $\mathbf{Reg}_1[\mathcal{A}] \cup \mathbf{Reg}_0[\mathcal{A}] \equiv \mathbf{Reg}[\mathcal{A}]$.

The standard interpretation of regular equations and inequalities in the regular algebra is as follows (for details see [Ant94]):

$$\mathbf{Reg}[\mathcal{A}] \models r \equiv s \quad \text{means} \quad L(r) \equiv L(s)$$

$$\mathbf{Reg}[\mathcal{A}] \models r <: s \quad \text{means} \quad L(r) \subseteq L(s)$$

Provided with this, we are able to distinguish syntactically a class of trivially inconsistent inequalities of the form $r <: s$ where $r \in \mathbf{Reg}_1[\mathcal{A}]$ and $s \in \mathbf{Reg}_0[\mathcal{A}]$.

This means the expression r matches the empty word ε and s does not, i. e.,

$$\varepsilon \in L(r) \quad \wedge \quad \varepsilon \notin L(s).$$

Therefore, if $r \in \mathbf{Reg}_1[\mathcal{A}]$ and $s \in \mathbf{Reg}_0[\mathcal{A}]$, it holds that

$$L(r) \not\subseteq L(s) \quad \iff \quad r \not\prec s.$$

If this situation occurs, the subtype check can stop immediately and return *FALSE*.

The subtyping algorithm uses a derivation calculus for regular inequalities, which derives a set of simpler inequalities from one complex inequality. The derivation process stops, when a level is reached where all inequalities are trivially *TRUE* or *FALSE*.

A single step in the derivation process is comprised of computing the so-called “partial derivatives” of a regular inequality.

2.2 Partial Derivatives of Regular Expressions

A partial derivative of a regular expression r for a given $\alpha \in \mathcal{A}$ is a representation $\partial_\alpha(r)$ of r reduced by α . The symbol $\partial_\alpha(r)$ denotes a set of regular expressions.

$$\partial_\alpha(r) := \{ e : \langle \alpha, e \rangle \in lf(r), e \neq \emptyset \}$$

Let \mathbf{SReg} be $\mathbf{Set}[\mathbf{Reg}[\mathcal{A}] \setminus \{\emptyset\}]$, then $\partial_\alpha(r) : \mathcal{A} \times \mathbf{Reg}[\mathcal{A}] \rightarrow \mathbf{SReg}$ is a function from ordered pairs consisting of a letter $\alpha \in \mathcal{A}$ and the regular expression r to be reduced by α into the set of finite sets of non-zero regular terms of \mathcal{A} .

The definition of partial derivatives of regular expressions contains a function lf . Antimirov introduces the so-called *linear form* lf of regular expressions, which defines the actual derivation of regular expressions. The linear form of a regular expression e is a reduced representation of e being “splitted” along the first occurrence of letters in e .

Function $lf : \mathbf{Reg}[\mathcal{A}] \rightarrow \mathbf{Lin}$ with $\mathbf{Lin} := \mathbf{Set}[\mathcal{A} \times \mathbf{Reg}[\mathcal{A}] \setminus \{\emptyset\}]$ is a function from the set of regular expressions into the set of ordered pairs consisting of a letter $\alpha \in \mathcal{A}$ as first element and a finite, non-empty regular expression as second element. We call lf the *linear form function*. It simplifies terms recursively as follows for all $\alpha \in \mathcal{A}$, $t, u \in \mathbf{Reg}[\mathcal{A}]$, $t_0 \in \mathbf{Reg}_0[\mathcal{A}]$ and $t_1 \in \mathbf{Reg}_1[\mathcal{A}]$:

$$lf(\emptyset) := \emptyset \tag{LF1}$$

$$lf(\varepsilon) := \emptyset \tag{LF2}$$

$$lf(\alpha) := \{\langle \alpha, \varepsilon \rangle\} \tag{LF3}$$

$$lf(N : def(n)) := lf(def(N)) \tag{LF4}$$

$$lf(t_0 \cdot u) := lf(t_0) \odot u \tag{LF5}$$

$$lf(t_1 \cdot u) := lf(t_1) \odot u \cup lf(u) \tag{LF6}$$

$$lf(t \mid u) := lf(t) \cup lf(u) \tag{LF7}$$

$$lf(t \{n, m\}) := lf(t) \odot t \{n, m\} \tag{LF8}$$

The symbols t and u denote regular expressions. The regular term \emptyset is the term matching nothing. The symbol ε denotes the empty regular term and \emptyset denotes the empty set. α denotes a letter from the alphabet \mathcal{A} .

The definition of lf involves a binary concatenation operation denoted by \odot : $\mathbf{Lin} \times \mathbf{Reg}[\mathcal{A}] \rightarrow \mathbf{Lin}$, which applies the concept of concatenation to linear forms. This operation is defined for all $l, l' \in \mathbf{Lin}$ and all $t, e \in \mathbf{Reg}[\mathcal{A}] \setminus \{\emptyset, \varepsilon\}$ as follows:

$$l \odot \emptyset := \emptyset \quad (\text{CL1})$$

$$l \odot \varepsilon := l \quad (\text{CL2})$$

$$\emptyset \odot t := \emptyset \quad (\text{CL3})$$

$$\{\langle x, \emptyset \rangle\} \odot t := \{\langle x, \emptyset \rangle\} \quad (\text{CL4})$$

$$\{\langle x, \varepsilon \rangle\} \odot t := \{\langle x, t \rangle\} \quad (\text{CL5})$$

$$\{\langle x, e \rangle\} \odot t := \{\langle x, e \cdot t \rangle\} \quad (\text{CL6})$$

$$(l \cup l') \odot t := (l \odot t) \cup (l' \odot t) \quad (\text{CL7})$$

Partial derivatives of a regular expression can be computed for a letter $\alpha \in \mathcal{A}$, a word $w \in \mathcal{A}^*$ or a set of words $W \subseteq \mathcal{A}^*$ as first argument.

$$\partial_{w\alpha}(t) := \partial_\alpha(\partial_w(t)) \quad \partial_W(t) := \bigcup_{w \in W} \partial_w(t)$$

The number of occurrences of letters $\alpha \in \mathcal{A}$ appearing in a regular expression e is called the *alphabetic width* of e . We denote the alphabetic width of e by $\eta(e)$.

Because each occurrence of a letter in a regular expression e can add an element to $\partial_{\mathcal{A}}(e)$, the cardinality of the set of partial derivatives will be less or equal to the alphabetic width of e :

$$|\partial_{\mathcal{A}^+}(e)| \leq \eta(e).$$

In Section 3.6 we will see that this fact makes the alphabetic width an indicator for the maximal number of inference steps necessary to prove or disprove an inequality $r <: s$.

The next section will introduce some examples for the computation of linear forms and partial derivatives of regular expressions.

2.3 Examples for the Derivation of Regular Expressions

We will first construct the linear forms of some regular expressions, from which we derive the partial derivatives.

First, consider the linear form of a single XML node type:

$$lf(\mathbf{foo})$$

Since `foo` is a single alphabetic element, it requires no further simplification. Obviously, we apply rule LF3. This leads to

$$\{\langle \text{foo}, \varepsilon \rangle\}.$$

Another example is

$$lf(\text{foo} \mid \text{bar}).$$

Here, it is obvious to apply rule LF7 to derive:

$$lf(\text{foo}) \cup lf(\text{bar}).$$

The application of rule LF3 to both occurrences of lf in this term leads to:

$$\begin{aligned} & \{\langle \text{foo}, \varepsilon \rangle\} \cup \{\langle \text{bar}, \varepsilon \rangle\} \\ & \{\langle \text{foo}, \varepsilon \rangle, \langle \text{bar}, \varepsilon \rangle\} \end{aligned}$$

A slightly more complex example is:

$$lf((\text{foo} \mid \text{bar}) \cdot \text{baz})$$

The derivation in this case is as follows:

$$\begin{aligned} lf((\text{foo} \mid \text{bar}) \cdot \text{baz}) & \stackrel{\text{LF5}}{=} lf(\text{foo} \mid \text{bar}) \odot \text{baz} \\ & \stackrel{\text{LF7}}{=} lf(\text{foo}) \cup lf(\text{bar}) \odot \text{baz} \\ & \stackrel{2 \times \text{LF3}}{=} \{\langle \text{foo}, \varepsilon \rangle\} \cup \{\langle \text{bar}, \varepsilon \rangle\} \odot \text{baz} \\ & \stackrel{\text{CL7}}{=} \{\langle \text{foo}, \varepsilon \rangle\} \odot \text{baz} \cup \{\langle \text{bar}, \varepsilon \rangle\} \odot \text{baz} \\ & \stackrel{\text{CL5}}{=} \{\langle \text{foo}, \text{baz} \rangle\} \cup \{\langle \text{bar}, \text{baz} \rangle\} \\ & = \{\langle \text{foo}, \text{baz} \rangle, \langle \text{bar}, \text{baz} \rangle\} \end{aligned}$$

In Section 1.5 we introduced a running example, which we will now refer to. Be types

$$\begin{aligned} r & \equiv \text{foo}[\text{str}] \cdot (\text{bar}[\text{int}] \cdot \text{foo})^* \\ s & \equiv (\text{foo}[\text{str}] \cdot \text{bar}[\text{int}])^* \cdot \text{foo}. \end{aligned}$$

like defined in Section 1.5. Now we construct the partial derivatives $\partial_{\text{foo}}(r)$ and $\partial_{\text{foo}}(s)$.

$$\begin{aligned} lf(r) & = lf(\text{foo}[\text{str}] \cdot (\text{bar}[\text{int}] \cdot \text{foo}[\text{str}])^*) \\ & \stackrel{\text{LF5}}{=} lf(\text{foo}[\text{str}]) \odot (\text{bar}[\text{int}] \cdot \text{foo}[\text{str}])^* \\ & \stackrel{\text{LF3}}{=} \{\langle \text{foo}[\text{str}], \varepsilon \rangle\} \odot (\text{bar}[\text{int}] \cdot \text{foo}[\text{str}])^* \\ & \stackrel{\text{CL5}}{=} \{\langle \text{foo}[\text{str}], (\text{bar}[\text{int}] \cdot \text{foo}[\text{str}])^* \rangle\} \end{aligned}$$

Note that the linear form function recognizes the leftmost name `foo` in the tree-shaped regular term `foo[str]` and performs the derivation by this name, not by the whole node type `foo[str]`.

Because $lf(r)$ yields $\{\langle \text{foo}[str], (\text{bar}[int] \cdot \text{foo})^* \rangle\}$ we derive

$$\partial_{\text{foo}}(r) \equiv \{(\text{bar}[int] \cdot \text{foo})^*\}.$$

The derivation of s is as follows:

$$\begin{aligned}
lf(s) &= lf((\text{foo}[str] \cdot \text{bar}[int])^* \cdot \text{foo}[str]) \\
&\stackrel{\text{LF6}}{=} (lf((\text{foo}[str] \cdot \text{bar}[int])^*) \odot \text{foo}[str] \cup lf(\text{foo}[str])) \\
&\stackrel{\text{LF8}}{=} (lf(\text{foo}[str] \cdot \text{bar}[int]) \odot (\text{foo}[str] \cdot \text{bar}[int])^* \odot \text{foo}[str]) \\
&\quad \cup lf(\text{foo}[str]) \\
&\stackrel{\text{LF5}}{=} ((lf(\text{foo}[str]) \odot \text{bar}[int]) \odot (\text{foo}[str] \cdot \text{bar}[int])^*) \\
&\quad \odot \text{foo}[str] \cup lf(\text{foo}[str]) \\
&\stackrel{\text{LF3}}{=} (\{\langle \text{foo}[str], \varepsilon \rangle\} \odot \text{bar}[int]) \odot (\text{foo}[str] \cdot \text{bar}[int])^* \\
&\quad \odot \text{foo}[str] \cup lf(\text{foo}[str]) \\
&\stackrel{\text{CL5}}{=} (\{\langle \text{foo}[str], \text{bar}[int] \rangle\} \odot (\text{foo}[str] \cdot \text{bar}[int])^*) \\
&\quad \odot \text{foo}[str] \cup lf(\text{foo}[str]) \\
&\stackrel{\text{CL6}}{=} (\{\langle \text{foo}[str], \text{bar}[int] \cdot (\text{foo}[str] \cdot \text{bar}[int])^* \rangle\}) \\
&\quad \odot \text{foo}[str] \cup lf(\text{foo}[str]) \\
&\stackrel{\text{CL6}}{=} \{\langle \text{foo}[str], \text{bar}[int] \cdot (\text{foo}[str] \cdot \text{bar}[int])^* \cdot \text{foo}[str] \rangle\} \cup lf(\text{foo}[str]) \\
&= \{\langle \text{foo}[str], \text{bar}[int] \cdot s \rangle\} \cup lf(\text{foo}[str]) \\
&\stackrel{\text{LF3}}{=} \{\langle \text{foo}[str], \text{bar}[int] \cdot s \rangle, \langle \text{foo}[str], \varepsilon \rangle\}
\end{aligned}$$

Applying the definition for partial derivatives of regular expressions leads to

$$\partial_{\text{foo}}(s) \equiv \{\text{bar}[int] \cdot s, \varepsilon\}.$$

In Chapter 3 we will see that the definition of partial derivatives of regular expressions is modified in order to apply it to XQuery subtyping in the implementation of Antimirov's algorithm by the XOBÉ project.

2.4 Partial Derivatives of Regular Inequalities

Analogously to the simplification of single regular expressions, Antimirov introduces partial derivatives of regular inequalities to represent the reduction of a whole inequality.

The partial derivative $\partial_w(r <: s)$ of a regular inequality $r <: s$ given $w \in \mathcal{A}^*$ is a representation of the inequality reduced by w . Its input is a regular inequality $r <: s$. Its output is a finite set of inequalities.

Given two regular expressions $r, s \in \mathbf{Reg}[\mathcal{A}]$ and a word $w \in \mathcal{A}^*$, a regular inequality $e <: f$ is a partial derivative of $r <: s$ given w only if $e \in \partial_w(r)$ and $f \equiv \Sigma\partial_w(s)$. Note that a word w can also consist of a single letter α .

The symbol $\Sigma\partial_w(s)$ denotes a so-called *word derivative* of s given w . The operator Σ is defined as follows:

$$\begin{aligned}\Sigma\emptyset &:= \emptyset \\ \Sigma\{t\} &:= t \\ \Sigma\{t\} \cup T &:= t \mid \Sigma T\end{aligned}$$

With $\partial_w(e) = \{t_0, t_1, \dots, t_n\}$ and $w \in \mathcal{A}^*$, we have:

$$\Sigma\partial_w(e) = t_0 \mid t_1 \mid \dots \mid t_n$$

The effect of the definition of $\partial_w(r <: s)$ is, that each partial derivative $p \in \partial_w(r)$ is compared to the disjunction of all partial derivatives $\partial_w(s)$. The set of all these inequalities is the output of $\partial_w(r <: s)$.

$$\partial_w(r <: s) := \{ e <: \Sigma\partial_w(s) : e \in \partial_w(r) \}$$

Hence the result of $\partial_w(r <: s)$ leads to an increased number of inequalities to check, but to a decreased degree of complexity within these inequalities. Note that $|\partial_w(r <: s)| = |\partial_w(r)|$.

We will not add an application example here because the concept of partial derivatives of regular inequalities will be extended and modified in Section 3.6.

2.5 Rewriting Calculus for Regular Expressions

We can formulate a containment calculus Φ for proving or disproving regular inequalities. Let an *atom* be either a boolean constant *TRUE* or *FALSE* or a regular inequality. Then Φ works on sets of atoms.

Let $S_0 = r <: s$ be an initial inequality. Then an inference in Φ is a sequence of sets of atoms denoted by S_i like the following:

$$S_0 \vdash S_1 \vdash \dots \vdash S_n$$

Each set S_{i+1} is an extension of the previous one S_i . S_{i+1} is derived from S_i by the application of one of the *inference rules* of Φ to an inequality in S_i .

There are three inference rules in Φ . The first rule, DIS, is the rule for disproving an inequality. It derives *FALSE* for an inequality $r <: s$ such that $r \in \mathbf{Reg}_1[\mathcal{A}]$ and $s \in \mathbf{Reg}_0[\mathcal{A}]$. The second and the third rule, UN1 and UN2, are to unfold an inequality into its partial derivatives.

$$\begin{array}{lll} \text{DIS:} & r_1 <: s_0 \vdash \text{FALSE} & \text{for } r_1 \in \mathbf{Reg}_1[\mathcal{A}], s_0 \in \mathbf{Reg}_0[\mathcal{A}] \\ \text{UN1:} & r_0 <: s \vdash \partial_{\mathcal{A}}(r_0 <: s) & \text{for } r_0 \in \mathbf{Reg}_0[\mathcal{A}], s \in \mathbf{Reg}[\mathcal{A}] \\ \text{UN2:} & r_1 <: s_1 \vdash \partial_{\mathcal{A}}(r_1 <: s_1) & \text{for } r_1 \in \mathbf{Reg}_1[\mathcal{A}], s_1 \in \mathbf{Reg}_1[\mathcal{A}] \end{array}$$

To see why the inference process terminates, we have to define the conditions under which the result for $r <: s$ is *TRUE* or *FALSE*.

