On Subtyping of Tree-structured Data
A Polynomial Approach

François Bry[1], Włodzimierz Drabent[2,3], and Jan Małuszyński[3]

[1] Ludwig-Maximilians-Universität München, Institut für Informatik, Oettingenstr. 67, D-80538 München, Germany, francois.bry@ifi.lmu.de
[2] Institute of Computer Science, Polish Academy of Sciences, ul. Ordona 21, Pl – 01-237 Warszawa, Poland, drabent@ipipan.waw.pl
[3] Linköping University, Department of Computer and Information Science, S – 581 83 Linköping, Sweden, jmz@ida.liu.se

Abstract. This paper discusses subtyping of tree-structured data encountered on the Web, e.g. XML and HTML data. Our long range objective is to define a type system for Web and/or Semantic Web query languages amenable to static type checking. We propose a type formalism motivated by XML Schema and accommodating two concepts of subtyping: inclusion subtyping (corresponding to XML Schema notion of type restriction) and extension subtyping (motivated by XML Schema's type extension). We present algorithms for checking both kinds of subtyping. The algorithms are polynomial if certain conditions are imposed on the type definitions; the conditions seem natural and not too restrictive.

1 Introduction

This paper discusses subtyping of tree-structured data. With the Web, the Web page markup language HTML, and the emergence of XML as data specification formalism of choice for data on the Web, tree-structured data are receiving an increasing attention. Indeed, HTML and XML documents are tree-structured – cycles induced by ID and IDREF attributes and/or links being neglected as it is common with Web query languages.

The long range objective of the research reported about in this paper is to define a type system for Web and/or Semantic Web query languages amenable to static type checking, the query language Xcerpt [7, 5] being a premier candidate for such an extension. Such a type system should support subtyping so that the well-typed procedures/methods of the language could also be safely applied to subtypes. The question is thus about the suitable concept of type and subtype. We provide a formalism for specifying types motivated by XML Schema [2] and we show two relevant concepts of subtyping: *inclusion subtyping*, motivated by XML Schema notion of *type restriction*, and *extension subtyping*, motivated by XML Schema notion of *type extension*. We show conditions for type definitions under which subtyping can be checked in polynomial time.

As XML data are essentially tree-structured, a natural approach is to view types as sets of trees and subtyping as set inclusion. To specify such types a formalism of regular expression types is proposed in [10] and inclusion subtyping is

discussed. Checking of the subtyping relations can then be reduced to checking inclusion of sets specified by regular tree grammars [8]. Tree grammars are a formalism of choice for specifying types for XML documents because both DTD and XML schemas are derived from them. The inclusion problem for languages defined by tree grammars is decidable but EXPTIME-complete. It is argued in [10] that for the regular expression types needed *in practice* checking of inclusion is usually quite efficient. We propose a formalism which is a restricted variant of regular expression types. We argue that the restrictions reflect the usual requirements of the XML Schema, thus our formalism is sufficiently expressive for practical applications. On the other hand, it makes it possible to identify source of potential inefficiency, and to formulate syntactic conditions on type definitions under which subtyping can be checked in polynomial time.

It seems that subtyping by inclusion is intuitively very close to the XML Schema concept of *type restriction*, and as argued in [3] replacement of the latter by the former would greatly simplify XML Schema.

In object-oriented processing the methods of a class must be as well applicable to the subclasses of the class. Subtyping by inclusion is not sufficient to capture the notion of subclass. For example, given a type `person` of XML documents we may define a type `student` where the documents have the same elements as `person` augmented with the obligatory new element `university`, showing the affiliation of the student. This kind of definitions is supported by XML Schema mechanism of *type extension*. Notice that in our example none of the classes is the subset of the other. However, we would like to be able to apply all the methods of the class `person` to the objects of the class `student`. This can be done by ignoring the additional element of the input document. As the objective is static typing of the methods, we need yet another notion of subtyping, in addition to subtyping by inclusion. For our type formalism we define a formal notion of *extension subtype* that formalizes such situations. In this paper we outline an algorithm for checking extension subtyping and we give sufficient condition for type definitions under which the check is polynomial.

The paper is organized as follows. Section 2 discusses the kind of tree-structured data we want to deal with, and introduces a formalism of *type definitions* for specifying sets of such data. The next section gives an algorithm for validation of tree-structured data w.r.r. type definitions. Sections 4, 5 discuss, respectively, inclusion and extension subtyping. Section 6 presents conclusions.

2 Tree-structured data

2.1 Data terms

This section formalizes our view of tree-structured data. The next one introduces a formalism for specifying decidable sets of such data.

We define a formal language of *data terms* to model tree-structured data such as XML documents. This definition does not explicitly capture the XML mechanism for defining and using references. We note that two basic concepts of XML are *tags* indicating nodes of an ordered tree corresponding to a document and

attributes[4] used to attach attribute-value mappings to the nodes of a tree. Such a finite mapping can be represented as an unordered tree (see Example 1 below). It should also be noticed that *all* group of XML Schema [2] allows specifying elements that may appear in the document in any order. These observations bring us to the conclusion that we want to deal with labelled trees where the children of a node are either linearly ordered or are unordered. We will call them *mixed trees* to indicate their distinction from both ordered trees, and unordered trees.

We assume two disjoint alphabets: a countably infinite alphabet $\mathcal{L}$ of **labels**, and an alphabet $\mathcal{B}$ of **basic constants**. Basic constants represent some basic values, such as numbers or strings, while labels are tree constructors.

We now define a formal language of *data terms* for representing mixed trees. The linear ordering of children will be indicated by the brackets $[,]$, while unordered children are placed in the braces $\{,\}$.

Definition 1. A **data term** is an expression defined inductively as follows:

- Any basic constant is a data term,
- If l is a label and $t_1, \ldots, t_n$ are $n \geq 0$ data terms, then $l[t_1 \cdots t_n]$ and $l\{t_1 \cdots t_n\}$ are data terms.

Data terms not containing $\{,\}$ will be called **ordered**.

The data terms $l[\,]$ or $l\{\}$ are different. One may consider it more natural not to distinguish between the empty sequence and the empty set of arguments. This however would result in some extra special cases in our definitions and algorithms further on.

Notice that the component terms are not separated by commas. This notation is intended to stress the fact that the label l in a data term $l[t_1 \cdots t_n]$ is not an n-argument function symbol. It has rather a single argument which is a sequence (string) of data terms $t_1, \ldots, t_n$ (where $n \geq 0$). Similarly the argument of l in $l\{t_1 \cdots t_n\}$ is a set of data terms.

