Hans Jürgen Ohlbach
Sebastian Schaffert (Eds.)

LNCS 3208

Principles and Practice of Semantic Web Reasoning

Second International Workshop, PPSWR 2004
St. Malo, France, September 2004
Proceedings

Springer

Hans Jürgen Ohlbach
Sebastian Schaffert (Eds.)

Principles and Practice of Semantic Web Reasoning

Second International Workshop, PPSWR 2004
St. Malo, France, September 6-10, 2004
Proceedings

Springer

Volume Editors

Hans Jürgen Ohlbach
Sebastian Schaffert
Ludwig-Maximilians-Universität
Institut für Informatik
Oettingenstr. 67, 80538 München, Germany
E-mail: {ohlbach, schaffer}@pms.ifi.lmu.de

Library of Congress Control Number: 2004110896

CR Subject Classification (1998): H.4, H.3, I.2, F.4.1, D.2

ISSN 0302-9743
ISBN 3-540-22961-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11317241 06/3142 5 4 3 2 1 0

Preface

The best informal definition of the Semantic Web is maybe found in the May 2001 Scientific American article "The Semantic Web" (Berners-Lee et al.), which says "The Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." People who work on the Semantic Web quite often base their work on the famous "semantic web tower", a product of Tim Berners-Lee's inspiring drawing on whiteboards. The lowest level is the level of character representation (Unicode) and the identification of resources on the Web (URIs). The highest level concerns the problem of trusting information on the Web. Somewhere in the middle of the tower is the logic level. It addresses the problem of representing information on the Web in a way so that inference rules can derive implicit information from explicitly stated information. The workshop "Principles and Practices of Semantic Web Reasoning" (PPSWR 2004) addressed problems on this level. It took place in September 2004 as a satellite event of the 20th International Conference on Logic Programming (ICLP) in St. Malo, France. After PPSWR 2003 in Mumbai, India, it was the second workshop in this series.

This book contains the articles presented at the workshop. The 11 papers were selected from 19 submissions by the program committee which consisted of

- François Bry, University of Munich, Germany
- François Fages, INRIA Rocquencourt, France
- Enrico Franconi, Free University of Bozen-Bolzano, Italy
- Georg Gottlob, University of Vienna, Austria
- Benjamin Grosof, MIT Sloan School of Management, USA
- Carsten Lutz, Dresden University of Technology, Germany
- Nicola Henze, University of Hannover, Germany
- Massimo Marchiori, W3C and University of Venice, Italy
- Hans Jürgen Ohlbach, University of Munich, Germany
- Sebastian Schaffert, University of Munich, Germany
- Michael Schröder, Dresden University of Technology, Germany
- Gerd Wagner, Eindhoven University of Technology, Netherlands
- Howard Williams, Heriot-Watt University, Edinburgh, UK
- Guizhen Yang, University at Buffalo, New York, USA

The main topics that were discussed in the workshop are:

- structures in XML documents:
  in the paper: *'On Subtyping of Tree-Structured Data – A Polynomial Approach'* by François Bry, Włodzimierz Drabent, and Jan Małuszyński;
- querying and updating XML documents:
  in the papers: *'Towards Generic Query, Update, and Event Languages for the Semantic Web'* by Wolfgang May, José Júlio Alferes, François Bry, and *'Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach'* by François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert;

- invoking ontologies into the querying process:
  in the papers: *'Rules and Queries with Ontologies: A Unified Logical Framework'* by Enrico Franconi and Sergio Tessaris, *'Semantic Web Reasoning for Ontology-Based Integration of Resources'* by Liviu Badea, Doina Tilivea, and Anca Hotaran, and *'Static Typechecking of Datalog with Ontologies'* by Jakob Henriksson and Jan Małuszyński;
- manipulating and invoking temporal notions:
  in the papers: *'Reasoning About Temporal Context Using Ontology and Abductive Constraint Logic Programming'* by Hongwei Zhu, Stuart E. Madnick, and Michael D. Siegel, *'Towards a Multi-calendar Temporal Type System for (Semantic) Web Query Languages'* by François Bry and Stephanie Spranger, and *'Calendrical Calculations with Time Partitionings and Fuzzy Time Intervals'* by Hans Jürgen Ohlbach;
- non-monotonic reasoning:
  in the paper: *'DR-DEVICE: A Defeasible Logic System for the Semantic Web'* by Nick Bassiliades, Grigoris Antoniou, and Ioannis Vlahavas;
- Web services:
  in the paper: *'A PDDL Based Tool for Automatic Web Service Composition'* by Joachim Peer.