An inequality $r <: s$ is not valid in $\mathbf{Reg}[\mathcal{A}]$ if and only if a set of atoms S_i containing *FALSE* is derivable in Φ from $r <: s$ (cf. [Ant94]). Therefore we can consider the partial derivatives of a regular inequalities not only as a set of disjunctions but as a conjunction of disjunctions. This is one of two cases in which the inference process terminates.

The second case is when a set S_i is derived that is *saturated*, i.e., it holds that $\partial_{\mathcal{A}}(S_i) = S_i$.

Because of $|\partial_{\mathcal{A}^+}(e)| \leq \eta(e)$ (cf. Section 2.2) and $|\partial_{\mathcal{A}^+}(r <: s)| = |\partial_{\mathcal{A}^+}(r)|$ (cf. Section 2.4), the number of different inequalities derivable from $r <: s$ in Φ is finite. Note that it may take up to $O(|\partial_{\mathcal{A}^+}(r <: s)|)$ inference steps to prove or disprove an inequality $r <: s$.

It follows that after a finite number of steps in the inference process one of the two cases is reached (for details see [Ant94]). Therefore the sequence of derivations ends up with a set of atoms S_i which is either inconsistent – i. e., contains *FALSE* – or saturated – i. e., $\partial_{\alpha}(S_i) = S_i$.

Note that this calculus defines a non-deterministic derivation process because partial derivatives for each $\alpha \in \mathcal{A}$ are computed in an arbitrary order. We will see that it is not useful to use all occurrences of letters for derivation. Thus we will have to add a decision mechanism which letters we will use for derivation in every step S_i . To derive an algorithm from this calculus, we have also to apply some modifications to enable the calculus to handle the XML tree types of Section 1.3.

Chapter 3

Extending Antimirov's Calculus to a Subtyping Algorithm

3.1 The Check for ε -Inclusion

Using DIS and *lf* we have to decide for a given regular expression e , if $e \in \mathbf{Reg}_1[\mathcal{A}]$ holds. Thus we have to implement a check that returns *TRUE* for a given $e \in \mathbf{Reg}[\mathcal{A}]$ only if $\varepsilon \in L(e)$.

We define function *nullable*: $\mathbf{Reg}[\mathcal{A}] \rightarrow \{\mathit{TRUE}, \mathit{FALSE}\}$ such that for a regular expression $e \in \mathbf{Reg}[\mathcal{A}]$ *nullable*(e) is *TRUE* only if $\varepsilon \in L(e)$, i. e., only if $e \in \mathbf{Reg}_1[\mathcal{A}]$:

$$\mathit{nullable}(e) \iff \varepsilon \in L(e) \iff e \in \mathbf{Reg}_1[\mathcal{A}]$$

The input of function *nullable* is a regular expression, the output will be *TRUE* if $e \in \mathbf{Reg}_1[\mathcal{A}]$ or *FALSE* if $e \in \mathbf{Reg}_0[\mathcal{A}]$.

Obviously the type \emptyset does not include ε , but ε of course includes itself:

$$\mathit{nullable}(\emptyset) := \mathit{FALSE} \quad (\text{NA1})$$

$$\mathit{nullable}(\varepsilon) := \mathit{TRUE}. \quad (\text{NA2})$$

Furthermore, we have to consider XML node types and primitive types, which are both obviously not nullable. Named types are nullable only if their definition is nullable.

$$\mathit{nullable}(\mathbf{n}[e]) := \mathit{FALSE} \quad (\text{NA3})$$

$$\mathit{nullable}(p) := \mathit{FALSE} \quad (\text{NA4})$$

$$\mathit{nullable}(N : \mathit{def}(e)) := \mathit{nullable}(\mathit{def}(N)) \quad (\text{NA5})$$

An alternation is nullable, only if at least one of its elements is nullable. A concatenation is nullable only if both elements are nullable.

$$\mathit{nullable}(e_1 \cdot e_2) := \mathit{nullable}(e_1) \wedge \mathit{nullable}(e_2) \quad (\text{NA6})$$

$$\mathit{nullable}(e_1 \mid e_2) := \mathit{nullable}(e_1) \vee \mathit{nullable}(e_2) \quad (\text{NA7})$$

Iteration expressions are nullable if either n is 0 or the iterated expression itself is nullable.

$$\begin{aligned} \text{nullable}(e \{n, m\}) & := \text{if } n = 0: \text{TRUE} \\ & \text{else: } \text{nullable}(e) \end{aligned} \tag{NA8}$$

Example: Applying this to our running example means checking if $\text{nullable}(r)$ or $\text{nullable}(s)$ hold. Let types r, s be defined as previously:

$$\begin{aligned} r & \equiv \text{foo}[str] \cdot (\text{bar}[int] \cdot \text{foo}[str])^* \\ s & \equiv (\text{foo}[str] \cdot \text{bar}[int])^* \cdot \text{foo}[str]. \end{aligned}$$

We analyze r as a concatenation $r_1 \cdot r_2$ of $r_1 \equiv \text{foo}[str]$ and $r_2 \equiv (\text{bar}[int] \cdot \text{foo}[str])^*$. Applying the rule for iteration, we find that $\text{nullable}(r_2)$ holds. As r_1 stands for a node type, we derive $\text{nullable}(r_1) = \text{FALSE}$ and therefore $\text{nullable}(r) = \text{FALSE}$:

$$\begin{aligned} & \text{nullable}(\text{foo}[str] \cdot (\text{bar}[int] \cdot \text{foo}[str])^*) \\ & \stackrel{\text{NA6}}{=} \text{nullable}(\text{foo}[str]) \wedge \text{nullable}((\text{bar}[int] \cdot \text{foo}[str])^*) \\ & \stackrel{\text{NA8}}{=} \text{nullable}(\text{foo}[str]) \wedge \text{TRUE} \\ & \stackrel{\text{NA3}}{=} \text{FALSE} \end{aligned}$$

Analyzing s leads to a concatenation $s_1 \cdot s_2$ of $s_1 \equiv (\text{foo}[str] \cdot \text{bar}[int])^*$ and $s_2 \equiv \text{foo}[str]$. s_1 is obviously nullable and s_2 is obviously not, so $\text{nullable}(s) = \text{FALSE}$:

$$\begin{aligned} & \text{nullable}((\text{foo}[str] \cdot \text{bar}[int])^* \cdot \text{foo}[str]) \\ & \stackrel{\text{NA6}}{=} \text{nullable}((\text{foo}[str] \cdot \text{bar}[int])^*) \wedge \text{nullable}(\text{foo}[str]) \\ & \stackrel{\text{NA3}}{=} \text{nullable}((\text{foo}[str] \cdot \text{bar}[int])^*) \wedge \text{FALSE} \\ & \stackrel{\text{NA8}}{=} \text{FALSE} \end{aligned}$$

3.2 Leading Names

Antimirov's calculus does not define any decision mechanism for which $\alpha \in \mathcal{A}$ the partial derivatives of $r <: s$ are computed in a step $S_i \vdash S_{i+1}$. To avoid computing the partial derivatives for all $\alpha \in \mathcal{A}$ occurring in r in the first step, a strategy has to be found to choose the next α that is used for derivation.

Consider Section 2.2 and recall that the linear form of a regular expression e is a reduced representation of e , “splitted” along the first occurrence of letters in e . This becomes apparent when we note that in the derivation examples of section 2.3 the resulting type pairs in $lf(e)$ have the leftmost occurrences of letters of e as their first components, e.g.,

$$lf((\text{foo}[str] \cdot \text{bar}[int])^* \cdot \text{foo}) = \{ \langle \text{foo}[str], \text{bar}[int] \cdot s \rangle, \langle \text{foo}[str], \varepsilon \rangle \}.$$

Obviously a letter α will only occur as a first component of any type pair in $lf(e)$ if it is a leftmost letter in e . Recall that $\partial_\alpha(e)$ is the subset of $lf(e)$ containing all elements whose first components are α . Therefore it is obvious that $\partial_\alpha(e)$ will be a non-empty set only if α is the leftmost letter or one of the leftmost letters occurring in e .

A modification made in the XOBÉ version of Antimirov's algorithm is to take always the leftmost letter(s) of a given expression to compute partial derivatives (cf. [KeLi03]).

We call the *leading names* of a regular expression the leftmost letters $\alpha \in \mathcal{A}$ of a regular expression. Regular expressions can have more than one leading name. We will now define the leading name function $ln: \mathbf{Reg}[\mathcal{A}] \rightarrow \mathcal{A}$.

The input to function ln is a regular expression e . The output will be the set of leftmost names of e as defined below.

$$ln(\emptyset) := \emptyset \quad (\text{LN1})$$

$$ln(\varepsilon) := \emptyset \quad (\text{LN2})$$

If the expression starts with an element name, we extract the name. In case we have some primitive type like *int*, ln returns the type. In case, the expression starts with a type name, ln is recursively applied on the definition of this name (with $def(name) \in \mathbf{Reg}[\mathcal{A}]$).

$$ln(\mathbf{n}[e]) := \{\mathbf{n}\} \quad (\text{LN3})$$

$$ln(p) := \{p\} \quad (\text{LN4})$$

$$ln(N : def(N)) := ln(def(N)) \quad (\text{LN5})$$

The result ln returns for composite regular expressions is intuitively understandable.

$$ln(e_1 \cdot e_2) := \text{if } nullable(e_1): ln(e_1) \cup ln(e_2) \quad (\text{LN6})$$

$$\text{else: } ln(e_1)$$

$$ln(e_1 \mid e_2) := ln(e_1) \cup ln(e_2) \quad (\text{LN7})$$

$$ln(e \{n, m\}) := ln(e) \quad (\text{LN8})$$

Example: Applied to our example, $ln(r)$ analyzes $r \equiv \mathbf{foo}[str] \cdot (\mathbf{bar}[int] \cdot \mathbf{foo}[str])^*$ as a concatenation of two regular expressions $r_1 \equiv \mathbf{foo}[str]$ and $r_2 \equiv (\mathbf{bar}[int] \cdot \mathbf{foo}[str])^*$. We previously discovered that r_1 is not nullable, so we have to return $ln(r_1)$ as result for $ln(r)$. For r_1 is an XML node type, the result is $\{\mathbf{foo}\}$:

$$\begin{aligned} ln(\mathbf{foo}[str] \cdot (\mathbf{bar}[int] \cdot \mathbf{foo}[str])^*) \\ &= ln(\mathbf{foo}[str]) \\ &\stackrel{\text{LN6}}{=} ln(\mathbf{foo}[str]) \\ &\stackrel{\text{LN3}}{=} \{\mathbf{foo}\} \end{aligned}$$

As s is analyzed as a concatenation of $s_1 \equiv (\mathbf{foo}[str] \cdot \mathbf{bar}[int])^*$ and $s_2 \equiv \mathbf{foo}[str]$ and s_1 is nullable, not only s_1 is relevant for the result of $ln(s)$

but also s_2 . Therefore the result of $ln(s)$ is $ln(s_1) \cup ln(s_2)$. For s_2 is an element name, we get the result $\{\mathbf{foo}\}$. The Kleene star in s_1 is “simplified away” by rule LN8, so we get $s_3 \equiv \mathbf{foo}[str] \cdot \mathbf{bar}[int]$. This is again a concatenation of $s_4 \equiv \mathbf{foo}[str]$ and $s_5 \equiv \mathbf{bar}[int]$. As s_4 is not nullable, only $ln(s_4)$ is of interest for computing the result of $ln(s_1)$, the analysis of s_5 becomes obsolete. To analyze s_4 we apply the first rule from above and get $ln(s_4) \equiv \mathbf{foo}$. So $ln(s_1) \cup ln(s_2)$ returns $\{\mathbf{foo}\} \cup \{\mathbf{foo}\}$ which is $\{\mathbf{foo}\}$:

$$\begin{aligned}
& ln((\mathbf{foo}[str] \cdot \mathbf{bar}[int])^* \cdot \mathbf{foo}[str]) \\
& \stackrel{\text{LN6}}{=} ln((\mathbf{foo}[str] \cdot \mathbf{bar}[int])^*) \cup ln(\mathbf{foo}[str]) \\
& \stackrel{\text{LN3}}{=} ln((\mathbf{foo}[str] \cdot \mathbf{bar}[int])^*) \cup \{\mathbf{foo}\} \\
& \stackrel{\text{LN8}}{=} ln(\mathbf{foo}[str] \cdot \mathbf{bar}[int]) \cup \{\mathbf{foo}\} \\
& \stackrel{\text{LN6}}{=} \{\mathbf{foo}\} \cup \{\mathbf{foo}\} \\
& = \{\mathbf{foo}\}
\end{aligned}$$

3.3 Two Wellformedness Constraints for Types

In practice, XQuery types are often defined by recursion as in:

$$T1 : int \cdot T1 \mid \varepsilon.$$

We have to avoid applying lf on a recursive occurrence, because this would lead to an endless recursion.

Example: Let us consider the case of type $T2$:

$$T2 : T2 \cdot int \mid \varepsilon.$$

The derivation of $T2$ will lead to an endless recursion as follows:

$$\begin{aligned}
lf(T2) & \stackrel{\text{LF4}}{=} lf(T2 \cdot int \mid \varepsilon) \\
& \stackrel{\text{LF7}}{=} lf(T2 \cdot int) \cup lf(\varepsilon) \\
& \stackrel{\text{LF2}}{=} lf(T2 \cdot int) \\
& \stackrel{\text{LF6}}{=} lf(T2) \odot int \cup lf(int) \\
& \stackrel{\text{LF4}}{=} lf(T2 \cdot int \mid \varepsilon) \odot int \cup lf(int) \\
& \vdots
\end{aligned}$$

It is obvious that the derivation of $T2$ becomes circular with a new application of rule LF4 to the recursive occurrence of $T2$, because each derivation leads to a new occurrence of $T2$, which was the initial type to derive.

In this section, we will define some *wellformedness constraints* which ensure avoiding endless recursions in the course of derivation (cf. [KeLi03]).

First we will analyze why the derivation of $T2$ fails. The derivation of a named type always starts with the application of rule LF4. This rule applies function lf to the definition of the initial named type, which we will call *root type* in the following.

Because the definition of the root type $T2$ contains recursive occurrences of this type, an endless, periodically repeated application of rule LF4 is unavoidable if lf is applied to any of these recursive occurrences.

So let us consider, which rules cause an application of lf to a recursive occurrence. In case of $T2$, obviously the application of rule LF6 causes the problem. Furthermore it is obvious, that in the course of a linear form computation process, also rules LF7 and LF8 will apply lf on a recursive occurrence, since this recursive occurrence is not located within a concatenation.

Example: Consider the following case.

$$\begin{aligned}
lf(T3 : T3 \cdot \text{foo} \mid T3 \mid \varepsilon) & \stackrel{\text{LF4}}{=} lf(T3 \cdot \text{foo} \mid T3 \mid \varepsilon) \\
& \stackrel{\text{LF7}}{=} lf(T3 \cdot \text{foo} \mid T3) \cup lf(\varepsilon) \\
& \stackrel{\text{LF2}}{=} lf(T3 \cdot \text{foo} \mid T3) \\
& \stackrel{\text{LF7}}{=} lf(T3 \cdot \text{foo}) \cup lf(T3) \\
& \vdots
\end{aligned}$$

In this case, we step into an endless recursion because rule LF7 causes an application of lf to a recursive occurrence. The reader may construct her own example for rule LF8 leading to an application of lf to a recursive occurrence.

We say that in case of type $T2$, the recursive occurrence of $T2$ stands in *head position* in the concatenation, because the concatenation starts with the recursive occurrence. In case of type $T1$ we say that the recursive occurrence of $T1$ stands in *tail position*, because the concatenation ends with the recursive occurrence.

In a concatenation containing n members, be 0 the index of the head position and $n - 1$ the index of the tail position. We call the partial concatenation expression $c_0 \cdot c_1 \cdot \dots \cdot c_l$ for all $0 < l < n - 1$ the *head* of the concatenation.

It is obvious, that the application of lf to recursive occurrences caused by rules LF7 and LF8 can only be avoided by restricting recursive occurrences to stand within concatenations.

But also with this restriction, problems arise if a recursive occurrence stands in head position within the concatenation like in type $T2$. In this case, the application of rules LF5 or LF6 could cause an application of lf to the recursive occurrence. An example for this situation is the application of rule LF5 in the derivation of type $T2$.

It is also obvious that recursive occurrences are not allowed to stand in another non-tail position because this can hurt regularity (cf. Section 7.2 for an example of a non-regular type).

This leads to the first wellformedness constraint:

Within the definition of a named type, recursive occurrences of the root type are only allowed in tail positions within concatenations.