Example 1. Consider the following XML document

```
<person friend="yes" coauthor="yes">
    <first-name>Francois</first-name>
    <last-name>Bry</last-name>
    <notes/>
</person>
```

It can be represented as a data term

$$\begin{array}{l} \mathit{person}[\,\mathit{attributes}\{\,\mathit{friend}[\mathrm{yes}]\ \mathit{coauthor}[\mathrm{yes}]\,\} \\ \qquad \mathit{first\text{-}name}[\,\mathrm{Francois}\,]\ \ \mathit{last\text{-}name}[\,\mathrm{Bry}\,]\ \ \mathit{notes}[\,]\ \ ] \end{array}$$

where yes, Francois, Bry are basic constants and *attributes, friend, coauthor, first-name, last-name, notes* are labels.

The **root** of a data term t, denoted $\mathit{root}(t)$, is defined as follows. If t is a constant then $\mathit{root}(t) = t$. Otherwise t is of the form $l[t_1 \cdots t_n]$ or $l\{t_1 \cdots t_n\}$ and $\mathit{root}(t) = l$.

[4] However, there is no syntactic difference between tag names and attribute names.

2.2 Specifying sets of data terms

We now present a metalanguage for specifying decidable sets of data terms, which will be used as types in processing of tree-structured data. The idea is similar to that of [10] (see the discussion at the end of this section) and is motivated by DTD's and by XML Schema.

We define the sets of data terms by means of grammatical rules. We assume existence of *base types*, denoted by **type constants** from the alphabet $\mathcal{C}$ and a countably infinite alphabet $\mathcal{V}$ of **type variables**, disjoint with $\mathcal{C}$. Type constants and type variables will be called **type names**.

The intention is that base types correspond to XML primitive types. We assume that each type constant $C \in \mathcal{C}$ is associated with a set $[\![C]\!] \subseteq \mathcal{B}$ of basic constants. We assume that for every pair $C_1, C_2 \in \mathcal{C}$ of type constants we are able to decide whether or not $[\![C_1]\!] \subseteq [\![C_2]\!]$ and whether or not $[\![C_1]\!] \bigcap [\![C_2]\!] = \emptyset$. Additionally we assume that for each $C \in \mathcal{C}$ and a finite tuple $C_1, \ldots, C_n \in \mathcal{C}$ we are able to decide whether $[\![C]\!] \subseteq [\![C_1]\!] \cup \cdots \cup [\![C_n]\!]$.

We first introduce an auxiliary syntactic concept of a *regular type expression*. As usually, we use ϵ to denote the empty sequence.

Definition 2. A **regular type expression** is a regular expression (see e.g. [9]) over the alphabet $\mathcal{V} \cup \mathcal{C}$.

Thus ϵ and any type constant or type variable T are regular type expressions, and if τ, τ_1, τ_2, are type expressions then $(\tau_1\tau_2)$, $(\tau_1|\tau_2)$ and (τ^*) are regular type expressions. As usually, every regular type expression r denotes a (possibly infinite) *regular language* $L(r)$ over the alphabet $\mathcal{V} \cup \mathcal{C}$: $L(\epsilon) = \{\epsilon\}$, $L(T) = \{T\}$, $L((\tau_1\tau_2)) = L(\tau_1)L(\tau_2)$, $L((\tau_1|\tau_2)) = L(\tau_1) \cup L(\tau_2)$, and $L((\tau^*)) = L(\tau)^*$. We adopt the usual notational conventions, where the parentheses are suppressed by assuming the following priorities of operators: concatenation, $|$, $*$ (see e.g. [9]).

It is well known that any language specified by a regular expression can also be defined by a *finite automaton*, deterministic (DFA) or non-deterministic (NFA). There are algorithms that transform any regular expression of length n into an equivalent NFAϵ with $O(n)$ states, and any NFAϵ into an equivalent DFA see e.g. [9]. In the worst case the latter transformation may exponentially increase the number of states. Brüggemann-Klein and Wood [6] introduced a class of 1-unambiguous regular expressions, for which this transformation is linear. For such regular expression, a natural transformation from NFAϵ to NFA results in an DFA.

Notice that the XML definition [1] requires (Section 3.2.1) that content models specified by regular expressions in element type declarations of a DTD are *deterministic* in the sense of Appendix E of [1]. This condition ensures existence of a DFA acceptor with number of states linear w.r.t. the size of the regular expression. It seems that this informal condition is equivalent with that of [6]. We do not put specific restrictions on our regular type expressions, but we expect that those used in practice would not cause exponential blow-up of the number of states of the constructed DFA acceptors.

As syntactic sugar for regular expressions we will also use the following notation:

– $\tau(n:m)$, where $m \geq n$ as a shorthand for $\tau^n|\tau^{n+1}|...|\tau^m$,
  notice that τ^* can be seen as $\tau(0:\infty)$
– τ^+ as a shorthand for $\tau\tau^*$,
– $\tau^?$ as a shorthand $\tau(0:1)$,

where τ is a regular expression and n is a natural number and m is a natural number or ∞.

Definition 3. A **multiplicity list** is a regular type expression of the form

$$s_1(n_1:m_1)\cdots s_k(n_k:m_k)$$

where $k \geq 0$ and $s_1, ..., s_k$ are distinct type names.

It can be easily seen that for the language defined by a multiplicity list there exists a DFA acceptor with the number of states linear w.r.t. to the number of the type names in the list.

We now introduce a grammatical formalism for defining sets of data terms. Such a grammar will define a finite family of sets, indexed by a finite number of type variables $T_1, ..., T_m$. Each variable T_i will be associated with a set of data terms, all of which have identical root label l_i. This is motivated by XML, where the documents defined by a DTD have identical main tags. It is not required that $l_i \neq l_j$ for $i \neq j$. Our grammatical rules can be seen as content definitions for classes of data terms. So they play a similar role for data terms as DTD's (or XML Schemas) play for XML documents.

Definition 4. A **type definition** D for (distinct) type variables $T_1, ..., T_n$, for $n \geq 1$ is a set of **rules** $\{R_1, ..., R_n\}$ where each rule R_i is of the form

$$T_i \rightarrow E_i$$

and E_i is an expression of the form $l_i[r_i]$ or of the form $l_i\{q_i\}$, $i = 1, ..., n$, where every l_i is a label, r_i is a regular type expression over $\{T_1, ..., T_n\} \cup \mathcal{C}$ and every q_i is a multiplicity list over $\{T_1, ..., T_n\} \cup \mathcal{C}$.