This constraint avoids the application of lf to a recursive occurrence by rules LF5 (cf. derivation of $T2$), LF7 (cf. derivation of $T3$) and LF8 and ensures regularity.

Computing the linear form of a concatenation involves a test if the head of the concatenation is nullable. The result of this test determines the choose for applying rule LF5 or LF6.

In some cases the application of *nullable* to a recursive occurrence will also lead to an endless derivation as well as in the case of lf . Granting recursive occurrences to stand in tail positions within concatenations does not avoid applying rule NA6 to a concatenation containing a recursive occurrence. Therefore, *nullable* could be applied to a recursive occurrence and cause an endless recursion in spite of the first wellformedness constraint being fulfilled.

Example: Consider the application of rule LF6 in the derivation of $T2$. We decided to apply rule LF6 and not rule LF5 because we know that type $T2$ is nullable. But this is a bypass in the derivation of $T2$: It is computationally undecidable whether type $T2$ is nullable or not, because the application of rule NA6 to the intermediate derivation result $T2 \cdot int$ would apply *nullable* to the recursive occurrence of $T2$ and therefore lead to an endless recursion. Thus after the application of rule LF7, the derivation would never yield a result and therefore never lead to the decision to apply rule LF6.

If we ensure, that recursive occurrences are preceded by a non-nullable head, we exclude the application of rule LF6 to the concatenation and therefore the erroneous effect which the application of this rule causes in the course of the derivation of $T2$.

Implementing rule NA6 we can use short circuit evaluation of the \wedge -operator to avoid application of rule NA6 leading to an endless recursion. If we apply rule NA6 to a concatenation with a non-nullable head, short circuit evaluation of the \wedge -operator will avoid evaluating if the recursive occurrence is nullable.

Example: Consider the computation of linear form of $T1$.

$$\begin{aligned}
lf(T1 : int \cdot T1 \mid \varepsilon) &\stackrel{\text{LF4}}{=} lf(int \cdot T1 \mid \varepsilon) \\
&\stackrel{\text{LF7}}{=} lf(int \cdot T1) \cup lf(\varepsilon) \\
&\stackrel{\text{LF2}}{=} lf(int \cdot T1) \\
&\stackrel{\text{LF5}}{=} lf(int) \odot T1 \\
&\stackrel{\text{LF3}}{=} \{\langle int, \varepsilon \rangle\} \odot T1 \\
&\stackrel{\text{CL5}}{=} \{\langle int, T1 \rangle\}
\end{aligned}$$

After rule LF7 was applied, we receive the expression

$$int \cdot T1$$

and have to test if it is nullable to decide whether to apply rule LF5 or LF6 to it. If it is nullable, rule LF6 has to be applied, otherwise rule LF5 has to be applied.

$$\begin{aligned}
\text{nullable}(\text{int} \cdot T1) & \stackrel{\text{NA6}}{=} \text{nullable}(\text{int}) \wedge \text{nullable}(T1) \\
& \stackrel{\text{NA4}}{=} \text{FALSE} \wedge \text{nullable}(T1) \\
& \quad \uparrow \\
& \quad \text{Stop evaluation.}
\end{aligned}$$

The short circuit evaluation stops before $\text{nullable}(T1)$ is evaluated. This will always be the case because we ensure a non-nullable head for concatenations containing recursive occurrences.

Putting it all together leads to the following two wellformedness constraints:

- 1) Recursive occurrences within regular expressions may appear only in tail positions within concatenations.
- 2) A recursive occurrence within a concatenation must be preceded by a non-nullable head.

The next section will discuss the technique for checking types for fulfillment of these wellformedness constraints.

3.4 Checking Wellformedness of Types

For each of these wellformedness constraints, we define a function that recursively tests an expression e for the corresponding constraint.

For both functions, the strategy is to pass a flag f that indicates whether a recursive occurrence is allowed within the type that is currently checked. The function itself checks the corresponding type and compares the result to the flag.

Function $tp: \mathbf{Reg}[\mathcal{A}] \times \{TRUE, FALSE\} \times \mathbf{Reg}[\mathcal{A}] \rightarrow \{TRUE, FALSE\}$ implements the check for tail positions. For all $t \in \mathbf{Reg}[\mathcal{A}]$, $f \in \{TRUE, FALSE\}$, and $e \in \mathbf{Reg}[\mathcal{A}]$ it returns $TRUE$ only if all recursive occurrences contained in e are in tail positions, otherwise $FALSE$. Note that t is the root type.

$$tp(t, f, \emptyset) := TRUE \quad (\text{TP1})$$

$$tp(t, f, \varepsilon) := TRUE \quad (\text{TP2})$$

$$tp(t, f, \mathbf{n}[e]) := TRUE \quad (\text{TP3})$$

$$tp(t, f, p) := TRUE \quad (\text{TP4})$$

$$\begin{aligned}
tp(t, f, N : \text{def}(N)) & := \text{if } (t = N) : f, \\
& \quad \text{else: } tp(N, TRUE, \text{def}(N))
\end{aligned} \quad (\text{TP5})$$

$$tp(t, f, e_1 \cdot e_2) := tp(t, FALSE, e_1) \wedge tp(t, f, e_2) \quad (\text{TP6})$$

$$tp(t, f, e_1 | e_2) := tp(t, f, e_1) \wedge tp(t, f, e_2) \quad (\text{TP7})$$

$$\begin{aligned}
tp(t, f, e \{n, m\}) & := \text{if } (n = 0 \wedge m = 1) : tp(t, f, e) \\
& \quad \text{else: } tp(t, FALSE, e)
\end{aligned} \quad (\text{TP8})$$

The first four rules say, that tp yields true for \emptyset , ε and all node types and primitive types. Rule TP5 says that each type which is a recursive occurrence, yields the value of the flag, while another implicit definition of a named type is checked separately with its own name as root type name.

The initialization value for f is $TRUE$ because for the top-level type, recursive occurrences are allowed.

Example: We want to find out, if type

$$FOO : \text{bar} \cdot \text{int} \cdot FOO \mid \varepsilon$$

fulfills wellformedness constraint 1, i.e., if all recursive occurrences in its definition are in tail positions.

$$\begin{aligned}
& tp(FOO, TRUE, FOO) \\
& \stackrel{TP5}{=} tp(FOO, TRUE, \text{bar} \cdot \text{int} \cdot FOO \mid \varepsilon) \\
& \stackrel{TP7}{=} tp(FOO, TRUE, \text{bar} \cdot \text{int} \cdot FOO) \wedge tp(FOO, TRUE, \varepsilon) \\
& \stackrel{TP2}{=} tp(FOO, TRUE, \text{bar} \cdot \text{int} \cdot FOO) \\
& \stackrel{TP6}{=} tp(FOO, FALSE, \text{bar} \cdot \text{int}) \wedge tp(FOO, TRUE, FOO) \\
& \stackrel{TP5}{=} tp(FOO, FALSE, \text{bar} \cdot \text{int}) \\
& \stackrel{TP6}{=} tp(FOO, FALSE, \text{bar}) \wedge tp(FOO, FALSE, \text{int}) \\
& \stackrel{TP4}{=} tp(FOO, FALSE, \text{bar}) \\
& \stackrel{TP3}{=} TRUE
\end{aligned}$$

Testing for a non-nullable head of recursive occurrences is done by function $nh: \mathbf{Reg}[\mathcal{A}] \times \{TRUE, FALSE\} \rightarrow \{TRUE, FALSE\}$.

For a flag $f \in \{TRUE, FALSE\}$ and a regular expression $e \in \mathbf{Reg}[\mathcal{A}]$, nh returns $TRUE$ if every recursive occurrence in e has a non-nullable head, otherwise $FALSE$. Flag f says if nullable occurrences are allowed in the expression to be tested.

$$nh(f, \emptyset) := TRUE \quad (\text{NH1})$$

$$nh(f, \varepsilon) := TRUE \quad (\text{NH2})$$

$$nh(f, \mathbf{n}[e]) := TRUE \quad (\text{NH3})$$

$$nh(f, p) := TRUE \quad (\text{NH4})$$

$$\begin{aligned}
nh(f, N : \text{def}(N)) & := \text{if } (\text{def}(N) = 0) : f, \\
& \quad \text{else: } nh(FALSE, \text{def}(N))
\end{aligned} \quad (\text{NH5})$$

$$nh(f, e_1 \cdot e_2) := nh(f, e_1) \wedge nh((f \vee \neg \text{nullable}(e_1)), e_2) \quad (\text{NH6})$$

$$nh(f, e_1 \mid e_2) := nh(f, e_1) \wedge nh(f, e_2) \quad (\text{NH7})$$

$$nh(f, e \{n, m\}) := nh(f, e) \quad (\text{NH8})$$

The first four rules say, that for \emptyset , ε and all node types and primitive types, nh yields $TRUE$. Rule NH5 says that for a recursive occurrence as input type,

the flag is returned. For another implicit definition of a named type, a separate check is performed on the definition.

The initial call must be passed $f = FALSE$. This is obvious because initially, no non-nullable head has been encountered yet.

Example: For type

$$FOO : \mathbf{bar} \cdot \mathit{int} \cdot FOO \mid \varepsilon$$

we want also to find out if it fulfills wellformedness constraint 2, i.e., if all recursive occurrences within the definition of FOO are preceded by non-nullable heads.

$$nh(FALSE, FOO)$$

$$\begin{aligned} & \stackrel{\text{NH5}}{=} nh(FALSE, \mathbf{bar} \cdot \mathit{int} \cdot FOO \mid \varepsilon) \\ & \stackrel{\text{NH7}}{=} nh(FALSE, \mathbf{bar} \cdot \mathit{int} \cdot FOO) \wedge nh(FALSE, \varepsilon) \\ & \stackrel{\text{NH2}}{=} nh(FALSE, \mathbf{bar} \cdot \mathit{int} \cdot FOO) \\ & \stackrel{\text{NH6}}{=} nh(FALSE, \mathbf{bar} \cdot \mathit{int}) \wedge nh(TRUE, FOO) \\ & \stackrel{\text{NH5}}{=} nh(FALSE, \mathbf{bar} \cdot \mathit{int}) \\ & \stackrel{\text{NH6}}{=} nh(FALSE, \mathbf{bar}) \wedge nh(TRUE, \mathit{int}) \\ & \stackrel{\text{NH4}}{=} nh(FALSE, \mathbf{bar}) \\ & \stackrel{\text{NH3}}{=} TRUE \end{aligned}$$

Our constraints exclude all types whose analysis could lead to endless derivations. But they also exclude some types which could be normalized such that they are wellformed, for example:

$$T4 : (T4 \cdot \mathbf{b}) \mid \mathbf{a} \quad \equiv \quad \mathbf{a} \cdot \mathbf{b}^*$$

It remains as future work to develop a normalization strategy that can recognize “repairable” non-wellformed types and rewrite them to equivalent types which are wellformed. This topic will be referenced in Section 7.2.

On the other hand, the wellformedness constraints do not ensure, that there will never occur an inequality $r <: s$ with r or s being a recursive occurrence. The wellformedness constraints only ensure that during the derivation process no application of lf or $nullable$ will lead to an endless recursion.

Furthermore, functions tp and nh need to access the definition of the named recursive type they are currently analyzing. This is necessary because rules TP5 and NH5 consider the definition of this type.

Hence we have to ensure that in each representation of a recursive named type passed to tp and nh , the recursive occurrences are replaced by the original definition of the type. This task is performed by function $unfold$. Because unfolding is not part of the algorithm but a mere technical requirement, its description is postponed until Section 5.2 in Chapter 5.

3.5 Partial Derivatives of Regular Expressions Re-defined

Since the types we have to handle represent XML tree structures, we have to extend the concept of partial derivatives such that the derivation descends into the tree structure of node types. Until now we were dealing with “flat” types only.

Antimirov defines partial derivatives of regular expressions as follows:

$$\partial_\alpha(r) := \{ e : \langle \alpha, e \rangle \in lf(r), e \neq \emptyset \}$$

If α is of the form $\mathbf{n}[e]$, i.e., an XML node type with content type e , it is not sufficient to cut off the leading name and analyze the partial derivatives. This strategy would not be able to distinguish node types which only differ in the structure of their content types. The original definition of Antimirov does not reflect this fact. We have to add a mechanism that descends into the tree structure. Hence we renew the definition of partial derivatives as follows:

$$\partial_\alpha(r) := \{ \langle cn(\alpha), e \rangle : \langle \alpha, e \rangle \in lf(r), e \neq \emptyset \}$$

In Antimirov’s original approach, we have $\partial_w(r <: s) \in \mathbf{SReg}$ with $\mathbf{SReg} \equiv \mathbf{Set}[\mathbf{Reg}[\mathcal{A}] \setminus \{\emptyset\}]$ (cf. Section 2.2). Now we have redefined $\partial_w(r)$ such that $\partial_w(r) \in \mathbf{SReg} \times \mathbf{SReg}$. Therefore partial derivatives of regular expressions consist of ordered pairs and not of regular expressions.

This definition involves the content function $cn : \mathbf{Reg}[\mathcal{A}] \rightarrow \mathbf{SReg}$. Function cn is only applied on first elements of partial derivatives. Therefore it is defined only on letter types, as there are XML node types and primitive types, because only a letter type can be a first component of a type pair included in a linear form.

$$cn(\mathbf{n}[e]) := e \tag{CN1}$$

$$cn(\mathbf{n}) := \varepsilon \tag{CN2}$$

$$cn(p) := \varepsilon \tag{CN3}$$

Applied to an XML node type $\mathbf{n}[e]$, cn yields a type again: the content of \mathbf{n} is e , i.e., the XML tree structure contained in the element \mathbf{n} . The content of XML node types that do not contain anything is ε . The content of a primitive type is ε .

Example: Application of function cn . Consider the following type.

$$\mathbf{a}[\mathbf{foo} \cdot \mathbf{bar}[\mathbf{baz}]]$$

An application of cn yields the content of \mathbf{a} :

$$cn(\mathbf{a}[\mathbf{foo} \cdot \mathbf{bar}[\mathbf{baz}]]) \stackrel{\text{CN1}}{=} \mathbf{foo} \cdot \mathbf{bar}[\mathbf{baz}]$$

Example: Application of redefined partial derivatives.

We will apply the new definition of partial derivatives to the previous running example. The leading names of r and s are the same: $\{\mathbf{foo}\}$. Antimirov's calculus reduces r and s by the name $\{\mathbf{foo}\}$. Thus, we will derive $\partial_{\mathbf{foo}}(r)$ and $\partial_{\mathbf{foo}}(s)$.

In accordance to the definition we receive:

$$\begin{aligned}\partial_{\mathbf{foo}}(r) &\equiv \{\langle str, ((\mathbf{bar}[int] \cdot \mathbf{foo}[str])^*) \rangle\} \\ \partial_{\mathbf{foo}}(s) &\equiv \{\langle str, \mathbf{bar}[int] \cdot s \rangle, \langle str, \varepsilon \rangle\}\end{aligned}$$

As a result of the derivation process, we receive additional instances of expressions, but these are simpler to check since they are of simpler structure.

Our redefinition of partial derivatives of regular expressions does not simplify the partial derivatives of regular inequalities. But we will also introduce a simplification technique for partial derivatives of regular inequalities.

Given a set of words $W \subseteq \mathcal{A}^+$, the number of inequalities within the set of partial derivatives of a given inequality is:

$$|\partial_W(r <: s)| = |\partial_W(r)|$$

as follows from the definition of partial derivatives of regular inequalities as stated in Section 2.4.

The simplification technique to be introduced in Section 3.6 increases the number of partial derivatives of regular inequality $r <: s$ to:

$$|\partial_W(r <: s)| = |\partial_W(r)| * 2^{|\partial_W(s)|+1}.$$

This will become obvious in the course of Section 3.6.

The benefit of this simplification will be a simpler structure of the resulting inequalities, although their number is increased at the same time.

3.6 A Simplification for Partial Derivatives of Regular Inequalities

According to the redefinition of partial derivatives of regular expressions in Section 3.5 the definition of partial derivatives of regular inequalities is adjusted. Recall from Section 2.4:

$$\Sigma \partial_w(e) = t_0 | t_1 | \dots | t_m \quad \text{for all } t_i \in \partial_w(e).$$

Because of $\partial_w(r) \in \mathbf{SReg} \times \mathbf{SReg}$ (as previously described in Section 3.5) we have also for each $i \in \{0, \dots, m\} : t_i \in \mathbf{SReg} \times \mathbf{SReg}$.

Let $n = |\partial_w(r)|$ and $m = |\partial_w(s)|$. As result of $\partial_w(r <: s)$ we receive a set of new inequalities of the form

$$\begin{aligned}\langle a_0, b_0 \rangle &<: \langle c_0, d_0 \rangle | \langle c_1, d_1 \rangle | \dots | \langle c_{m-1}, d_{m-1} \rangle \\ \langle a_1, b_1 \rangle &<: \langle c_0, d_0 \rangle | \langle c_1, d_1 \rangle | \dots | \langle c_{m-1}, d_{m-1} \rangle \\ &\vdots \\ \langle a_{n-1}, b_{n-1} \rangle &<: \langle c_0, d_0 \rangle | \langle c_1, d_1 \rangle | \dots | \langle c_{m-1}, d_{m-1} \rangle.\end{aligned}$$

Since Antimirov's derivation calculus operates on types (not on pairs of types), we need to transform the new inequalities. In [Hos00] a set theoretic observation is made which we can use for simplification. This simplification will cause an increase of $|\partial_w(r <: s)|$ but a decrease of complexity in the resulting inequalities.

Be $\mathbb{A} := \mathbf{Reg}[\mathcal{A}]$. Then for two types a and b a cross product $a \times b$ is equal to $(a \times \mathbb{A}) \cap (\mathbb{A} \times b)$. The first step is to rewrite the resulting inequalities from above as follows for each $i \in \{0, 1, \dots, n-1\}$:

$$\langle a_i, b_i \rangle <: \langle c_0, \mathbb{A} \rangle \cap \langle \mathbb{A}, d_0 \rangle \mid \dots \mid \langle c_{m-1}, \mathbb{A} \rangle \cap \langle \mathbb{A}, d_{m-1} \rangle.$$

The second step is to apply distributivity of intersections over unions. Let e.g., be $m = 2$. Then this leads to the following for each $i \in \{0, 1, \dots, n-1\}$:

$$\begin{aligned} \langle a_i, b_i \rangle <: & \{ \langle c_0, \mathbb{A} \rangle \} \mid \{ \langle c_1, \mathbb{A} \rangle \} \cap \\ & \{ \langle c_0, \mathbb{A} \rangle \} \mid \{ \langle \mathbb{A}, d_1 \rangle \} \cap \\ & \{ \langle \mathbb{A}, d_0 \rangle \} \mid \{ \langle c_1, \mathbb{A} \rangle \} \cap \\ & \{ \langle \mathbb{A}, d_0 \rangle \} \mid \{ \langle \mathbb{A}, d_1 \rangle \} \end{aligned}$$

This case can easily be extended to cases, where m is arbitrary. We recognize, that the number of intersections in each of the new inequalities is 2^m . In each clause, if c_i appears, the corresponding d_i does not appear and vice versa. Hence, we can rewrite each clause as

$$(\bigvee_{j \in J} \langle c_j, \mathbb{A} \rangle) \mid (\bigvee_{j \in \bar{J}} \langle \mathbb{A}, d_j \rangle)$$

with $J = \{1, \dots, m\}$ and $\bar{J} = \{1, \dots, m\} \setminus J$. Note that for $i = 0$, a type t_i represents \emptyset in accordance to the definition of Σ .

The third step is easy: Because the conjunctive form above consists of the intersection of such forms for all subsets of J , – i. e., for every $p \in \mathcal{P}(J)$, – we only have to check the following:

$$\langle a_i, b_i \rangle <: (\bigvee_{j \in J} \langle c_j, \mathbb{A} \rangle) \mid (\bigvee_{j \in \bar{J}} \langle \mathbb{A}, d_j \rangle).$$

This is equivalent to:

$$\langle a_i, b_i \rangle <: ((\bigvee_{j \in J} c_j, \mathbb{A})) \mid (\langle \mathbb{A}, \bigvee_{j \in \bar{J}} d_j \rangle).$$

The fact, that each clause on the right has \mathbb{A} as one of its arguments, the fourth and last step of the simplification removes \mathbb{A} . Thus it is sufficient to test for each $i \in \{0, 1, \dots, n-1\}$.

$$a_i <: (\bigvee_{j \in J} c_j) \quad \vee \quad b_i <: (\bigvee_{j \in \bar{J}} d_j).$$

This also means, that $\partial_\alpha(r <: s) \in \mathbf{Reg}[\mathcal{A}] \times \mathbf{Reg}[\mathcal{A}]$.

It is obvious, that this transformation leads always to $n * 2^m$ disjunctions and $n * 2^{m+1}$ new inequalities to test (cf. Section 3.5). Now consider the alphabetic width η of r and s . Because $n \leq \eta(r)$ and $m \leq \eta(s)$ as stated in Section 2.2, it holds that

$$|\partial_{\mathcal{A}^+}(r <: s)| \leq \eta(r) * 2^{\eta(s)+1}.$$

Because the number of inference steps necessary to prove or disprove a given inequality is in $O(|\partial_{\mathcal{A}^+}(r <: s)|)$ it follows that it is in $O(\eta(r) * 2^{\eta(s)+1})$.

For $n = 1$ and $m = 2$ this leads to $1 * 2^{2+1} = 8$ inequalities in 4 disjunction clauses:

$$\begin{array}{llll} (a <: c_1 \mid c_2 & \vee & b <: \emptyset) & \wedge \\ (a <: c_1 & \vee & b <: d_2) & \wedge \\ (a <: c_2 & \vee & b <: d_1) & \wedge \\ (a <: \emptyset & \vee & b <: d_1 \mid d_2) & \end{array}$$

Because in our running example m is also 2, the result of $\partial_{\text{foo}}(r <: s)$ is:

$$\begin{array}{llll} (str <: str \mid str & \vee & (\text{bar}[int] \cdot \text{foo})^* <: \emptyset) & \wedge \\ (str <: str & \vee & (\text{bar}[int] \cdot \text{foo})^* <: \varepsilon) & \wedge \\ (str <: str & \vee & (\text{bar}[int] \cdot \text{foo})^* <: \text{bar}[int] \cdot s) & \wedge \\ (str <: \emptyset & \vee & (\text{bar}[int] \cdot \text{foo})^* <: \text{bar}[int] \cdot s \mid \varepsilon) & \end{array}$$

Before we show in Section 3.8 how the derivation technique and calculus Φ are transformed into a subtyping algorithm, we will have to examine the structure of those inequalities for which no derivation is needed to determine the result.

3.7 Trivial Cases

The notion of “trivial inconsistency” was used so far for inequalities $r <: s$ with $\text{nullable}(r)$ and $\neg \text{nullable}(s)$ as follows:

$$\forall (r \in \mathbf{Reg}_1[\mathcal{A}], s \in \mathbf{Reg}_0[\mathcal{A}]) : \quad r <: s \vdash \text{FALSE}. \quad (\text{T1})$$

For such an inequality, rule DIS in Φ derives *FALSE*.

Calculus Φ ensures termination by avoiding the circular derivation of inequalities that were previously analyzed as described in Section 2.5. We stated that deriving an inequality, that has been previously analyzed, means it was not possible to derive any new inequalities from this inequality. In this case, the calculus yields *TRUE*.

$$\forall (r <: s) : \quad r <: s \vdash \dots \vdash r <: s \implies r <: s \vdash \text{TRUE} \quad (\text{T2})$$

Only these two cases were mentioned in Section 2.5 but there are some other trivial cases not yet discussed which make derivation obsolete. We will consider these trivial inconsistencies now because they can shorten the path to the result in many cases.

Obviously, the following case is trivial:

$$\forall (t \in \mathbf{Reg}[\mathcal{A}]) : \quad t <: t \vdash \text{TRUE} \quad (\text{T3})$$

Hence if the derivation process faces an inequality $r <: s$ with $r \equiv s$, immediately *TRUE* is derived.

Now let us consider inequalities with \emptyset . For \emptyset , the following holds:

$$\forall t \in \mathbf{Reg}[\mathcal{A}] : \quad \emptyset <: t \vdash \mathit{TRUE} \quad (\text{T4})$$

$$\forall t \in \mathbf{Reg}[\mathcal{A}] \setminus \{\emptyset\} : \quad t <: \emptyset \vdash \mathit{FALSE} \quad (\text{T5})$$

Therefore it is obvious that the presence of \emptyset as a top-level type in an inequality constitutes immediately decidable cases. (Note that $\emptyset <: \emptyset$ is *TRUE* in accordance to T3.)

Another trivial case is obviously induced by the presence of ε as a top-level type of an inequality.

$$\forall t \in \mathbf{Reg}_0[\mathcal{A}] : \quad \varepsilon <: t \vdash \mathit{FALSE} \quad (\text{T6})$$

$$\forall t \in \mathbf{Reg}_1[\mathcal{A}] : \quad \varepsilon <: t \vdash \mathit{TRUE} \quad (\text{T7})$$

In cases T6 and T7 it becomes obvious, that the result from $\varepsilon <: t$ is equivalent with *nullable*(t). This includes also $\varepsilon <: \varepsilon$ which is *TRUE* (cf. case T3).

If $s \equiv \varepsilon$ the following holds:

$$\forall t \in \mathbf{Reg}[\mathcal{A}], \ln(t) \neq \emptyset : \quad t <: \varepsilon \vdash \mathit{FALSE}. \quad (\text{T8})$$

If a type is formed only of \emptyset and ε , it has no leading names. In this case, no partial derivatives can be computed. But according to the trivial cases concerning \emptyset and ε above, we can define this case as *TRUE*, if we ensure, that within the algorithm, cases T5 and T6 are checked before the check for case T9 is performed.

$$\forall r \in \mathbf{Reg}[\mathcal{A}] \setminus \{\varepsilon\}, s \in \mathbf{Reg}[\mathcal{A}] \setminus \{\emptyset\}, \ln(r) \equiv \emptyset : \quad r <: s \vdash \mathit{TRUE}. \quad (\text{T9})$$

To benefit from the absence of leading names, in an implementation of the subtyping algorithm, the check for case T6 will be done before checking case T9.

If the set of partial derivatives for at least one input type is empty, no derivation is possible and therefore we have to return *FALSE*.

Thus for $\ln(r) \neq \emptyset$:

$$\forall r, s \in \mathbf{Reg}[\mathcal{A}], \partial_{\ln(r)}(r) \equiv \emptyset \vee \partial_{\ln(r)}(s) \equiv \emptyset : \quad r <: s \vdash \mathit{FALSE} \quad (\text{T10})$$

Section 3.8 presents an example for this case and shows some connection to case T2.

3.8 From a Calculus to an Algorithm

The idea of the subtyping algorithm is the successive simplification of the expressions the inequality consists of. The formal way to simplify the expressions is the calculus Φ . Our task in this section is to extend this calculus to an algorithm.

The following pseudo-code shows the subtyping algorithm based on Antimirov's calculus. It is very similar to the variant of Antimirov's algorithm used in the XOBÉ Project of the University of Lübeck (cf. [KeLi03]).

Function $\text{prove}(r <: s, A)$

input : regular inequality $r <: s$, assumption set A
output : $TRUE$ iff $r <: s \vdash TRUE$, else $FALSE$.

```

1 if ( $r \equiv s \vee r <: s \in A \vee r \equiv \emptyset$ ) then
2    $\lfloor$  return  $TRUE$ ;
3 if ( $(s \equiv \emptyset) \vee (\text{nullable}(r) \wedge \neg \text{nullable}(s))$ ) then
4    $\lfloor$  return  $FALSE$ ;
5 if ( $r \equiv \varepsilon$ ) then
6    $\lfloor$  return  $\text{nullable}(s)$ ;
7  $\text{names} \leftarrow \text{leadingNames}(r)$ ;
8 if ( $\text{names} \equiv \emptyset$ ) then
9    $\lfloor$  return  $TRUE$ ;
   else
10   $\lfloor$  if ( $s \equiv \varepsilon$ ) then
11     $\lfloor$  return  $FALSE$ ;
12 foreach ( $n \in \text{names}$ ) do
13    $\lfloor$   $\text{pd} \leftarrow \text{pd} \cup \partial_n(r <: s)$ ;
14  $\text{result} \leftarrow TRUE$ ;
15 if ( $\text{pd} \equiv \emptyset$ ) then
16    $\lfloor$  return  $FALSE$ ;
   else
17    $\lfloor$   $A \leftarrow A \cup \{r <: s\}$ ;
18    $\lfloor$  foreach ( $(r_1 <: s_1) \vee (r_2 <: s_2) \in \text{pd}$ ) do
19      $\lfloor$   $\text{result} \leftarrow \text{result} \wedge (\text{prove}(r_1 <: s_1, A) \vee \text{prove}(r_2 <: s_2, A))$ ;
20 return  $\text{result}$ ;
```

The input of the algorithm is an inequality $r <: s$. The output is $TRUE$ or $FALSE$.

In line 1 the algorithm starts the check for some trivial inconsistency, e.g., if $r <: s$ is already analyzed. If this is verified, the algorithm did not succeed in deriving new inequalities from a previously analyzed input, i.e., it accepts the input as $TRUE$ because we reached a saturated partial derivative (cf. case T2). In the same line, also cases T3 and T4 are checked.

Line 3 contains the test, if r is nullable and s is not nullable. This is the implementation of DIS (cf. case T1). The same line also checks for case T5.

Note that at first we try to apply the rule DIS. Only if DIS is not applicable

we start the derivation. This ensures, that we apply UN1 and UN2 only if DIS is surely not applicable. (One can say, DIS has a higher priority than UN1 and UN2.)

If the inequality is not yet proven or disproven, we have to go on to check for trivial cases. This is done from line 5 down. If a trivial case is verified, the algorithm terminates yielding the corresponding result.

The non-trivial case, represented in lines 7 to 20 of the algorithm, extracts the leading names from r , and computes all partial derivatives of $r <: s$ for all leading names of r . This is done in the **for**-loop in line 12, which represents the implementation of UN1 and UN2.

Having finished this step, the inequality $r <: s$ is completely processed by the algorithm and is marked as ‘previously analyzed’ by insertion into an assumption set A , which is empty in the beginning. Note that $r <: s$ is only added to A if no trivial case could be verified. Hence, successively, all analyzed non-trivial inequalities are collected in A .

It is important to collect only inequalities that do not evaluate trivially. Inequalities like

$$int <: string$$

evaluate trivially to *FALSE* because none of the leading names of *int* occurs in *string*. Therefore, no partial derivatives can be computed, and hence, the inequality evaluates to *FALSE* in accordance to case T10. Adding such trivial inequalities to A causes evaluating them trivially to *TRUE* by applying case T2 if they occur again during the derivation process. In many cases this will lead to an incorrect result¹. Only non-trivial inequalities with a non-empty set of partial derivatives are added to A . Trivially evaluating inequalities must not be analyzed by case T2 and therefore not be added to A .

After updating A , in line 19 the algorithm is recursively applied to each inequality in the set of partial derivatives by checking each disjunction in this set. Hence, the algorithm only returns *TRUE*, if no inconsistent disjunction of partial derivatives is computed.

3.9 Completion of the Running Example

Now we will apply the algorithm to the example of r and s as introduced. As we have seen when we considered leading names, the first invocation of the algorithm leads to $names \equiv \{foo\}$ and $A \equiv \{r <: s\}$. We have previously shown the result of the computation of partial derivatives. The result of the redefined version of partial derivatives is:

$$\begin{aligned} \partial_{foo}(r) &\equiv \{\langle str, (\mathbf{bar}[int] \cdot \mathbf{foo})^* \rangle\} \\ \partial_{foo}(s) &\equiv \{\langle str, (\mathbf{bar}[int] \cdot s) \rangle, \langle str, \varepsilon \rangle\} \end{aligned}$$

Thus, we have to check if the following holds:

$$\langle str, (\mathbf{bar}[int] \cdot \mathbf{foo})^* \rangle <: (\langle str, (\mathbf{bar}[int] \cdot s) \rangle \mid \langle str, \varepsilon \rangle)$$

¹This is an error in the algorithm presented in [KeLi03]: The version of the algorithm shown there adds inequalities to A even if they have no partial derivatives.

According to [Hos00] (cf. Section 3.6) the derivation of this inequality produces the following 8 inequalities. They are pairwise connected by the boolean operator \vee .

$$\begin{aligned}
\text{pd} \equiv & \{(str <: str \mid str) \vee & (1) \dots T \\
& ((\mathbf{bar}[int] \cdot \mathbf{foo}[str])^* <: \emptyset), & (2) F \\
(str <: str) & \vee & (3) T \\
& ((\mathbf{bar}[int] \cdot \mathbf{foo}[str])^* <: \varepsilon), & (4) F \\
(str <: str) & \vee & (5) T \\
& ((\mathbf{bar}[int] \cdot \mathbf{foo}[str])^* <: (\mathbf{bar}[int] \cdot s)), & (6) F \\
(str <: \emptyset) & \vee & (7) F \\
& ((\mathbf{bar}[int] \cdot \mathbf{foo}[str])^* <: (\mathbf{bar}[int] \cdot s) \mid \varepsilon)\} & (8) ?
\end{aligned}$$

Now we have to analyze the derived inequalities. Inequality (1) does not evaluate trivially but leads to trivial inequalities in its partial derivatives and leads *TRUE* in a new recursive invocation. It is up to the reader to track the derivation of (1). The inequalities (3) and (5) are trivially *TRUE* by case T3. (2) is *FALSE* by case T5, (4) is also *FALSE* by case T8 and (6) is *FALSE* by case T1. So the first three disjunctions are *TRUE*. (7) does not hold by case T5, so (8) has to be checked in a new recursion of the algorithm. The next recursion starts with:

$$\begin{aligned}
r' & \equiv (\mathbf{bar}[int] \cdot \mathbf{foo}[str])^* \\
s' & \equiv (\mathbf{bar}[int] \cdot s) \mid \varepsilon
\end{aligned}$$

This leads to:

$$\text{names} \equiv \{\mathbf{bar}\} \quad \text{and} \quad A \equiv \{(r <: s), (r' <: s')\}$$

The following partial derivatives are generated in the course of the algorithm (it is up to the reader to reconstruct their whole derivation process):

$$\begin{aligned}
\text{pd} \equiv & \{(int <: int \vee & (1) T \\
& r <: \emptyset), & (2) F \\
(int <: \emptyset & \vee & (3) F \\
& r <: s)\} & (4) T
\end{aligned}$$

Inequality 4 is *TRUE*, because it is already in our set of assumptions:

$$r <: s \in A$$

The result contains no inconsistent partial derivative, so $r <: s$ is accepted as correct and the result of the computation is *TRUE*.