Thus, we use two kinds of rules, which describe construction of ordered or unordered trees (data terms). As formally explained below, rules of the form $T \rightarrow l[r]$ describe a family T of trees where the children of the root are ordered and their allowed sequence is described by a general regular expression r. The rules of the form $T \rightarrow l\{q\}$ describe a family T of trees where the children of the root are unordered. The ordering of the children is thus irrelevant and it makes no sense to use a general regular expression for describing them. We use instead the multiplicity list q which specifies allowed number of children of each type. A type definition not including the rules of the form $T \rightarrow l\{r\}$ (where $L(r)$ contains a non-empty string) will be called an **ordered** type definition.

We illustrate the definition by the following example. In our type definitions the type names start with capital letters, labels with lower case letters, and type constants with symbol #.

Example 2. We want to represent genealogical information of people by means of data terms. A person would be represented by the name, sex and a similar information about his/her parents. The latter may be unknown, in which case it will be missing in the data term. This intuition is reflected by the following grammar.

$$
\begin{array}{l}
\mathit{Person} \rightarrow \mathit{person}[\,\mathit{Name}\,(M|F)\,\mathit{Mother}?\,\mathit{Father}?\,] \\
\mathit{Name} \rightarrow \mathit{name}[\#\mathit{name}] \\
M \rightarrow m[\,] \\
F \rightarrow f[\,] \\
\mathit{Mother} \rightarrow \mathit{person}[\mathit{Name}\;F\;\mathit{Mother}?\,\mathit{Father}?\,] \\
\mathit{Father} \rightarrow \mathit{person}[\mathit{Name}\;M\;\mathit{Mother}?\,\mathit{Father}?\,]
\end{array}
$$

In the sequel we give a formal semantics of type definitions, which will correspond to this intuition.

Definition 4 requires that each type name maps to a label, but the mapping may not be one-one, as illustrated by the above example, where the types $Person, Father$ and $Mother$ map to the same label *person*. This is more general than XML, where there is a one-one correspondence between element types defined by a DTD and tags (see Section 3.1 in [1]).

Such a restriction facilitates validation of documents but excludes subtyping understood as document set inclusion. It turns out that we can facilitate validation and still have inclusion subtyping if the one-one correspondence between types and labels is enforced only locally for type symbols occurring in the regular expression of each rule of the grammar. This is reflected by the following definition.

Definition 5. The type definition D of Definition 4 is said to be **proper** if for each E_i $(i = 1, \ldots, n)$

- for any distinct type variables T_{i_1}, T_{i_2} occurring in E_i, $l_{i_1} \neq l_{i_2}$, and
- for any distinct type constants C_1, C_2 occurring in E_i, $[\![C_1]\!] \cap [\![C_2]\!] = \emptyset$.

Notice that the type definition of Example 2 is not proper. The regular expression of the first rule includes different types $Mother$ and $Father$ with the same label *person*. Replacing (in three rules) each of them by the type $Person$ would make the definition proper.

A type definition D associates with each type variable T_i a set of data terms, as explained below.

Definition 6. A **data pattern** is inductively defined as follows

- a type variable, a type constant, and a basic constant are data patterns,
- if $d_1, \ldots, d_n$ for $n \geq 1$ are data patterns and l is a label then $l[d_1 \cdots d_n]$ and $l\{d_1 \cdots d_n\}$ are data patterns.

Thus, data terms are data patterns, but not necessarily vice versa, since a data pattern may include type variables and type constants in place of data terms. Given a type definition D we use it to define a rewrite relation $\rightarrow_D$ on data patterns.

Definition 7 (of $\rightarrow_D$). Let d, d' be data patterns. $d \rightarrow_D d'$ iff one of the following holds:

1. For some type variable T
   - there exists a rule $T \rightarrow l[r]$ in D and a string $s \in L(r)$, or
   - there exists a rule $T \rightarrow l\{r\}$ in D and a string $s_0 \in L(r)$ and a permutation s of s_0

   such that d' is obtained from d by replacing an occurrence of T in d, respectively, by $l[s]$, or by $l\{s\}$.
2. d' is obtained from d by replacing an occurrence of a type constant S by a basic constant in $[\![S]\!]$.

Iterating the rewriting steps we may eventually arrive at a data term. This gives a semantics for type definitions.

Definition 8. Let D be a type definition for $T_1, ..., T_n$. A **type** $[\![T_i]\!]_D$ associated with T_i by D is defined as the set of all data terms t that can be obtained from T_i:

$$[\![T_i]\!]_D = \{\, t \mid T_i \rightarrow_D^* t \text{ and } t \text{ is a data term} \,\}$$

Additionally we define the set of data terms specified by a given data pattern d, and by a given regular expression r:

$$[\![d]\!]_D = \{\, t \mid d \rightarrow_D^* t \text{ and } t \text{ is a data term} \,\},$$
$$[\![r]\!]_D = \{\, t_1 \cdots t_k \mid t_1 \in [\![T_1]\!]_D, \ldots, t_k \in [\![T_k]\!]_D \text{ for some } T_1 \cdots T_k \in L(r) \,\}.$$

Definition 9 (of $label_D(T)$ and $type_D(c_i, r)$). Notice that:

- Every type variable T has in D only one rule defining it, the label of this rule will be denoted $label_D(T)$.
- For a proper type definition D, if $l[c_1 \cdots c_n] \in [\![T]\!]$ and $T \rightarrow l[r] \in D$ or $l\{c_1 \cdots c_n\} \in [\![T]\!]$ and $T \rightarrow l\{r\} \in D$ then for each c_i which is not a constant the root label of c_i determines a unique type variable S such that S occurs in r and $c_i \in [\![S]\!]$. Similarly, for each constant c_i there is a unique type constant S occurring in r such that $c_i \in [\![S]\!]$. Such type variable or constant S will be denoted $type_D(c_i, r)$. For a data term d which for any S occurring in r is not a member of a $[\![S]\!]$, we assume that $type_D(d, r) = S_0$, where S_0 is some fixed type name not occurring in D.

If it is clear from the context which type definition is considered, we can omit the subscript in the notation $[\![\ ]\!]_D$, $label_D(\ )$ and $type_D(\ ,\ )$.