The complete derivation process of the running example is contained in the logfile enclosed in Appendix A.

Chapter 4

JAVA Class Design

4.1 The Hierarchy of Types

The type system of XQuery that has to be represented by the subtyping algorithm, consists of 8 different kinds of types:

- 1) Letters $\alpha \in \mathcal{A}$: \emptyset, ε , XML node types, primitive types
- 2) Regular operations: concatenation \cdot , alternation $|$, iteration $\{n, m\}$
- 3) Named types

All 8 kinds are represented by public subclasses of a common public superclass `RType`.

The none type \emptyset is represented by class `RNoneType` and the empty type ε is represented by class `REmptyType`.

In a non-prototypical real world implementation, the classes `RNodeType` representing XML node types, and `RPrimitiveType` representing XML Schema primitive types would be superclasses for the standard node types in XML and the XQuery primitive types. Because there are no relevant differences between the kinds of node types which may affect the subtyping process, all node types are represented by class `RNodeType`. For the same reason class `RPrimitiveType` represents all primitive types. The implementation of these types is completely extensible and has not to be recompiled after adding classes for all node types and primitive types.

Named types are represented by class `RNameType`.

The three regular operations concatenation, alternation and iteration are implemented in subclasses `RConcatenationType`, `RAlternationType` and `RIterationType`. Class `RIterationType` represents every form of iterated expression as follows.

min/max	symbol	constructor call
0 - ∞	e^*	<code>new RIterationType(e)</code>
1 - ∞	e^+	<code>new RIterationType(e, 1, RName.INFINITY)</code>
0 - 1	$e^?$	<code>new RIterationType(e, 0, 1)</code>
$n - m$	$e\{n, m\}$	<code>new RIterationType(e, n, m).</code>

Except class `RPrimitiveType` and class `RNodeType`, all classes discussed so far are declared as `final`.

The 8 classes representing the 8 kinds of types are referred to as “type classes”. Instances of type classes are sometimes referred to as “type instances”.

All type classes extend their common abstract superclass

```
public abstract class RType.
```

Class `RType` contains two protected members `child1` and `child2` of type `RType` with corresponding public getter- and setter-methods, which are all inherited by the subclasses of `RType`. The use of dynamic binding on these two members enables the representation of complex types using only a few type classes. Instances of type classes can recursively own instances of other type classes. This technique is a simple variant of the so-called “composite pattern” (cf. [GHV95]).

Classes `RNameType`, `RNodeType` and `RIterationType` use only one of their members. Classes `RNoneType`, `REmptyType` and `RPrimitiveType` do not use any member and classes `RConcatenationType` and `RAlternationType` use both members to represent types, since the latter classes represent the binary regular constructors ‘.’ and ‘|’. Class `RIterationType` has two additional members of type `int` which represent the minimal and maximal number of occurrences of the iterated type.

In every subclass of `RType`, setter-methods for unused members are overridden by an empty implementation. Hence it is impossible to assign types to members that have no meaning.

Alternations and concatenations consisting of more than two members are represented by instances of `RAlternationType` or `RConcatenationType` which own other `RAlternationTypes` or `RConcatenationTypes`. In general, complex types are represented by instances of type classes that own other type instances.

The following example shows the representation of type

$$r \equiv \text{foo}[\text{str}] \cdot (\text{bar}[\text{int}] \cdot \text{foo})^*$$

by the subtyping algorithm.

Example: Concrete representation of type $r \equiv \text{foo}[\text{str}] \cdot (\text{bar}[\text{int}] \cdot \text{foo})^*$.

```
RType r = new RConcatenationType(
    new RNodeType("foo",
        new RPrimitiveType("str")),
    new RIterationType(
        new RConcatenationType(
            new RNodeType("bar",
                new RPrimitiveType("int")),
            new RNodeType("foo",
                new RPrimitiveType("str"))
        )
    )
);
```

Recursive occurrences of named types are represented by an instance of `RNameType` with member `child1 == null`. The name of the occurrence has to be equal to the name of the top-level named type of the occurrence.

Class `RType` defines the following abstract methods implemented in the subclasses:

```
public abstract boolean isNullable();
public abstract Set leadingNames();
public abstract boolean checkTailPosition(RName root,
                                         boolean flag);
public abstract boolean checkNonNullableHead(boolean flag);
public abstract RType unfold(Hashtable nameTable);
public abstract boolean equals(RType r);
public abstract RType toString();
```

Method `boolean isNullable()` represents the implementation of function *nullable*. It yields *TRUE* if the type instance on which it is called is nullable, otherwise *FALSE*.

Method `Set leadingNames()` implements function *ln*. It returns the set of leading names of the type instance. If there are no leading names, an empty set is returned.

Methods `boolean checkTailPosition()` and `checkNonNullableHead()` implement functions *tp* and *nh*. They check the type instance recursively if it fulfills the two wellformedness constraints.

Method `RType unfold()` is used to ensure correct derivation of recursive types by replacing every recursive occurrence by its definition.

Methods `boolean equals()` and `String toString()` are service methods for comparing and printing types. Method `equals()` checks for structural equivalence of types. Two structural equivalent named types, even if their names differ, are analyzed as equal, regardless of being recursive or not.

Some behaviour is identical in more than one type class. This behaviour is represented by methods of class `RType` that are not abstract. These methods are overridden by some type classes and inherited unmodified by most other type classes:

```
public Set getPartialDerivatives(Set leadingNames);
public RType content();
public RType concatenate(Hashtable nameTable);
public boolean isWellformed();
```

Method `Set getPartialDerivatives(Set leadingNames)` represents the implementation of $\partial_\alpha(t)$. The code in `RType` implements rule LF3 for the computation of linear forms with the restriction to add only those elements to the resulting set, whose first elements are of identical name with the leading names passed to the method as parameter. This rule is inherited by `RNodeType` and `RPrimitiveType`. All other type classes override this method with own implementations of the corresponding rule for the computation of linear forms.

Method `RType content()` implements function *cn*. It yields the content of the type instance if *cn* is defined on the type, otherwise an exception is thrown. Its standard implementation in class `RType` throws an exception. Only classes `RNodeType` and `RPrimitiveType` override `content()` with non-trivial implementations in accordance to the definition of *cn*.

Method `RType concatenate(RType r)` implements rule CL6 for concatenation of linear forms. This rule is used by all type classes except `RNoneType` which implements rule CL4 and `REmptyType`, which implements rule CL5.

For more convenient wellformedness checking, class `RType` provides method `boolean isWellformed()` which is a wrapper for `checkTailPosition()` and `checkNonnullableHead()`. It is inherited unmodified by all type classes and provides the interface for wellformedness checking on type classes.

4.2 Names

Class `public final class RName` provides an interface for the representation of names. Names can be the names of primitive types like `int`, XML node types or the names of named types. Class `RName` works as a wrapper for class `java.lang.String` and provides several standard operations like check for equality with another name or returning the internal string representation.

Additionally `RName` defines some numeric or literal constants as `static` members like `RName.INFINITY` or `RName.NULL`. Using these constants provides integrity of changes with minimal effort.

4.3 Sets

In the course of the algorithm several representations of sets are important. To represent the leading names of a regular expression, sets of names are needed. Partial derivatives of regular expressions are represented by sets of type pairs. Partial derivatives of regular inequalities consist of sets of regular inequalities. Method `prove()` uses a set of regular inequalities as assumption set.

Sets are represented by class `public final Set`. `Set` aggregates instances of classes implementing interface `SetElement`, which declares only method

```
public boolean equals(SetElement e)
```

to ensure comparability of elements in the `Set`.

Interface `SetElement` is implemented by class `RName` as well as by class `TypePair` which represents ordered pairs of types (the representation of type pairs will be discussed in Section 4.4). Class `Inequality` which represents regular inequalities is also a `SetElement` (discussed in Section 4.5). The implementation of interface `SetElement` makes these three classes aggregable by `Set`.

Class `Set` internally declares a private member of type `java.util.HashSet`, which performs the aggregation of elements.

`Set` defines standard operations for inserting elements, removing elements, union with another `Set`, check for containment of a single element, check for emptiness, and returning its cardinality. Insertion and union operations ensure the elimination of duplicates.

Removing an element works by reference and also by equality. If method

```
boolean remove(SetElement e)
```

does not find the reference `e` in the `Set` but an element, that is equal to `e`, it removes this element.

The containment check works in the same way. Containment of an element is checked by reference as well as by equality: Method

```
boolean contains(SetElement e)
```

tests for a containment by reference of element `e` using the facilities of class `java.util.HashSet`. If the result is *FALSE*, a check for equality is performed on each element, using method `boolean e.equals()` declared by `SetElement` with every element in the `Set` as input parameter.

A special feature of class `Set` is a partial implementation of concatenation of linear forms. `Set` implements interface `Concatenable` which declares method

```
public Set concatenate(RType r).
```

`Concatenable` is the interface for concatenation facility (for details see Section 5.6.)

A legal application of concatenation rules is only possible, if the `Set` instance on which the method is called, has up to two elements which are type pairs. The input to this method is a `RType r` and the output is a `Set` containing up to two type pairs.

The implementation of `concatenate()` in `Set` checks if either rule CL3 or rule CL7 is applicable. If rule CL3 is applicable, an empty `Set` is returned. If rule CL7 is applicable, the implementation of `concatenate()` of class `TypePair` is called for each instance of `TypePair` in the `Set`. (Details of concatenation of linear forms are discussed in Section 5.6.)

Method `int size()` returns the number of elements in the `Set` and method `boolean isEmpty()` returns *TRUE* if the `Set` does not contain any elements. Method `clear()` removes all elements from the `Set`, `clone()` returns a shallow copy of the `Set` and `String toString()` returns a literal representation of the `Set`.

The elements in a `Set` are unordered. They can only be manipulated by the methods discussed in this section. If an representation of the `Set` is needed that provides an order of elements (as needed to process the partial derivatives of regular inequalities), method `toArray()` is used which returns an array that holds all set elements. If there are no elements in the `Set`, method `toArray()` returns `null`.

4.4 Type Pairs

Linear forms and partial derivatives of regular expressions consist of sets of type pairs. A single type pair is modeled by class `TypePair`. `TypePair` has two members `t1` and `t2` of type `RType` with corresponding setter and getter methods. Since `TypePair` is a `SetElement`, it defines `public boolean equals(SetElement e)` which is a wrapper for

```
public boolean equals(TypePair p).
```

It returns *TRUE* if the elements of `p` are equal to the elements of the `TypePair` instance. To analyze if this is true, method `equals()` is recursively called on `t1` and `t2` with `p.t1` and `p.t2` as parameter.

Class `TypePair` implements also interface `Concatenable` and therefore defines method

```
public Set concatenate(RType r)
```

for rules CL1 and CL2. If none of them is applicable, method `public RType concatenate(RType r)` is called on the second element of the pair with type `r` as input parameter (for details of concatenation of linear forms see Section 5.6).

4.5 Regular Inequalities

The interface for using the subtyping facility is class `Inequality` which models a regular inequality $r <: s$.

Class `Inequality` has two members of type `RType` representing the types `r` and `s` and defines method

```
public Inequality[] getPartialDerivatives(Set names)
```

to compute the partial derivatives of the inequality for names `names`. Internally it calls `getPartialDerivatives()` on `r` and `s`. The method computes the partial derivatives of the inequality based on the set theoretic observation of [Hos00] discussed in Section 3.6. The result is an array *A* of instances of `Inequality` that represents a set of disjunctions such that between an even and an odd index a disjunction operator is assumed. Between an odd and an even index a conjunction operator is assumed (for details see Section 6.1).

Class `Inequality` also defines method

```
public boolean prove()
```

which performs the subtyping of the two types by passing them to the recursive worker method

```
private boolean wprove(Set assumptionSet, Counter calls).
```

Method `wprove()` contains the implementation of the subtyping algorithm. To provide a logging facility, the recursive calls of `wprove()` are counted. The recursive structure of this method requires calls to be counted by a counter that is passed by reference. This is done by an internal class `Counter` of class `Inequality`.

Method `wprove()` uses method `getPartialDerivatives()` for computing the partial derivatives of the inequality. The output of `wprove()` is passed to be the output of `prove()`, which is *TRUE* or *FALSE* in accordance to the validity of $r <: s$.

Each `Inequality` can be configured with its own logging policy and own evaluation mode, i.e., for each `Inequality` short circuit evaluation of the recursive calls of `wprove()` can be turned on or off. The logging facility includes a class that does the logging for later debugging sessions. Each `Inequality` can be configured with its own logging class instance. For an example of the logging facility see Appendix A.

4.6 Exceptions

To be able to explore errors occurring during the derivation process more easily, some exceptions are defined that can be thrown on several occasions during the derivation.

Class `TypeException` declares an abstract superclass for three exceptions: A `NoWellformedTypeException` occurs if the check for wellformedness on a given type fails. This exception is thrown by the check methods implementing the wellformedness constraints, by method `isNullable()` and also by method `getPartialDerivatives()` if situations occur which are impossible to occur if the constraints are met.

An `IncompleteTypeException` is thrown whenever a type is not constructed properly, e.g., if a `RAlternationType` has only one member or a `RNameType` has no name. This exception is only thrown by the constructors of such type classes where the possibility to make mistakes exists.

An `IrregularContentRequestException` is thrown if method `content()` is called on a type on which *cn* is not defined.

There exist also two exceptions beside this hierarchy: `IllegalConcatenationException` is thrown by every implementation of method `concatenate()` which is declared in interface `Concatenable`. It indicates that a concatenation was tried to be performed on illegal input, i.e., on a `Set` with more than two elements.

A `BitArrayIndexException` is thrown by class `BitArray` whenever an element is tried to be accessed which exceeds the array bounds. Class `BitArray` is discussed in the course of Chapter 6.

Chapter 5

Implementation of Auxiliary Functions

5.1 Check for Wellformedness

The check for wellformedness of types is performed by methods

```
boolean checkTailPosition(RName rootName, boolean flag)
```

and

```
boolean checkNonnullableHead(boolean flag).
```

Method `checkTailPosition(RName rootName, boolean flag)` performs the check if all recursive occurrences of named types are in tail positions within concatenations. It implements function *tp* from Section 3.3.

Method `checkNonnullableHead(boolean flag)` checks if all recursive occurrences are preceded by a non-nullable term. It implements function *nh* from Section 3.3.

Both methods are declared as `abstract` in class `RType`. Each type class implements the corresponding rule according to the definition of *tp* and *nh*.

Example: Method `checkNonnullableHead()` in class `RNameType` (rule NH5).

```
1 public boolean checkNonnullableHead(boolean flag) {
2
3     if (this.child1 != null)
4         return this.child1.checkNonnullableHead(false);
5
6     return flag;
7 }//checkNonnullableHead
```

If the instance does not represent a recursive occurrence (i.e., `(this.child1 != null)`), the check is invoked on the definition of the type, otherwise, the flag is the result. This is the implementation of rule NH5.

5.2 Unfolding Recursive Types

The check for wellformedness of types prevent the occurrence of undefined situations in the course of the derivation process by preventing the application of methods `getPartialDerivatives()` and `isNullable()` on recursive occurrences. But the wellformedness constraints do not prevent a recursive occurrence of a named type occurring as one of the member types of a regular inequality.

Method `getPartialDerivatives()` in class `RNameType` invokes method `getPartialDerivatives()` on the inner member `child1` of the instance, which represents the definition of the named type. If `getPartialDerivatives()` is invoked on some `RNameType` having `child1 == null` it comes to an undefined situation and a `NoWellformedTypeException` is thrown.

Rules LF5, LF7 and CL5 cause recursive occurrences of named types to become a possible second component of an ordered pair in the set of partial derivatives. Hence, recursive occurrences of named types can be the input of method `prove()`.