Example 3. Consider the following type definition D (which is proper):

$$Person \rightarrow person[Name\ (M|F)\ Person(0:2)]$$
$$Name \rightarrow name[\#name]$$
$$M \rightarrow m[\,]$$
$$F \rightarrow f[\,]$$

Let $\mathrm{john}, \mathrm{mary}, \mathrm{bob} \in [\![\#name]\!]$. Extending the derivation

$$Person \to person[Name\ M\ Person] \to^* person[name[\#name]\ m[\,]\ Person]$$

one can check that the following data term is in $[\![Person]\!]$

$$person[name[\mathrm{john}]\ m[\,]\ person[name[\mathrm{mary}]\ f[\,]\ person[name[\mathrm{bob}]m[\,]]\,]\,].$$

Our type definitions are similar to those of [10]. The main differences are: 1. Our data are mixed trees instead of ordered trees. 2. Our types are sets of trees; sequences of trees described by regular expressions play only an auxiliary role. In addition, all elements of any type defined in our formalism have the same root label. In contrast to that, types of [10] are sets of sequences of trees. Allowing mixed trees creates better data modelling possibilities and we expect it to be useful in applications.

Apart of the use of mixed trees, our formalism is a restriction of that of [10] since a set of trees can be seen as a set of one-element sequences of trees. Our restriction seems not to be essential since we can also specify sets of sequences of trees by means of regular type expressions, even though such sets are not considered types. It reflects the intuition that type definitions are used for describing tree-structured data with explicitly labelled roots, and that data of the same type have identical root labels. This conforms to the practice of XML and makes it possible to design new validation and type inclusion algorithms with a potential for better complexity than the algorithms of [10].

In the rest of the paper we consider only proper data definitions, unless stated otherwise. This results in simpler algorithms. The class of ordered (i.e. without $\{\}$) proper type definitions is essentially the same as single-type tree grammars of [11]. Restriction to proper definitions seems reasonable, as the sets defined by main XML schema languages (DTD and XML Schema) can be expressed by such definitions [11].

3 Validating data terms

A data definition D describes expected structure of data and we will use it to validate given data items d, i.e. to check whether or not $d \in [\![T]\!]$, for a given type defined by D. This section gives a general algorithm for validating data terms against proper data definitions and examines its complexity.

Validating ordered data terms. We first consider proper type definitions which are ordered (i.e. include no rules of the form $T \to \{r\}$). In that case each $[\![T]\!]$ is the set of ordered data terms derivable by the rules. We show an algorithm that for a given proper ordered type definition D, type name T, and data term $d = c[d_1 \cdots d_k]$ $(k \geq 0)$ decides whether or not $d \in [\![T]\!]_D$.

The algorithm depends on the fact that D is proper. This implies that for each distinct type names S, S' occurring in a regular expression r from D, $[\![S]\!]_D \cap [\![S']\!]_D = \emptyset$. Thus when checking whether a sequence $d_1 \cdots d_k$ of data terms is in

$[\![r]\!]_D$ we need, for a given i, to check $d_i \in [\![S]\!]_D$ for at most one type name S, namely $S = type(d_i, r)$.

The algorithm employs checking whether $x \in L(r)$ for a string x and a regular expression r. This can be done in time $O(|r| \cdot |x|)$ [4]. Alternatively, one can construct a DFA for $L(r)$ for each regular expression in D; this is to be done once. Then the checking requires $|x|$ steps.

The validation algorithm is described as follows.

validate(d, T) :
  IF T is a type constant THEN
    check whether d is a basic constant in $[\![T]\!]$ and return the result
  ELSE (T is a type variable)
  IF d is a basic constant then return *false*
  ELSE
  IF the rule for T in D is $T \to c[r]$ THEN
    IF $root(d) \neq c$ THEN return *false*
    ELSE
      let $d = c[d_1 \cdots d_k]$ $(k \geq 0)$,
      let $T_i = type(d_i, r)$ for $i = 1, \ldots k$,
      IF $T_1 \cdots T_k \notin L(r)$
        THEN return *false*
      ELSE
        return $\bigwedge_{i=1}^{k} validate(d_i, T_i)$
  ELSE (no rule for T) return *false*.

This algorithm traverses the tree d. It checks if $x \in L(r)$, for some strings and regular expressions. The sum of the lengths of all the strings subjected to these checks is not greater than the number of nodes in the tree. Some nodes of d may require validation against base types. The time complexity of the algorithm is thus linear w.r.t. the size of d provided that the validation against base types is also linear.

Dealing with mixed trees. We now generalize the validation algorithm of the previous section to the case of mixed terms. So a type definition may contain rules of the form $T \to l\{r\}$, where r is a multiplicity list. The validation algorithm is similar, just the order of $d_1, \ldots, d_k$ within $l\{d_1 \cdots d_k\}$ does not matter.

validate(d, T) :
  IF T is a type constant THEN
    check whether d is a basic constant in $[\![T]\!]$ and return the result
  ELSE (T is a type variable)
  IF d is a basic constant then return *false*
  ELSE
  IF the rule for T in D is of the form $T \to l\{r\}$ THEN
    IF d is of the form $l[d_1 \cdots d_k]$ $(k > 0)$ THEN return *false*
    ELSE
      let $d = d\{d_1 \cdots d_k\}$ $(k \geq 0)$,
      let $T_i = type(d_i, r)$ for $i = 1, \ldots k$,

let N be the set of the type names occurring in r
(notice that according to the definition of $type(d_i, r)$
each $T_i \in N \cup \{S_0\}$, where $S_0 \notin N$),
for each $S \in N \cup \{S_0\}$ count the number n_S of the occurrences of S
in $T_1 \cdots T_n$,
IF $n_{S_0} = 0$ and for each $S(i:j)$ occurring in the multiplicity list r
$i \leq n_S \leq j$ THEN return $\bigwedge_{i=1}^{k} validate(d_i, T_i)$ ELSE return *false*
ELSE
IF the rule for T in D is of the form $T \to l[r]$ THEN
IF d is of the form $l\{d_1 \cdots d_k\}$ $(k > 0)$ THEN return *false*,
ELSE (now as in the previous algorithm)
let $d = c[d_1 \cdots d_k]$ $(k \geq 0)$,
let $T_i = type(d_i, r)$ for $i = 1, \ldots k$,
IF $T_1 \cdots T_k \notin L(r)$
THEN return *false*
ELSE
return $\bigwedge_{i=1}^{k} validate(d_i, T_i)$,
ELSE (no rule for T in D)
return *false*.

As in the previous case, the algorithm is linear.

4 Checking Type Inclusion

The main subject of this section is an algorithm for checking type inclusion. Before presenting the algorithm, we introduce some auxiliary notions. A simpler algorithm for a more restricted class of type definitions was presented in [12].

A natural concept of subtyping is based on set inclusion.

Definition 10. *A type S (with a definition D) is an* **inclusion subtype** *of type T (with a definition D') iff $[\![S]\!]_D \subseteq [\![T]\!]_{D'}$.*

We will denote this as $S \subseteq T$, provided D, D' are clear from the context.

In this section we show an algorithm for checking type inclusion. Assume that we want to check $S \subseteq T$ for some types defined by proper type definitions D, D' respectively. We assume that for each type constants C, C' from these definitions we know whether $[\![C]\!] \subseteq [\![C']\!]$ and $[\![C]\!] \cap [\![C']\!] = \emptyset$. We also assume that for each tuple of type constants $C, C_1, \ldots, C_n$ (where $[\![C_1]\!], \ldots, [\![C_n]\!]$ are pairwise disjoint) we know whether $[\![C]\!] \subseteq [\![C_1]\!] \cup \cdots \cup [\![C_n]\!]$. These facts can be recorded in tables. Notice that in the latter case it is sufficient to consider only such $C_1, \ldots, C_n$ for which $[\![C]\!] \cap [\![C_i]\!] \neq \emptyset$ for $i = 1, \ldots, n$. (If some formalism is used to define the sets corresponding to (some) type constants then we require that algorithms for the checks above are given.)

By a **useless symbol** in a regular expression r over an alphabet Σ we mean a symbol $a \in \Sigma$ not occurring in any string $x \in L(r)$. Notice that if r does not contain the regular expression ϕ then r does not contain useless symbols. A type name T is **nullable** in a type definition D if $[\![T]\!]_D = \emptyset$.

To introduce our inclusion checking algorithm we need some auxiliary notions. For a pair of type variables S, T let us define a set $C(S,T)$ as the smallest (under $\subseteq$) set of pairs of type variables such that

- if $label_D(S) = label_{D'}(T)$ then $(S,T) \in C(S,T)$,
- if
  - $(S',T') \in C(S,T)$,
  - D, D' contain, respectively, rules $S' \to l[r_1]$ and $T' \to l[r_2]$, or $S' \to l\{r_1\}$ and $T' \to l\{r_2\}$ (with the same l),
  - type variables S'', T'' occur respectively in r_1, r_2, and $label_D(S'') = label_{D'}(T'')$

  then $(S'',T'') \in C(S,T)$. If D, D' are proper then for every S'' in r_1, there exists at most one T'' in r_2 satisfying this condition, and vice versa.

$C(S,T)$ is the set of pairs of types which should be compared in order to find out whether $S \subseteq T$.

$C(S,T)$ can be computed in time $O(kn^2 \log(kn))$, where n is the number of rules in the definitions and k is the maximal size of a regular expression in the definitions. There are examples of D, D' where $C(S,T)$ contains all the pairs of type variables form D, D' respectively.

Consider a type variable T in a type definition D. The unique rule $T \to l\alpha_{T,D} r_{T,D} \beta_{T,D}$ in D for T (where $\alpha_{T,D}\beta_{T,D}$ is $[\,]$ or $\{\}$) determines the regular expression $r_{T,D}$ and the parentheses $\alpha_{T,D}\beta_{T,D}$. When the parentheses are $[\,]$ then we are interested in the sequences of root labels in all children of the root l of the data terms in $[\![T]\!]_D$. This *label language* is defined as follows. For a given regular expression r

$$LL_D(r) = \left\{ l_1, \ldots, l_n \;\middle|\; \begin{array}{l} T_1 \cdots T_n \in L(r) \text{ and for } i = 1, \ldots, n \\ l_i = label_D(T_i) \text{ if } T_i \text{ is a type variable} \\ l_i \in [\![T_i]\!] \text{ if } T_i \text{ is a type constant} \end{array} \right\}$$

We often skip the subscript in LL_D when it is clear from the context.

For rules with parentheses $\{\}$ we will deal with permutations of the strings from label languages. For any language L we define

$$perm(L) = \{\, x \mid x \text{ is a permutation of some } y \in L \,\}.$$

Now we discuss some necessary conditions for type inclusion and show that they are also sufficient. Assume that D does not contain nullable symbols, the regular expressions in D do not contain useless symbols and D' is proper. Let $S, T \in \mathcal{V}$ and $[\![S]\!]_D \subseteq [\![T]\!]_{D'}$. Then 1. $label_D(S) = label_{D'}(T)$ and 2. $\alpha_{S,D} = \alpha_{T,D'}$. 3. If $\alpha_{S,D}\beta_{S,D} = [\,]$ then $LL_D(r_{S,D}) \subseteq LL_{D'}(r_{T,D'})$. 4. If $\alpha_{S,D}\beta_{S,D} = \{\}$ then $perm(LL_D(r_{S,D})) \subseteq perm(LL_{D'}(r_{T,D'}))$ (equivalently $LL_D(r_{S,D}) \subseteq perm(LL_{D'}(r_{T,D'}))$.

These inclusions (and the fact that D' is proper and $r_{S,D}$ does not contain useless symbols) imply that for every type variable X in $r_{S,D}$ there exists a unique type variable Y in $r_{T,D'}$, such that $(X,Y) \in C(S,T)$ (i.e. such that $label_D(X) = label_{D'}(Y)$). This holds for both kinds of parentheses in the rules for S, T. Moreover, $[\![X]\!]_D \subseteq [\![Y]\!]_{D'}$, as $[\![r_{S,D}]\!]_D \subseteq [\![r_{T,D'}]\!]_{D'}$. Thus, by induction,

conditions $2, 3, 4$ hold for each pair of type variables from $C(S,T)$. (Condition 1 follows from the definition of $C(S,T)$).

On the other hand, assume that $2, 3, 4$ hold for each pair from $C(S,T)$. Take an $(X,Y) \in C(S,T)$ and assume that $X \to_D l[X_1 \cdots X_n] \to_D^* l[X_1' \cdots X_n']$, where $X_1' \cdots X_n'$ is obtained from $X_1 \cdots X_n$ by replacing the type constants occurring in $X_1 \cdots X_n$ by basic constants. Then there exist $Y_1, \ldots, Y_n, Y_1', \ldots, Y_n'$ such that $Y \to_D l[Y_1 \cdots Y_n] \to_D^* l[Y_1' \cdots Y_n']$, and for each $i = 1, \ldots, n$, $(X_i, Y_i) \in C(S,T)$ and $(X_i, Y_i) = (X_i', Y_i')$, or X_i, Y_i are type constants and $X_i' = Y_i' \in [\![Y_i]\!]$. An analogical property holds when $[\,]$ is replaced by $\{\}$. By induction we obtain that whenever a data term is derived from X in D then it is derived from Y in D'. Thus $[\![X]\!]_D \subseteq [\![Y]\!]_{D'}$.