Example: Consider the linear form of named type $T1 : \text{int} \cdot T1 \mid \varepsilon$ as defined in Section 3.3. It fulfills both wellformedness constraints and is equivalent to int^* .

$$\begin{aligned}
 lf(T1) & \\
 & \stackrel{\text{LF4}}{=} lf(\text{int} \cdot T1 \mid \varepsilon) \\
 & \stackrel{\text{LF7}}{=} lf(\text{int} \cdot T1) \cup lf(\varepsilon) \\
 & \stackrel{\text{LF2}}{=} lf(\text{int} \cdot T1) \cup \emptyset \\
 & \stackrel{\text{LF5}}{=} lf(\text{int}) \odot T1 \\
 & \stackrel{\text{LF3}}{=} \{\langle \text{int}, \varepsilon \rangle\} \odot T1 \\
 & \stackrel{\text{CL5}}{=} \{\langle \text{int}, T1 \rangle\}
 \end{aligned}$$

It is clear that the definition of partial derivatives of regular inequalities will lead to $r' \equiv T1$ for some inequalities $r' <: s' \in \partial_{ln(r)}(r <: s)$ and therefore to a recursive occurrence as input type for method `prove()`.

For correct derivation of named types, we have to ensure that in each input of each invocation of the subtyping algorithm the recursive occurrences of named types are replaced by their definition. This technique is called “unfolding”. Without unfolding, in case of a recursive occurrence as input type, method `prove()` would face a named type without definition and run into an undefined situation.

We define function *unfold*: $\mathbf{Reg}[\mathcal{A}] \rightarrow \mathbf{Reg}[\mathcal{A}]$ for replacing any recursive

occurrence in a named type by its definition. For all $e \in \mathbf{Reg}[\mathcal{A}]$ we define:

$$\mathit{unfold}(\emptyset) := \emptyset \quad (\text{UF1})$$

$$\mathit{unfold}(\varepsilon) := \varepsilon \quad (\text{UF2})$$

$$\mathit{unfold}(\mathbf{n}[e]) := \mathbf{n}[\mathit{unfold}(e)] \quad (\text{UF3})$$

$$\mathit{unfold}(p) := p \quad (\text{UF4})$$

$$\mathit{unfold}(n : \mathbf{null}) := n : \mathit{def}(n) \quad (\text{UF5})$$

$$\mathit{unfold}(n : \mathit{def}(n)) := n : \mathit{unfold}(\mathit{def}(n)) \quad (\text{UF6})$$

$$\mathit{unfold}(e_1 \cdot e_2) := \mathit{unfold}(e_1) \cdot \mathit{unfold}(e_2) \quad (\text{UF7})$$

$$\mathit{unfold}(e_1 \mid e_2) := \mathit{unfold}(e_1) \mid \mathit{unfold}(e_2) \quad (\text{UF8})$$

$$\mathit{unfold}(e \{n, m\}) := \mathit{unfold}(e) \{n, m\} \quad (\text{UF9})$$

Simple types need not to be unfolded. Rule UF5 handles recursive occurrences. De facto, it replaces the value assigned to `child1`, which is `null` in case of a recursive occurrence, by the definition of the named type. For composite types the call of $\mathit{unfold}()$ is simply distributed to its children (cf. rules UF7, UF8, UF9).

Class `RType` declares method

```
public abstract RType unfold(Hashtable nameTable)
```

This method is implemented in the type classes such that each type class implements the corresponding rule.

If $\mathit{unfold}()$ has to do changes to the type, the type is copied. This ensures that there are no changes done to any type instance during the prove process.

The implementation of method $\mathit{unfold}()$ in classes `RNoneType`, `REmptyType` and `RPrimitiveType` simply returns `this` as result in accordance to the definition of unfold for simple types.

Classes `RNodeType`, `RAlternationType`, `RConcatenationType` and `RIterationType` invoke $\mathit{unfold}()$ in accordance to the definition of unfold on their members and return the unfolded type. The return value is an unfolded copy of the original type.

The implementation of $\mathit{unfold}()$ in class `RNameType` implements unfolding rules UF5 and UF6.

The application of rule UF5 or rule UF6 is dependent on the check, if the named type denotes a recursive occurrence or not. If the type instance denotes a recursive occurrence, its definition has to be looked up in a name table which is passed as argument to $\mathit{unfold}()$. If the type instance is not a recursive occurrence, its definition has to be added to this name table. Thus, a later lookup for this name in the course of the unfolding process will yield the definition of the name.

Example: Method `unfold()` in class `RNameType` (rules UF5, UF6).

```
1 public RType unfold(Hashtable nameTable) {
2
3     RType unfoldedChild = null;
4
5     // UF5: if instance is a recursive occurrence, add its definition
6     if (this.child1 == null) {
7
8         RName name = null;
9         Iterator it = nameTable.keySet().iterator();
10
11         while (it.hasNext()) {
12             name = (RName)it.next();
13             if (this.getName().equals(name))
14                 break;
15         }
16
17         unfoldedChild = (RType)nameTable.get(name);
18
19     // UF6: if type is not a recursive occurrence, unfold
20     } else {
21
22         if (nameTable.containsKey(this.getName()) == false) {
23
24             nameTable.put(this.getName(), this.getFirstChild());
25         }
26
27         unfoldedChild = this.child1.unfold(nameTable);
28     }
29
30     return new RNameType(
31         new RName(this.name.toString()),
32         unfoldedChild
33     );
34 } //unfold
```

Only instances of `RNameType` have to be really unfolded. But on each input type of method `prove()`, method `unfold()` is called, because this allows to exclude knowledge about the nature of the types currently to be proven.

5.3 Function `nullable()`

The check for ε -inclusion is performed by method

```
public boolean isNullable(),
```

which is declared as `abstract` in class `RType` and implemented in each type class. Each implementation of `isNullable()` implements the corresponding rule of Section 3.1.

In classes `RNoneType`, `RNodeType` and `RPrimitiveType`, the implementation of method `isNullable()` simply returns *FALSE*.

In class `REmptyType`, method `isNullable()` returns *TRUE*.

The implementation of method `isNullable()` in the three classes `RAlternationType`, `RConcatenationType` and `RNameType` passes the invocation of `isNullable()` to their members in accordance to the rules in Section 3.1.

Class `RIterationType` represents a nullable type only if either the minimal number of occurrences for its inner type is 0 or its inner type is nullable.

Example: Method `isNullable()` in class `RIterationType` (rule NA8).

```

1 public boolean isNullable() {
2
3     return (this.minOccurs == 0) ? true : this.child1.isNullable();
4 }//isNullable

```

An iteration is nullable, only if the number of occurrences can be 0 or if a nullable regular expression is iterated.

5.4 Function `leadingNames()`

The leading names of a type are represented by an instance of class `Set` containing instances of `RName`. This set is returned by the implementations of method

```
public abstract Set leadingNames()
```

declared in class `RType`. The implementations of `leadingNames()` within the type classes represent function *ln* as defined in Section 3.2.

Method `leadingNames()` in classes `RNoneType` and `REmptyType` yields an instance of `Set` which does not contain any elements.

In classes `RAlternationType`, `RConcatenationType`, `RIterationType` and `RNameType` method `leadingNames()` invokes `leadingNames()` recursively on its members in accordance to the corresponding rules in Section 3.2.

Classes `RNodeType` and `RPrimitiveType` implement `leadingNames()` by returning a `Set` containing their `RName` member. Hence they are the only classes which add names to the set of leading names in accordance to the definition of *ln*.

Example: Method `leadingNames()` in class `RConcatenationType` (rule LN6).

```

1 public Set leadingNames() {
2
3     Set result = this.child1.leadingNames();
4
5     try {
6         if (this.child1.isNullable()) {
7             result = result.union(this.child2.leadingNames());
8         }
9     } catch (TypeException te) {
10
11         ... // exception handling
12     }
13     return result;
14 }//leadingNames

```

5.5 Function `content()`

The implementation of function `cn` is provided by method

```
public RType content()
```

Because function `cn` is only defined for letters, only the two classes `RNodeType` and `RPrimitiveType` contain a substantial implementation of `content()`, in accordance to the definition of `cn` in Section 3.5.

The implementation of method `content()` in class `RNodeType` returns the first member `child1` or – if the node is empty – an instance of `REmptyType`. The implementation in class `RPrimitiveType` returns a new instance of `REmptyType`.

The implementation of all other type classes does not perform a computation but only throws an `IrregularContentRequestException`.

5.6 Concatenation of Linear Forms

The recursive definition of linear forms involves a binary operation \odot which applies the concept of concatenation to linear forms (cf. Section 2.2). The definition of this operation does not consist of concatenation rules for single types, therefore it cannot be implemented by inheritance in the type classes.

The concatenation of linear forms affects sets, pairs and types. Hence, the implementation of concatenation of linear forms is distributed over the three classes that are involved in concatenation operations: `Set`, `TypePair` and `RType`.

Inheritance of method `RType concatenate(RType r)` by class `RType` and implementation of interface `Concatenable` declaring method

```
public Set concatenate(RType r)
```

enables distributing the functionality of concatenation over the type hierarchy and classes `Set` and `TypePair`. Interface `Concatenable` is implemented by classes `Set` and `TypePair` to provide access to the concatenation facility.

A concatenation operation “adds” a regular expression to a linear form. Since linear forms are represented by `Sets`, the top-level call for a concatenation will be on class `Set` passing parameter `r` of type `RType`. The definition of `concatenate()` in class `Set` implements rules CL3 and CL7. The implementation checks if the `Set` contains up to two elements. For an empty `Set`, rule CL3 is applied and for a `Set` containing two elements, rule CL7 is applied. If the `Set` contains more than two elements, an `IllegalConcatenationException` is thrown.

Example: The implementation of `Set concatenate(RType r)` in class `Set`.

```
1 public Set concatenate(RType r)
2 throws IllegalConcatenationException {
3
4     Iterator it;
5     Concatenable c1 = null;
6     Concatenable c2 = null;
7
8     switch (this.size()) {
9
10    // rule CL3
11    case 0:
12
13        return this;
14
15    // rule CL1, CL2, CL4, CL5, CL6
16    case 1:
17
18        it = this.iterator();
19        try {
20
21            c1 = (Concatenable)it.next();
22        }
23        catch (ClassCastException cce) {
24            throw new IllegalConcatenationException("No Concatenables.");
25        }
26        return c1.concatenate(r);
27
28    // rule CL7
29    case 2:
30
31        it = this.iterator();
32        try {
33
34            c1 = (Concatenable)it.next();
35            c2 = (Concatenable)it.next();
36
37        } catch (ClassCastException cce) {
38            throw new IllegalConcatenationException("No Concatenables.");
39        }
40
41        Set result = c1.concatenate(r);
42        result = result.union(c2.concatenate(r));
43        return result;
44
45    default:
46
47        throw new IllegalConcatenationException("No Concatenables.");
48
49    } //switch
50 } //concatenate
```

If the `Set` has one element, `concatenate()` is called on this element. Type `r` is passed to this call. The only `Concatenable` which is also a `SetElement` and can therefore be found in `Sets` is class `TypePair`. If the element is not of type `Concatenable`, an `IllegalConcatenationException` is thrown.

The definition of `concatenate()` in class `TypePair` implements rules CL1 and CL2. If `r` is not an instance of `RNoneType` or `REmptyType`, none of these two rules are applicable. In this case, the call is passed to the second element in the `TypePair`.

Example: The implementation of `Set concatenate(RType r)` in class `Set`.

```

1 public Set concatenate(RType r) {
2
3     // rule CL1
4     if (r instanceof RNoneType)
5         return new Set();
6
7     // rule CL2
8     if (r instanceof REmptyType)
9         return new Set((SetElement)this);
10
11    // rule CL4, CL5, CL6
12    return new Set(
13        (SetElement)
14        new TypePair(this.getFirstElement(),
15                    this.getSecondElement().concatenate(r)
16                    );
17    );
18 }//concatenate

```

The elements of a `TypePair` are instances of subclasses of `RType` and therefore own an implementation of method

```
public RType concatenate(RType r).
```

Method `concatenate()` is defined in class `RType`, which inherits the implementation of rule CL6 as standard implementation to every type class. Method `RType concatenate()` is only overridden in class `RNoneType` where it implements rule CL4 and in class `REmptyType` where it implements rule CL5.

The result of the call of `RType concatenate()` on the second element of a `TypePair` will be a new instance of `RType` which represents the result of the concatenation. This instance is the new second element of the `TypePair` and thus the concatenation is completed.

5.7 Partial Derivatives of Types

Partial derivatives of types are sets of type pairs and are consequently modelled as instances of `Set` containing instances of `TypePair`.

Because $\partial_\alpha(t)$ for every $t \in \mathbf{Reg}[\mathcal{A}]$ is a subset of the set of linear forms of t , we use the linear form to compute the partial derivatives of type t . Method

```
public abstract Set getPartialDerivatives(Set names)
```

declared in class `RType` is implemented in the subclasses of `RType`. Each type class implements the corresponding linear form computation rule.

Rule LF1 is implemented by class `RNoneType`. An invocation of method `getPartialDerivatives()` in this class will yield an empty `Set`. Similarly, the implementation of `getPartialDerivatives()` in class `REmptyType` yields an empty `Set`, in accordance to corresponding rule LF2.

The rules for iteration (rule LF8) and alternation (rule LF7) of linear forms are implemented by classes `RIterationType` and `RAlternationType`. The two rules for concatenation of linear forms (rules LF5 and LF6) are both implemented by `RConcatenationType`.

Method `Set getPartialDerivatives(Set names)` in class `RNameType` returns the partial derivatives of the first member. If the first member is `null`, it throws a `NoWellformedTypeException` because wellformedness constraint 2 must have been violated by the type. This implements rule LF4.

Classes `RNodeType` and `RPrimitiveType` implement rule LF3 with a small modification. Remember that $\partial_\alpha(t)$ is the subset of $lf(t)$ consisting of all $\langle n, e \rangle \in lf(t)$ with $n = \alpha$ and $e \neq \emptyset$. Rule LF3 is the only rule which is not recursive and generates a pair. In the implementation of rule LF3 in classes `RNodeType` and `RPrimitiveType` only those pairs are added to the result, whose first element is contained in input set `names` and whose second element is not \emptyset . Hence the result of an invocation of method `getPartialDerivatives()` on a type t for a set of names $\mathcal{N} \subseteq \mathcal{A}$ yield:

$$\{ \langle n, e \rangle : \langle n, e \rangle \in lf(t), n \in \mathcal{N}, e \neq \emptyset \}$$

It is obvious that this is the definition of the partial derivatives $\partial_\alpha(t)$.

The modified version of rule LF3 is implemented as the standard rule in class `RType` and inherited by `RNodeType` and `RPrimitiveType`. The other type classes override method `getPartialDerivatives()`.

Example: Method `getPartialDerivatives()` in class `RType` (rule LF3).

```

1  public Set getPartialDerivatives(Set names) {
2
3      Set result    = new Set();
4      RType content = null;
5
6      if (names != null && names.contains(this.getName())) {
7
8          try {
9
10             result = new Set(
11                 (SetElement)
12                 new TypePair(this.content(), new REmptyType())
13             );
14
15         } catch (IrregularContentRequestException icre) {
16             // ... exception handling
17         }
18
19     } //if
20
21     return result;
22 } //getPartialDerivatives

```

The code consists of an invocation of `content()` on the first element of the resulting type pair (in accordance to the description in Section 3.5) and the generation of a new instance of `REmptyType` as second element. The pair will only be part of the result if its name is contained in `names`. This modification ensures that not the whole linear form will be returned but only the partial derivatives. Hence each invocation of `getPartialDerivatives()` yields the partial derivatives of the type class instance which the method was called upon.

Chapter 6

Implementation of the Subtyping Algorithm

6.1 Partial Derivatives of Regular Inequalities

As stated in Section 3.5 partial derivatives $\partial_\alpha(r <: s)$ of a regular inequality $r <: s$ with $n = |\partial_w(r)|$ and $m = |\partial_w(s)|$ can be described as a set of disjunctions as follows for all $\langle a, b \rangle \in \partial_\alpha(r)$ and all $\langle c, d \rangle \in \partial_\alpha(s)$:

$$a_i <: (\bigvee_{j \in J} c_j) \quad \vee \quad b_i <: (\bigvee_{j \in \bar{J}} d_j).$$

for each $i \in \{1, \dots, n\}$ where $J = \{1, \dots, m\}$ and $\bar{J} = \{1, \dots, m\} \setminus J$.

Because an inequality φ evaluates to *TRUE* only if there are no disjunctions derivable from φ that are *FALSE*, $\partial_\alpha(r <: s)$ can also be interpreted as a conjunction of disjunctions.

The partial derivatives of a regular inequality are computed by method

```
public Set getPartialDerivatives(Set names)
```

in class `Inequality`. By this design, inequalities as well as types can compute their own partial derivatives.