These considerations result in the following algorithm for checking type inclusion and the proposition expressing its correctness and completeness.

inclsubtype(S, T) :

IF S, T are type constants THEN return $[\![S]\!] \subseteq [\![T]\!]$

IF one of the symbols S, T is a type constant and the other type variable THEN return *false*

ELSE IF $label_D(S) \neq label_{D'}(T)$ THEN return *false*
ELSE For each pair $(X,Y) \in C(S,T)$ do the following:

Let $X \to l\alpha r_{X,D}\beta \in D$ and $Y \to l\alpha' r_{Y,D'}\beta' \in D'$ (where $\alpha, \beta, \alpha', \beta'$ are parentheses) be the rules for X and Y in D, D'.

IF $\alpha\beta \neq \alpha'\beta'$ THEN return *false*

ELSE IF $\alpha\beta = \alpha'\beta' = [\,]$ THEN check whether

$$LL_D(r_{X,D}) \subseteq LL_{D'}(r_{Y,D'})$$

ELSE ($\alpha\beta = \alpha'\beta' = \{\}$) check whether

$$LL_D(r_{X,D}) \subseteq perm(LL_{D'}(r_{Y,D'})).$$

IF the checks for all the pairs from $C(S,T)$ succeed THEN return *true*
ELSE return *false*.

Proposition 1. *Let D, D' be type definitions and S, T type names.*
If inclsubtype(S,T) returns true then $[\![S]\!]_D \subseteq [\![T]\!]_{D'}$.
Assume that D has no nullable symbols, the regular expressions in D have no useless symbols and D' is proper. If $[\![S]\!]_D \subseteq [\![T]\!]_{D'}$ then inclsubtype(S,T) returns true.

What remains is to check inclusion for label languages. An algorithm for $LL(r) \subseteq LL(r')$ follows directly from the proposition below.

Proposition 2. *Let r, r' be regular expressions occurring respectively in type definitions D, D'. Let D' be proper and r do not contain useless symbols. For each type constant C from r consider those type constants $C_{C,1}, \ldots, C_{C,k_C}$ from r' for which $[\![C]\!] \cap [\![C_{C,i}]\!] \neq \emptyset$. Let r_C be the regular expression $C_{C,1}| \cdots |C_{C,k_C}$.*
Let s be r with each type constant C replaced by r_C and each type variable T replaced by $label_D(T)$. Let s' be r' with each type variable T replaced by $label_{D'}(T)$.

Then

$$LL_D(r) \subseteq LL_{D'}(r') \quad \textit{iff} \quad L(s) \subseteq L(s'), \textit{ and } [\![C]\!] \subseteq \bigcup_{i=1}^{k_C} [\![C_{C,i}]\!] \textit{ for each } C \in \mathcal{C} \textit{ occurring in } r.$$

Proof. We skip the easy "if" part of the proof. Assume that $LL_D(r) \subseteq LL_{D'}(r')$. This means that for each $w \in L(r)$, $LL_D(w) \subseteq LL_{D'}(r')$. Hence for each type variable T occurring in w there occurs a type variable U in r' such that $label_D(T) = label_{D'}(U)$. Also, for each type constant C in w and each $c \in [\![C]\!]$ there is a type constant C' in r' (thus $C' \in \{ C_{C,1}, \ldots, C_{C,k_C} \}$) such that $c \in [\![C']\!]$. Hence $[\![C]\!] \subseteq \bigcup_{i=1}^{k_C} [\![C_{C,i}]\!]$. Moreover, each $c \in [\![C]\!]$ is a member of at most one $[\![C_{C,i}]\!]$ (due to D' being proper). This holds for each type constant C from r, as each such C occurs in some $w \in L(r)$ (because r has no useless symbols).

Take an $x \in L(s)$. The string x can be obtained from some string $w \in L(r)$ by replacing each type variable T by $label_D(T)$ and each occurrence of any constant C by some $C_{C,j}$. From $LL_D(r) \subseteq LL_{D'}(r')$ it follows that if $w = T_1 \cdots T_n \in L(r)$ then $U_1 \cdots U_n \in L(r')$, where, for $i = 1, \ldots, n$, either both T_i, U_i are variables and $label_D(T_i) = label_{D'}(U_i)$ or both are type constants and $U_i = C_{T_i,j}$, for some j. Hence each $x \in L(s)$ can be obtained from a string $U_1 \cdots U_n \in L(r')$ by replacing each type variable U_i by $label_{D'}(U_i)$. Thus $L(s) \subseteq L(s')$. □

For the inclusion checking algorithm we also need a method for checking whether

$$LL_D(r) \subseteq perm(LL_{D'}(r')) \tag{1}$$

for given multiplicity lists r, r'. Inclusion (1) implies the following conditions.

1. For each $T(m\,{:}\,n)$ occurring in r, where T is a type variable and $n > 0$, r' contains $U(m'\,{:}\,n')$ such that $label_D(T) = label_{D'}(U)$ and $m' \leq m$, $n \leq n'$.
2. For each $U(m'\,{:}\,n')$ occurring in r', where U is a type variable and $m' > 0$, r contains $T(m\,{:}\,n)$ such that $label_D(T) = label_{D'}(U)$ and $m' \leq m$, $n \leq n'$.
3. For each $C(m\,{:}\,n)$, $C \in \mathcal{C}$, occurring in r, if $n > 0$ then $[\![C]\!] \subseteq \bigcup_{i=1}^{k_C} [\![C_{C,i}]\!]$, where $C_{C,1}, \ldots, C_{C,k_C}$ are as in Proposition 2.
4. For each $C'(m'\,{:}\,n')$, $C' \in \mathcal{C}$, occurring in r', let $B_{C',1}, \ldots, B_{C',l_{C'}}$ ($l_{C'} \geq 0$) be those type constants of r for which $[\![B_{C',j}]\!] \cap [\![C']\!] \neq \emptyset$. Let $B_{C',1}(g_1\,{:}\,h_1), \ldots, B_{C',l_{C'}}(g_{l_{C'}}\,{:}\,h_{l_{C'}})$ be (the corresponding) subexpressions of r. Let $g'_j = g_j$ if $[\![B_j]\!] \subseteq [\![C']\!]$ and $g'_j = 0$ otherwise. Then

$$m' \leq \sum_{i=1}^{l_{C'}} g'_i \quad \text{and} \quad \sum_{i=1}^{l_{C'}} h_i \leq n'$$

To justify the last condition, notice that an $x \in LL(B_{C',j}(g_j\,{:}\,h_j))$ contains at most h_j and at least g'_j constants from $[\![C']\!]$.