To generate the conjunction of all disjunctions contained in the partial derivatives of an inequality $r <: s$ several steps have to be done:

1. Compute partial derivatives of **r** for all leading names of **r**.
2. Compute partial derivatives of **s** for all leading names of **r**.
3. Compute power set Λ of all left elements of all type pairs in the partial derivatives of **s**. Transform each element $\lambda_i \in \Lambda$ into an alternation expression c_i constructed from all regular expressions contained in λ_i . The result will be set $\mathcal{T}_{left} \equiv \{c_0, \dots, c_{2^m-1}\}$ with $c_0 \equiv \emptyset$.
4. Compute \mathcal{T}_{right} analogously for the right elements of all type pairs in the partial derivatives of **s**: Transform each element ϱ_i of the power set R of all right elements in the partial derivatives of **s** into an alternation expression d_i constructed from all regular expressions contained in ϱ_i .

5. Construct disjunction clauses with inequalities in accordance to the description above in this section.
6. Construct conjunction of these disjunctions.

Steps 1 and 2 are implemented by calling `getPartialDerivatives()` with `r.leadingNames()` on input types `r` and `s` as actual parameter.

We transform the resulting sets into arrays because we need a well-defined order of all elements.

We will represent the partial derivatives of a regular inequality as an array of inequalities. The size of this array will be the number of inequalities in the partial derivatives. Hence, the size of the result array is $n * 2^{m+1}$ with $n = |\partial_{ln(r)}(r)|$ and $m = |\partial_{ln(r)}(s)|$ as stated in Section 3.6. We declare and allocate this array after computing the partial derivatives of `r` and `s`.

To compute the resulting inequalities, a loop is used that iterates over the partial derivatives of `r`. Obviously in each iteration of this loop, a single pair of $\partial_{ln(r)}(r)$ is analyzed and yields two types: its first element a_i and its second element b_i for all $i \in \{0, \dots, n\}$.

Furthermore in each iteration, a second loop iterates over the partial derivatives of `s` and performs steps 3 and 4 by constructing all types $c_{j \in J}$ and $d_{j \in \bar{J}}$ with $J = \{0, \dots, 2^m\}$ and $\bar{J} = \{0, \dots, 2^m\} \setminus J$.

Hence, each iteration of the inner loop constructs one single disjunction

$$a_i <: c_{j \in J} \quad \vee \quad b_i <: d_{j \in \bar{J}}$$

which is added to the result array.

When the outer loop terminates, the array is filled with all partial derivatives of `r <: s` (the code is shown in Section 6.2 in detail). The interpretation of this array as a conjunction of disjunctions has to be done by the caller. Thus, steps 5 and 6 have to be performed by method `prove()` that calls `this.getPartialDerivatives()`: Between $A[i]$ and $A[i + 1]$ for each even i a disjunction operator is assumed. Between $A[i + 1]$ and $A[i + 2]$ a conjunction operator is assumed. For an even number i we thus have:

$$\dots \wedge (A[i] \vee A[i + 1]) \wedge (A[i + 2] \vee A[i + 3]) \wedge \dots$$

Steps 3 and 4 require further explanation. They consist of two tasks: the computation of the power sets of all left and right elements in the type pairs of the partial derivatives of `s`, which will be described in Section 6.2, and the construction of the corresponding regular types, which will be described in Section 6.3.

6.2 Power Set Computation

Following the observations in Section 3.6, constructing all inequalities in $\partial_\alpha(r <: s)$ requires computation of the power set Λ of the set of all first elements of all pairs in $\partial_{ln(r)}(s)$ and of the power set R of the set of all second elements of all pairs in $\partial_{ln(r)}(s)$.

The strategy is not to compute both power sets and store them into primary memory, but only to enumerate successively all elements of Λ and R such that in each enumeration step for $\lambda_{j \in J} \in \Lambda$ the corresponding $\rho_{j \in \bar{J}} \in R$ is generated. In each enumeration step, we construct two types $c_{j \in J}$ and $d_{j \in \bar{J}}$ that are used as input types for the inequalities in the disjunction clause $a_i <: c_j \vee b_i <: d_j$ currently to be computed.

A power set $\mathcal{P}(S)$ of a set S is the set of all subsets of S . Hence for each element $\mu_i \in \mathcal{P}(S)$ for $i \in \{0, \dots, 2^{|S|} - 1\}$ each element $\sigma_k \in S$ for $k \in \{0, \dots, |S| - 1\}$ has one of two possible states. Either it holds that $\sigma_k \in \mu_i$ or $\sigma_k \notin \mu_i$. Thus, each μ_i can be expressed by a bit vector of length $|S|$ where each bit is assigned to one element of S . If a bit has value 1, the corresponding element of S is also an element of μ_i , otherwise the element is not an element of μ_i . Thus $\mathcal{P}(S)$ is represented by $2^{|S|}$ bit vectors, each with length $|S|$.

It is clear, that all these bit vectors can be enumerated by starting with a vector in which all bits are 0, interpreting it as a binary number and adding 1 to it till all bits have value 1. Thus it takes $2^m - 1$ additions to enumerate all possible bit vectors and therefore all $\mu_i \in \mathcal{P}(S)$.

For the representation of bit vectors class `BitArray` was generated. A `BitArray` is a representation of a boolean array that is represented by a number of bits with a convenient user interface that provides the standard access operations `getElement()`, `setElement()`, `setAll()` and `size()`. Methods `add()` and `subtract()` add or subtract 1 to the vector, interpreting it as a binary number.

The internal representation is done by an array of `int`. In JAVA, data type `int` has a size of 32 bit and is signed. For this reason, the manipulation of the single bits is done using a bit mask. We do not shift any bits within the `int` array itself.

The usage of `BitArray` enables subtyping of types with an arbitrary number of partial derivatives. In normal cases, the number does nearly never exceed 3. A simpler solution would be to represent the boolean array by a single instance of type `byte` whose size is 8 bit or type `short` whose size is 16 bit. For those low capacities the usage of a bitmask may not be necessary, because using a `short` provides representation facilities for up to 15 partial derivatives. The use of `BitArray` avoids giving an artificial limit for the number of partial derivatives.

One can understand the enumeration technique best by looking at the code.

Example: Method `getPartialDerivatives()` in class `Inequality`.

```

1 public Inequality[] getPartialDerivatives(Set names) throws
2     IllegalConcatenationException,
3     IncompleteTypeException,
4     NoWellformedTypeException {
5
6     // if we have leading names
7     if (names.isEmpty() == false && names != null) {
8
9         int m = 0;
10        int n = 0;
11        int index = 0;

```

```

12
13 //compute partial derivatives
14 Set pdLeftCl = this.r.getPartialDerivatives(names);
15 Set pdRightCl = this.s.getPartialDerivatives(names);
16
17 if (pdLeftCl.isEmpty() || pdRightCl.isEmpty())
18     return null;
19
20 //make arrays from sets of partial derivatives
21 SetElement[] leftClause = pdLeftCl.toArray();
22 SetElement[] rightClause = pdRightCl.toArray();
23
24 // m = |delta(r)|
25 m = (pdLeftCl.isEmpty()) ? 0 : leftClause.length;
26 // n = |delta(s)|
27 n = (pdRightCl.isEmpty()) ? 0 : rightClause.length;
28
29 // compute number of inequalities in partial derivatives
30 // = m * $2^(n+1)$
31 int numberOfIq = m * (int)(Math.pow(2, n+1));
32 if (numberOfIq < 1)
33     return null; //no inequalities, no resulting set
34
35 Inequality[] result = new Inequality[numberOfIq];
36
37 // synchronize length of bit patterns and type clauses
38 BitArray s1Subset = new BitArray(rightClause.length);
39 BitArray s2Subset = new BitArray(rightClause.length);
40
41 // iterate partial derivatives of r
42 for (int i = 0; i < m; i++) {
43
44     s1Subset.setAll(false); // type s1 starts as nothing
45     s2Subset.setAll(true); // type s2 starts with the disjunction
46                             // of all second elements of all pairs
47
48     // construct initial types r1, r2, s1, s2
49     RType r1 = (RType)
50         (((TypePair)leftClause[i]).getFirstElement());
51     RType r2 = (RType)
52         (((TypePair)leftClause[i]).getSecondElement());
53     RType s1, s2;
54
55     s1 = s1Subset.construct(rightClause, true);
56     s2 = s2Subset.construct(rightClause, false);
57
58     // iterate right side of partial derivatives
59     // (construct power set of first and second elements
60     // of right side)
61     for (int j = 0; j < Math.pow(2, n); j++) {
62
63         result[index] = new Inequality(r1, s1, this.logmanager);
64         index++;
65         result[index] = new Inequality(r2, s2, this.logmanager);
66         index++;
67
68         // modify the bitpattern

```

```

69         s1Subset.add();
70         s2Subset.subtract();
71
72         // construct new types
73         s1 = s1Subset.construct(rightClause, true);
74         s2 = s2Subset.construct(rightClause, false);
75
76         }// for j (right side iteration)
77
78     }// for i (left side iteration)
79
80     return result;
81
82     // no leading names, no resulting set
83 } else
84
85     return null;
86
87 }//getPartialDerivatives

```

Section 6.1 explained that partial derivatives of regular inequalities are computed by a loop over the partial derivatives of \mathbf{r} that contains a loop over the partial derivatives of \mathbf{s} which constructs the single disjunction clauses. Therefore, this inner loop has to perform the power set computation.

The inner loop over j declares two bit vectors `s1Subset` and `s2Subset`. Bit vector `s1Subset` starts with each bit being 0, the other vector `s2Subset` starts with each bit being 1. Then `s1Subset` enumerates all elements of Λ in ascending order, `s2Subset` enumerates the elements of R in descending order. Thus, each iteration of the inner loop generates one disjunction clause to be added to the resulting set of partial derivatives of $r <: s$.

6.3 Type Construction

Method `getPartialDerivatives()` calls method

```
public RType construct(SetElement[] pairs, boolean firstElem)
```

for translating the bit vector into a type.

This method is part of class `BitArray` and enables a `BitArray` to compute and construct the type corresponding to the bit pattern it represents using the elements of an array of type pairs.

The input of `construct()` is an array of type pairs, from which some pairs are chosen to construct a type of either their first or their second elements. The output of `construct()` is an alternation type.

Each index position in array `pairs` corresponds to the same index position in the `BitArray`. If the bit with index i has value 1, an element of the type pair with index i will be chosen as a disjunction member in the resulting alternation type.

If `construct()` uses the first or the second element of the type pair depends on the value of parameter `firstElem`: `TRUE` causes `construct()` to choose

the first element of the marked pairs, passing *FALSE* will result in choosing the second element.

So method `construct()` constructs another type in each run of the inner loop in method `getPartialDerivatives()` because the `BitArray` on which `construct()` is called is modified by a call of `add()` or `subtract()` in each loop run.

Example: Method `construct()` in class `BitArray`.

```

1  public RType construct(SetElement[] pairs, boolean firstElem)
2  throws BitArrayIndexException
3  {
4      // result type
5      RType ty = null;
6
7      // arrays of same length?
8      if (this.size() != pairs.length) {
9          // ... error handling
10     }
11
12     // if pattern contains no marks for insertion, return none
13     if (this.all(false)) return new RNoneType();
14
15     // step to first non-null type pair to insert
16     int i = 0;
17     while ( (i < this.size())
18         && (this.getElement(i) == false || ((pairs[i]) == null))) {
19
20         i++;
21     }//while
22
23     // if there is a non-null type pair to insert, start insertion
24     if (i < this.size()) {
25         try {
26             ty = (firstElem) ? (RType)
27                 ((TypePair)pairs[i]).getFirstElement() :
28                 (RType)
29                 ((TypePair)pairs[i]).getSecondElement();
30         } catch (NullPointerException npe) {
31             //...exception handling
32         }
33
34         // insert each marked type pair into the return type
35         for (int j=i+1; j<this.size(); j++) {
36
37             if (this.getElement(j) == true) {
38
39                 try {
40                     ty = (firstElem) ?
41                         // construct s1 from first element
42                         new RAlternationType(
43                             ty,
44                             (RType)((TypePair)pairs[j]).getFirstElement()) :
45
46                         // construct s2 from second element
47                         new RAlternationType(

```

```

48             ty,
49             (RType)((TypePair)pairs[j]).getSecondElement());
50
51         } catch (IncompleteTypeException ite) {
52             // cannot occur here
53         } //catch
54
55         } //if
56     } //for
57 } //if i
58
59     return ty;
60
61 } //construct

```

With method `construct()` being a part of class `BitArray`, we model the bit vector representation of subsets such that they translate themselves into types. This reflects the intention to keep class `RType` independent from class `BitArray`.

6.4 Trivial Case Checking

An important part of the implementation are the trivial case checks. Implementing the cases in Section 3.7, we avoid unnecessary containment tests on `r` and `s` by taking the order of the trivial case checks into consideration.

Examples:

1. Inequality $r <: \emptyset$ is *FALSE* for all $r \neq \emptyset$ (case T4). On the other hand $\emptyset <: \emptyset$ is *TRUE* (case T3). If we check case T3 before checking case T4 we need not to check `r` separately for being `∅`.
2. If we test for $\text{nullable}(r) \wedge \neg \text{nullable}(s) \rightarrow r <: s \vdash \text{FALSE}$ (case T1) before we check if $\text{ln}(r) \equiv \emptyset \rightarrow r <: s \vdash \text{TRUE}$ (case T9), we do not need to take any special care about cases like $(\emptyset \mid \varepsilon) <: \text{foo} \vdash \text{FALSE}$ while checking case T9.
3. The check for $\text{ln}(r) \equiv \emptyset \rightarrow r <: s \vdash \text{TRUE}$ (case T9) has to precede the check for $s \equiv \varepsilon \wedge \text{ln}(r) \neq \emptyset \rightarrow r <: s \vdash \text{FALSE}$ (case T8), otherwise we need to recognize cases like $\varepsilon^* <: \varepsilon \vdash \text{TRUE}$.

Each check for a trivial case is implemented in a corresponding private method of class `Inequality`. These methods are called by method `prove()`. Their purpose is to make the code of method `prove()` more readable and to make logging convenient.

The order of the calls of these methods is not arbitrary. Some methods like

```

boolean rNullableSNotNullable(),
boolean noLeadingNames() and
boolean sIsEpsilon()

```

make assumptions about previously tested cases.

The succession of trivial case checks in the implementation is as follows (performed sequentially).

1. $r \equiv s \vdash TRUE$ (case T3)
2. $r <: s \in A \vdash TRUE$ (case T2)
3. $r \equiv \emptyset \vdash TRUE$ (case T4)
4. $s \equiv \emptyset \vdash FALSE$ (case T5)
5. $nullable(r) \wedge \neg nullable(s) \vdash FALSE$ (case T1)
6. $r \equiv \varepsilon \vdash nullable(s)$ (cases T6 and T7)
7. $ln(r) \equiv \emptyset \vdash TRUE$ (case T9)
8. $s \equiv \varepsilon \vdash FALSE$ (case T8)
9. $\partial_{ln(r)}(r) \equiv \emptyset \vee \partial_{ln(s)}(s) \equiv \emptyset \vdash FALSE$ (case T10)

6.5 Implementation of Method `prove()`

Method `boolean prove()` in class `Inequality` returns `TRUE` only if the inequality $r <: s$ represented by the current instance evaluates to `TRUE`, otherwise `FALSE`. Method `prove()` does not have any formal parameters. This is to keep the user interface of class `Inequality` simple.

Internally, `prove()` calls a recursive worker method

```
private boolean wprove(Set assumptionSet, Counter counter)
```

which performs the derivation of the inequalities. This design is used because of the logging facility. A single recursive method `prove()` would not enable a really convenient logging.

The task performed by `prove()`, is to allocate a `Set` to collect the assumptions, to generate a `Counter` for counting the calls needed to derive `TRUE` or `FALSE` and to add the final lines to the logging output¹.

Method `wprove()` represents the implementation of the subtyping algorithm described in Section 3.8.

Example: Method `wprove()` in class `Inequality` (without logging and exception handling code).

```
1 private boolean wprove(Set assumptionSet, Counter calls)
2   throws NoWellformedTypeException,
3         IncompleteTypeException,
4         IllegalConcatenationException {
5
```

¹For an example of the logging facility see Appendix A.

```

6     // if one or both input types are null, stop
7     if ((this.r == null) || (this.s == null)) {
8         //... throw exception
9     }
10
11    // unfold types within inequality
12    Inequality iq = this.unfold();
13
14    // check wellformedness
15    this.isWellformed();
16
17    // perform trivial case checks 1 - 6
18    ...
19
20    // take leading names of r and partial derivatives of all these
21    // names on r <: s
22    Set lNames = iq.r.leadingNames();
23
24    // perform trivial case checks 7, 8
25    ...
26
27    Inequality[] pd = iq.getPartialDerivatives(lNames);
28
29    // 9.) no Partial Derivatives
30    if (noPartialDerivatives(pd, callID)) {
31        return false;
32    }
33
34    // if there are valid partial derivatives, add currently
35    // handled inequality to the assumption set
36    assumptionSet.add(iq);
37
38    // test all partial derivatives
39    boolean result = true;
40
41    for (int i=0; i<pd.length; i++) {
42
43        result = result & (pd[i].wprove(assumptionSet, calls) |
44                            pd[i+1].wprove(assumptionSet, calls));
45        i++;
46    }//for
47
48    return result;
49
50 }//wprove

```

Method `wprove()` starts with a check if some member type of the inequality is `null`. Then both types are unfolded by the wrapper method `unfold()` of class `Inequality` which calls `unfold()` on both member types.