Conversely, notice that the conditions $1, 2, 3, 4$ imply (1). Thus checking (1) boils down to checking $1, 2, 3, 4$ This completes the description of our inclusion checking algorithm.

Now we discuss the complexity of the inclusion checking algorithm. All the facts concerning the sets corresponding to type constants are recorded in tables, and

can be checked in constant time. Checking inclusion of label languages is invoked by *inclsubtype* at most $|C(S,T)|$ times. The latter number is polynomial w.r.t. the size of the definitions D, D'. Checking condition (1) is polynomial. Inclusion checking for languages represented by DFA's is also polynomial.[5] However when the languages are represented by regular expressions it may be exponential. If however the regular expressions satisfy the condition of 1-unambiguity [6] (cf. the discussion in Sect. 2.2) then they can be in linear time transformed into DFA's. This makes checking whether $L(s) \subseteq L(s')$ polynomial. We obtain:

Proposition 3. *The presented algorithm for checking type inclusion is polynomial for type definitions in which the regular expressions are 1-unambiguous. In a general case it is exponential (w.r.t. the maximal size of a regular expression).*

5 Extension subtyping

We introduce in this section a different kind of subtyping, which is motivated by the *extension* mechanism of XML Schema. We give a definition and an algorithm for checking whether two given types are in this relation.

In our approach a type T is a set of trees of the form $l(t_1...t_n)$ where l is a label, each t_i is in some specific type T_i, and $label(t_i) = label(t_j)$ iff $T_i = T_j$. It may be desirable to consider another set of trees, obtained by adding children to the trees of T. We assume that the labels of the added children are different from the labels of already existing children. This restriction seems to be in conformance with the extension mechanism of XML Schema.

We will use the standard notion of a (language) homomorphism. By an *erasing homomorphism* we mean a homomorphism h such that $h(a) = a$ or $h(a) = \epsilon$, for any symbol a (from the domain of h).

The concept of extension subtype is formalized by

Definition 11. A set S_1 of data terms is an **extension subtype** of a set S_2 of data terms, denoted $S_1 \preceq S_2$, if $S_1 = S_2$ or there exist proper type definitions D_1, D_2 and a type variable T such that $S_1 = [\![T]\!]_{D_1}$ and $S_2 = [\![T]\!]_{D_2}$ and for each rule $U \to l[r_2]$ or $U \to l\{r_2\}$ of D_2, D_1 contains a rule $U \to l[r_1]$ or, respectively, $U \to l\{r_1\}$ such that $r_2 = h_U(r_1)$ for some erasing homomorphism h_U.

So for each $d_2 \in S_2$ there exists a $d_1 \in S_1$ obtained from d_2 by removing some subtrees, and for each $d_1 \in S_1$ there exists such $d_2 \in S_2$.

Example 4. The following data type definitions define types A and A' such that $[\![A']\!]_{D'} \preceq [\![A]\!]_D$, as it is easy to rename type variables in D' so that the conditions of the definition above are satisfied. (# is a type constant; we may assume that $[\![\#]\!]$ is the set of character strings.)

[5] In order to check whether $L(M) \subseteq L(M')$ for DFA's M, M', construct a product automaton $M \times M'$ and check whether no its reachable state is a pair of a final state of M and a non-final state of M'.

$$D = \{\, A \to address[NSC],\quad N \to name[(FL)|I],\quad I \to inst[\#],\quad S \to street[\#],\quad C \to city[\#]\},\quad F \to first[\#],\quad L \to last[\#]\ \ \}$$

$$D' = \{\, A' \to address[N'S'C'P'],\quad N' \to name[(F'M'L')|I'],\quad I' \to inst[\#],\quad S' \to street[\#],\quad C' \to city[\#]\},\quad F' \to first[\#],\quad M' \to middle[\#],\quad L' \to last[\#],\quad P' \to pcode[T'Z'],\quad Z' \to zip[\#],\quad T' \to country[\#]\ \ \}$$

If T_1, T_2 are both type constants then $[\![T_1]\!]_{D_1} \preceq [\![T_2]\!]_{D_2}$ iff $[\![T_1]\!] = [\![T_2]\!]$. If one of T_1, T_2 is a constant and the other a variable, then $[\![T_1]\!]_{D_1} \not\preceq [\![T_2]\!]_{D_2}$. Otherwise T_1, T_2 are type variables and the following algorithm can be used to check whether $[\![T_1]\!]_{D_1} \preceq [\![T_2]\!]_{D_2}$. The algorithm first constructs the set $C(T_1, T_2)$ of pairs of type variables, as described in Section 4. Then it checks if type T_1 is an extension subtype of type T_2 in the following way.

extsubtype(T_1, T_2) :

IF $C(T_1, T_2) = \emptyset$ THEN return *false*

ELSE IF $C(T_1, T_2)$ contains a pair (T, U) such that one of the variables T, U is nullable (in, respectively, D_1 or D_2) and the other is not THEN return *false*

ELSE IF $C(T_1, T_2)$ contains a pair (T, U) such that the rules $T \to l\alpha_1 r_1 \beta_1$ and $U \to l\alpha_2 r_2 \beta_2$ (where $\alpha_i \beta_i$ is $[\,]$ or $\{\}$, for $i = 1, 2$) for T, U from D_1, D_2, respectively, contain different kind of parentheses (i.e. $\alpha_1\beta_1 \neq \alpha_2\beta_2$) THEN return *false*

ELSE for each $(T_1', T_2') \in C(T_1, T_2)$ do the following:

IF both T_1', T_2' are not nullable THEN

Let $T_1' \to l\alpha r_1 \beta$ and $T_2' \to l\alpha r_2 \beta$ be rules of D_1, D_2, respectively

Let $h\colon \mathcal{V} \cup \mathcal{C} \to \mathcal{V} \cup \mathcal{C} \cup \{\epsilon\}$ be the erasing homomorphism that erases

1. each type variable T such that $label_{D_1}(T) \notin \{\, label_{D_2}(U) \mid U \text{ occurs in } r_2,\ U \in \mathcal{V}\,\}$, and
2. each type constant T such that $[\![T]\!] \neq [\![U]\!]$ for each $U \in \mathcal{C}$ occurring in r_2

(so for all other type names $h(T) = T$).