After unfolding, the wellformedness is checked. Method `isWellformed()` in class `Inequality` calls `isWellformed()` on member types `r` and `s` and returns *TRUE* if both types are wellformed, otherwise *FALSE*.

Trivial case checks 1 – 6 as described in Section 6.4 are performed by corresponding methods. The code itself is not very insightful.

After computing the leading names of `r`, trivial case checks 7 and 8 are

performed, which require the leading names.

If the inequality was not yet proven or disproven, the partial derivatives of $\mathbf{r} <: \mathbf{s}$ are computed and if they are a non-empty set, the inequality is marked as ‘analyzed’ by adding it to the assumption set `assumptionSet`.

The result value is initialized with *TRUE*. The following loop performs the recursive derivation of the inequalities of the partial derivatives of the current inequality. The results of this derivation are structured as conjunction of disjunctions. (The original code contains a switching mechanism that can turn on short circuit evaluation of the boolean operators.)

After analyzing all inequalities, the result is returned and the subtyping process is completed.

Chapter 7

Related and Future Work

7.1 XML Schema Type Import

This thesis relies on the import of XML Schema types. The level of abstraction on which this thesis is based, recognizes types as regular expressions. The genuine XML Schema types are not regular expressions. An example for the transformation of XML Schema types into regular expressions can be found in Section 1.3.

The translation of XML Schema types into regular expressions is a fully developed part of the Pathfinder project. It is explained in [Ri03].

7.2 Normalization of non-wellformed types

In Section 3.3 we developed two wellformedness constraints that ensure regularity and derivability of types without stepping into an endless recursion. These constraints require recursive occurrences of named recursive regular expressions standing only in tail positions within concatenations and being preceded by a non-nullable head.

The constraints exclude some types that could be rewritten such that they fulfill the constraints.

$$T5 : (T5 \cdot \mathbf{b}) \mid \mathbf{a} \quad \equiv \quad T6 : \mathbf{a} \cdot \mathbf{b}^*.$$

Type $T5$ violates wellformedness constraint 1 and 2. But in this case, the recursion can be completely eliminated and replaced by an iteration.

$$T7 : (T7 \cdot \mathbf{a}) \mid (T7 \cdot \mathbf{b}) \mid \varepsilon \quad \equiv \quad T8 : (\mathbf{a} \cdot T8) \mid (\mathbf{b} \cdot T8) \mid \varepsilon \quad \equiv \quad T9 : (\mathbf{a}^* \mid \mathbf{b}^*)^*$$

Type $T7$ violates both wellformedness constraints, too. We can either try to rewrite the type such that the recursive occurrences are placed in tail-positions which leads to $T8$, or we apply the same strategy like in the case of $T5$ and replace the recursive occurrences by iterations which leads to $T9$.

Obviously there are some kinds of types that violate at least one constraint and cannot be rewritten as a wellformed expression:

$$T10 : \mathbf{a} \cdot T10 \cdot \mathbf{b} \mid \varepsilon$$

Type *T10* violates wellformedness constraint 1 and regularity because it cannot be transformed into a finite automaton. There is also no possibility for rewriting *T10* into a wellformed regular expression. (In Section 3.3 we stated that constraint 1 ensures regularity. This case illustrates this fact.)

Two aspects remain as future work:

1. It would be helpful if we had the possibility of efficiently recognizing if a given type is rewritable into a wellformed regular expression or not.
2. If a type is recognized as rewritable, we should have a system of rewriting rules that ensures the transformation into a wellformed expression.

A regular expression fulfilling both wellformedness constraints seems to be a top-level alternation expression defining at least one non-recursive alternative. But this is at most a necessary condition because it does not ensure regularity which becomes lucid by looking at *T10*.

We shew a rough outlines of two possible strategies of normalization: replacing the recursive occurrences by iterations or rewriting the recursive type with recursive occurrences re-positioned. But normalization, however, requires much more exploration.

7.3 Non-empty Intersections of Types

Ongoing research suggests that Antimirov's calculus can be transformed into a calculus that tests if

$$L(r) \cap L(s) \neq \emptyset$$

holds for given types r and s .

In the Pathfinder project, we use a variant of Antimirov's calculus that works in practice and performs the test for a non-empty intersection of two regular expressions.

However, no theoretical foundation of this variant has been formulated, yet. This remains as future work and will not be explained in this thesis.

Appendix A

Logfile for Running Example

The following logfile was generated by the logging facility of method `prove()` and shows the derivation of the running example defined in Section 1.5.

```
1 +-----Call: 0
2 |
3 | Input r: foo[string] (bar[integer] foo[string])*
4 | Input s: (foo[string] bar[integer])* foo[string]
5 |
6 | Leading names of r: [foo]
7 |
8 | PD of r <: s for all leading names of r (added to A):
9 |
10 | TypePairs in left clause :
11 |   0: <string, (bar[integer] foo[string])*>
12 |
13 | TypePairs in right clause:
14 |   0: <string, bar[integer] (foo[string] bar[integer])* foo[string]>
15 |   1: <string, e>
16 |
17 | 8 Inequalities in the Partial Derivatives of:
18 | foo[string] (bar[integer] foo[string])* <: (foo[string] bar[integer])* foo[string]
19 |
20 | (0) string <: none √
21 | (1) (bar[integer] foo[string])* <: (bar[integer] (foo[string] bar[integer])* foo[string] | e)
22 | /\
23 | (2) string <: string √
24 | (3) (bar[integer] foo[string])* <: e
25 | /\
26 | (4) string <: string √
27 | (5) (bar[integer] foo[string])* <: bar[integer] (foo[string] bar[integer])* foo[string]
28 | /\
29 | (6) string <: (string | string) √
30 | (7) (bar[integer] foo[string])* <: none
31 |
32 +-----Call: 1
33 |
34 | Input r: string
35 | Input s: none
36 |
37 | s is none.
38 |
39 | Result: false
40 +-----Call: 1
41 | √
42 +-----Call: 2
43 |
44 | Input r: (bar[integer] foo[string])*
45 | Input s: (bar[integer] (foo[string] bar[integer])* foo[string] | e)
46 |
47 | Leading names of r: [bar]
48 |
49 | PD of r <: s for all leading names of r (added to A):
50 |
51 | TypePairs in left clause :
52 |   0: <integer, foo[string] (bar[integer] foo[string])*>
53 |
54 | TypePairs in right clause:
55 |   0: <integer, (foo[string] bar[integer])* foo[string]>
56 |
57 | 4 Inequalities in the Partial Derivatives of:
58 | (bar[integer] foo[string])* <: (bar[integer] (foo[string] bar[integer])* foo[string] | e)
59 |
60 | (0) integer <: none √
```

```

61 | | (1) foo[string] (bar[integer] foo[string])* <: (foo[string] bar[integer])* foo[string]
62 | | /\
63 | | (2) integer <: integer \/
64 | | (3) foo[string] (bar[integer] foo[string])* <: none
65 | |
66 | | +-----Call: 3
67 | | |
68 | | | Input r: integer
69 | | | Input s: none
70 | | |
71 | | | s is none.
72 | | |
73 | | | Result: false
74 | | | +-----Call: 3
75 | | | \
76 | | | +-----Call: 4
77 | | | |
78 | | | | Input r: foo[string] (bar[integer] foo[string])*
79 | | | | Input s: (foo[string] bar[integer])* foo[string]
80 | | | |
81 | | | | r<:s already analyzed.
82 | | | |
83 | | | | Result: true
84 | | | | +-----Call: 4
85 | | | |
86 | | | | /\
87 | | | |
88 | | | | +-----Call: 5
89 | | | | |
90 | | | | | Input r: integer
91 | | | | | Input s: integer
92 | | | | |
93 | | | | | r same type as s.
94 | | | | |
95 | | | | | Result: true
96 | | | | | +-----Call: 5
97 | | | | | \
98 | | | | | +-----Call: 6
99 | | | | |
100 | | | | | Input r: foo[string] (bar[integer] foo[string])*
101 | | | | | Input s: none
102 | | | | |
103 | | | | | s is none.
104 | | | | |
105 | | | | | Result: false
106 | | | | | +-----Call: 6
107 | | | | |
108 | | | | | Result: true
109 | | | | | +-----Call: 2
110 | | | | |
111 | | | | | /\
112 | | | | |
113 | | | | | +-----Call: 7
114 | | | | | |
115 | | | | | | Input r: string
116 | | | | | | Input s: string
117 | | | | | |
118 | | | | | | r same type as s.
119 | | | | | |
120 | | | | | | Result: true
121 | | | | | | +-----Call: 7
122 | | | | | | \
123 | | | | | | +-----Call: 8
124 | | | | | |
125 | | | | | | Input r: (bar[integer] foo[string])*
126 | | | | | | Input s: e
127 | | | | | |
128 | | | | | | s is epsilon.
129 | | | | | |
130 | | | | | | Result: false
131 | | | | | | +-----Call: 8
132 | | | | | |
133 | | | | | | /\
134 | | | | | |
135 | | | | | | +-----Call: 9
136 | | | | | | |
137 | | | | | | | Input r: string
138 | | | | | | | Input s: string
139 | | | | | | |
140 | | | | | | | r same type as s.
141 | | | | | | |
142 | | | | | | | Result: true
143 | | | | | | | +-----Call: 9
144 | | | | | | | \
145 | | | | | | | +-----Call: 10
146 | | | | | | |
147 | | | | | | | Input r: (bar[integer] foo[string])*
148 | | | | | | | Input s: bar[integer] (foo[string] bar[integer])* foo[string]
149 | | | | | | |
150 | | | | | | | r nullable, s not nullable.

```

```

151 | |
152 | | Result: false
153 | | +-----Call: 10
154 | |
155 | | /\
156 | |
157 | | +-----Call: 11
158 | |
159 | | Input r: string
160 | | Input s: (string | string)
161 | |
162 | | Leading names of r: [string]
163 | |
164 | | PD of r <: s for all leading names of r (added to A):
165 | |
166 | |   TypePairs in left clause :
167 | |     0: <e, e>
168 | |
169 | |   TypePairs in right clause:
170 | |     0: <e, e>
171 | |
172 | | 4 Inequalities in the Partial Derivatives of:
173 | | string <: (string | string)
174 | |
175 | | (0) e <: none \/
176 | | (1) e <: e
177 | | /\
178 | | (2) e <: e \/
179 | | (3) e <: none
180 | |
181 | | +-----Call: 12
182 | |
183 | | Input r: e
184 | | Input s: none
185 | |
186 | | s is none.
187 | |
188 | | Result: false
189 | | +-----Call: 12
190 | | \/
191 | | +-----Call: 13
192 | |
193 | | Input r: e
194 | | Input s: e
195 | |
196 | | r same type as s.
197 | |
198 | | Result: true
199 | | +-----Call: 13
200 | |
201 | | /\
202 | |
203 | | +-----Call: 14
204 | |
205 | | Input r: e
206 | | Input s: e
207 | |
208 | | r same type as s.
209 | |
210 | | Result: true
211 | | +-----Call: 14
212 | | \/
213 | | +-----Call: 15
214 | |
215 | | Input r: e
216 | | Input s: none
217 | |
218 | | s is none.
219 | |
220 | | Result: false
221 | | +-----Call: 15
222 | |
223 | | Result: true
224 | | +-----Call: 11
225 | | \/
226 | | +-----Call: 16
227 | |
228 | | Input r: (bar[integer] foo[string])*
229 | | Input s: none
230 | |
231 | | s is none.
232 | |
233 | | Result: false
234 | | +-----Call: 16
235 | |
236 | | Result: true
237 | | +-----Call: 0
238 | |
239 | | r <: s is true.
240 | | Number of recursive calls: 16

```


Bibliography

- [Ant94] Antimirov, Valentin M.: “**Rewriting Regular Inequalities.**” in: Reichel, ed.: *Fundamentals of Computation Theory*, vol. 965 of LNCS, pages 116-125. Springer Verlag Heidelberg, 1994.
- [AntMos95] Antimirov, Valentin; Mosses, Peter D.: “**Rewriting Extended Regular Inequalities.**” in: *Theoretical Computer Science 1995*. Vol. 143 no. 1 pages 51-72. First published in a short version in: G. Rozenberg and A. Salomaa, eds.: *Developments in Language Theory - At the Crossroads of Mathematics, Computer Science and Biology*, pages 195-209. World Scientific, Singapore, 1994.
- [Ant95] Antimirov, Valentin M.: “**Partial Derivatives of Regular Expressions and Finite Automata Constructions.**” in: E.W. Mayr and C. Puech, eds.: LNCS 900: *Proceedings of STACS '95*, pages 455-466. Munich, Germany, March 1995. Springer Verlag.
- [BW98] Brüggemann-Klein, Anne; Wood, Derrick: “**One-unambiguous Regular Languages.**” in: *Information and also Computation 1998*. vol. 140 no. 2 pages 229-253 and vol. 142 no. 2 pages 182-206.
- [CFS02] Choi, Byron; Fernández, Mary; Siméon, Jérôme: “**The XQuery Formal Semantics: A Foundation for Implementation and Optimization.**” May 2002. Available at: <http://www.cis.upenn.edu/~kkchoi/galax.pdf>
- [GHV95] Gamma, Richard; Helm, Vlissides, Johnson: **Design Patterns**. Addison-Wesley 1995.
- [Glu61] Glushkov, V.M.: “**The abstract theory of automata.**” *Russian Mathematical Surveys* vol. 16, 1961, pages 1-53.
- [Hos00] Hosoya, Haruo; Vouillon, Jérôme; Pierce, Benjamin C.: “**Regular Expression Types for XML.**” in: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFB '00)*, Montreal, Canada, vol. 35 no. 9 of SIGPLAN Notices, pages 11-22. ACM, September 18-21 2000.
- [HP01] Hosoya, Haruo; Pierce, Benjamin C.: “**Regular Expression Pattern Matching for XML.**” in: *ACM SIGPLAN Notices 2001*, vol. 36 Nr. 3, pages 67-80.

- [HP02] Hosoya, Haruo; Pierce, Benjamin C.: “**XDuce: A Statically Typed XML Processing Language.**” ACM Transactions on Internet Technology, 2002. Available at: <http://www.cis.upenn.edu/~bcpierce/papers/xduce-toit.pdf>
- [HU79] Hopcroft, J.E; Ulman, J.D.: **Introduction to Automata Theory, Languages and Computation.** Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA 1979.
- [KeLi03] Kempa, Martin; Linnemann, Volker: “**Type Checking in XOBEL.**” in: E. Rahm G. Weikum, H. Schoning, eds.: Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung, pages 265-274, Leipzig, 2003.
- [KeLi02] Kempa, Martin; Linneman, Volker: “**XML-Based Applications Using XML Schema.**” EDBT Workshops 2002, pages 67-90.
- [MY60] McNaughton,R. and Yamada,H.: “**Regular expressions and state graphs for automata.**” IRE Transactions on Electronic Computers, EC-9(1) pages 39-47, March 1960.
- [Ri03] Rittinger, Jan: “**XML Schema Import for the Pathfinder XQuery Compiler.**” Forthcoming.
- [SW03] Siméon, Jérôme; Wadler, Philip: “**The Essence of XML.**” in: POPL 2003. New Orleans. January 2003.
- [TATA] Comon, Hubert; Dauchet, Max; Gilleron, Rémi; Jacquemard, Florent; Lugiez, Denis; Tison, Sophie; Tommasi, Marc: **Tree Automata Techniques and Applications.** Free Manuscript. Available at: <http://www.grappa.univ-lille3.fr/tata/>
- [Wad02] Wadler, Philip: “**XQuery: a typed functional language for querying XML.**” in: Advanced Functional Programming. Oxford, August 2002.
- [XPL03] **XML Path Language (XPath) 2.0** W3C Working Draft. 22 August 2003. Available at: <http://www.w3.org/TR/2003/WD-xpath20-20030822>
- [XQFS03] **XQuery 1.0 and XPath 2.0 Formal Semantics.** W3C Working Draft. 22 August 2003. Available at: <http://www.w3.org/TR/2003/WD-xquery-semantics-20030822/>
- [XQL03] **XQuery 1.0: An XML Query Language.** W3C Working Draft. 22 August 2003. Available at: <http://www.w3.org/TR/2003/WD-xquery-20030822/>
- [XSD01] **XML Schema Part 2: Datatypes** W3C Working Draft. 2 May 2001. Available at: <http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/>