Now connect the non-erased type names from r_1 to the corresponding ones from r_2. Formally:

Construct a homomorphism $f\colon \mathcal{V} \cup \mathcal{C} \to \mathcal{V} \cup \mathcal{C}$ such that

1. for any type variable T occurring in $h(r_1)$, $f(T) = type_{D_2}(label_{D_1}(T), r_2)$ (this means that $f(T)$ is the type variable U occurring in r_2 such that $label_{D_1}(T) = label_{D_2}(U)$, hence $(T, U) \in C(T_1, T_2)$), and
2. for any type constant T occurring in $h(r_1)$, $f(T) = U$ where U is the type constant occurring in r_2 such that $[\![T]\!] = [\![U]\!]$.

(The previous step assures that $f(T)$ is defined for any type name T occurring in $h(r_1)$. As D_2 is proper, $f(T)$ is unique.)

IF the rules for T_1', T_2' contain [ ] THEN
check whether $L(f(h(r_1))) = L(r_2)$
ELSE (the rules for T_1', T_2' contain {})
check whether the multiplicity lists $f(h(r_1))$, r_2 are permutations of each other.
IF all the checks succeed THEN return *true* ELSE return *false*

Proposition 4. *Let D_1, D_2 be proper type definitions and T_1, T_2 type variables. If extsubtype(T_1, T_2) returns true then $[\![T_1]\!]_{D_1} \preceq [\![T_2]\!]_{D_2}$.*

Proof. Assume that the algorithm returns *true*. Without lack of generality we can assume that the sets of type variables occurring in D_1 and D_2 are disjoint.

For each $(T_1', T_2') \in C(T_1, T_2)$ we create a pair of new rules. Consider the rule $T_1' \to l\alpha r_1\beta$ of D_1 (where $\alpha\beta$ is [ ] or {}), the rule $T_2' \to l\alpha r_2\beta$ of D_2, and the homomorphisms h, f used for (T_1', T_2') by the algorithm. We can assume that $f(T) = T$ for any T not occurring in $h(r_1)$. The new rules are: $A_{T_1', T_2'} = T_2' \to l\alpha f(r_1)\beta$ and $B_{T_1', T_2'} = T_2' \to l\alpha f(h(r_1))\beta$. So the first rule is a renamed rule from D_1; type names from D_1 are replaced by the corresponding ones from D_2, whenever such corresponding name exists. The second one is the rule from D_2 with the regular expression replaced by an equivalent one. Notice that $f(h(r_1)) = h(f(r_1))$.

We construct two definitions

$$D_1' = \{\, A_{T_1', T_2'} \mid (T_1', T_2') \in C(T_1, T_2) \,\} \cup D_1$$
$$D_2' = \{\, B_{T_1', T_2'} \mid (T_1', T_2') \in C(T_1, T_2) \,\}$$

By Def. 11, $[\![T_2]\!]_{D_1'} \preceq [\![T_2]\!]_{D_2'}$. For each $(T_1', T_2') \in C(T_1, T_2)$, $[\![T_2']\!]_{D_1'} = [\![T_1']\!]_{D_1}$ and $[\![T_2']\!]_{D_2'} = [\![T_2']\!]_{D_2}$. Thus $[\![T_2]\!]_{D_1} \preceq [\![T_2]\!]_{D_2}$. □

The algorithm performs checks for equality of languages defined by regular expressions. Such a check can be done by converting the regular expressions into DFA's, minimizing them and comparing. The latter two steps are polynomial and the first, as discussed previously, is polynomial for 1-unambiguous regular expressions. From this fact, and from inspection of *extsubtype*, we conclude:

Proposition 5. *The presented algorithm for checking extension subtyping is polynomial (w.r.t. the size of the type definitions involved) provided the regular expressions in the definitions are 1-unambiguous. Otherwise it is exponential (w.r.t. to the maximal size of a non 1-unambiguous regular expression).*

The notion of extention subtyping introduced here seems interesting and useful, because it is close to the extension mechanism of XML Schema and an efficient checking algorithm for it exists. What is missing, is a completeness proof of the algorithm, i.e. that whenever it returns *false* then indeed the first type is not an extension subtype of the other. We conjecture that the algorithm is complete. The future work is to prove this. In case it turns out to be false, we intend to modify the algorithm (and/or the definition of extension subtyping), in order to develop an efficient sound and complete algorithm for checking extension subtyping.

6 Conclusions

We discussed subtyping of tree-structured data, such as XML and HTML documents. We proposed a type formalism motivated by XML Schema and accommodating two concepts of subtyping: inclusion subtyping (corresponding to XML Schema notion of type restriction) and extension subtyping (motivated by XML Schema's type extension). We provided algorithms for checking both kinds of subtyping. Two restrictions on the type definitions are imposed. To simplify the algorithms, we require that type definitions are proper (cf. Def. 5). For the algorithms to be polynomial, the regular expressions in type definitions should be 1-unambiguous (in the sense of [6]). The restrictions seem acceptable; this opinion needs however practical verification.

References

1. Extensible markup language (XML) 1.0 (second edition), W3C recommendation. `http://www.w3.org/TR/REC-xml`, 2000.
2. XML schema part 0: Primer. `http://www.w3.org/TR/xmlschema-0/`, 2001.
3. A.Brown, M.Fuchs, J. Robie, and P. Wadler. MSL: A model for W3C XML Schema. In *Proc. of WWW10*, 2001.
4. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
5. Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Proceedings of 29th Intl. Conference on Very Large Databases, Berlin, Germany (9th–12th September 2003)*, 2003.
6. A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, May 1998.
7. François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of International Conference on Logic Programming, Copenhagen, Denmark (29th July–1st August 2002)*, volume 2401 of *LNCS*, 2002.
8. H.Common, M.Dauchet, R.Gilleron, F.Jacquemard, D.Lugiez, S.Tison, and M.Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata/`, 1999.
9. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2nd edition, 2001.
10. H. Hosoya, J. Vouillon, and B.C.Pierce. Regular expression types for XML. In *Proc. of the International Conference on Functional Programming*, pages 11–22. ACM Press, 2000.
11. M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. Submitted, 2003.
12. A. Wilk and W. Drabent. On types for XML query language Xcerpt. In *International Workshop, PPSWR 2003, Mumbai, India, December 8, 2003, Proceedings*, number 2901 in LNCS, pages 128–145. Springer Verlag, 2003.