The PPSWR workshop was supported by the EU Network of Excellence CoLogNet (http://www.colognet.net) and the new EU Network of Excellence REWERSE (http://www.rewerse.net). The four-year program REWERSE (REasoning on the WEb with Rules and SEmantics) includes 27 European research and development organizations, and is supposed to bolster Europe's expertise in Web reasoning systems and applications, particularly Semantic Web systems. It consists of the main working groups: 'Rule Markup Languages', 'Policy Language, Enforcement, Composition', 'Composition and Typing', 'Reasoning-Aware Querying', 'Evolution', 'Time and Location', 'Adding Semantics to the Bioinformatics Web', and 'Personalized Information Systems'. The papers in this volume reflect most of the topics in REWERSE.

PPSWR 2005 is scheduled for September 12–16, 2005. It will take place as a 'Dagstuhl Seminar' in the Dagstuhl castle in Germany (http://www.dagstuhl.de/05371/).

September 2004 Hans Jürgen Ohlbach and Sebastian Schaffert

Table of Contents

On Subtyping of Tree-Structured Data: A Polynomial Approach

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

[1] Institut für Informatik, Ludwig-Maximilians-Universität München, Germany
[2] Institute of Computer Science, Polish Academy of Sciences, Warszawa, Poland
[3] Department of Computer and Information Science, Linköping University, Sweden
francois.bry@ifi.lmu.de drabent@ipipan.waw.pl 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 [5, 2] 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 [13] 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 [8] and inclusion subtyping is discussed. Checking of the subtyping relations can then be reduced to checking

H.J. Ohlbach and S. Schaffert (Eds.): PPSWR 2004, LNCS 3208, pp. 1–18, 2004.

inclusion of sets specified by regular tree grammars [6]. 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 [8] 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. We 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.

Detailed comparison of our formalism with those of XML Schema and of other schema languages is outside of the scope of this paper and is a subject of future work. Also we do not deal here with relation of these formalisms with ontologies; roughly speaking the former deal with the syntax of documents and the latter with semantics [9].

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.t. 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*[1] 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 2 below). It should also be noticed that *all* group of XML Schema [13] 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 2. 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

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

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

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

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 [8] (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 3. A **regular type expression** is a regular expression (see e.g. [7]) over the alphabet $\mathcal{V} \cup \mathcal{C}$.

Thus ϵ, ϕ 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 $\mathcal{V} \cup \mathcal{C}$: $L(\epsilon) = \{\epsilon\}$, $L(\phi) = \emptyset$, $L(T) = \{T\}$, $L((\tau_1\tau_2)) = L(\tau_1)L(\tau_2)$, $L((\tau_1|\tau_2)) = L(\tau_1) \cup L(\tau_2)$, and $L((\tau^*)) = L(\tau)^*$. We adopt the usual notational conventions [7], where the parentheses are suppressed by assuming the following priorities of operators: $*$, concatenation, $|$.

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 exist 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. [7]). In the worst case the latter transformation may exponentially increase the number of states. Brüggemann-Klein and Wood [4] 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 [12] 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 [12]. 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 [4]. 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 $\tau^n|\tau^{n+1}|\cdots|\tau^m$, 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 4. 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 $s_1, \ldots, s_k$ 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 $T_1, \ldots, T_m$. 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 5. A **type definition** D for (distinct) type variables $T_1, \ldots, T_n$, for $n \geq 1$ is a set of **rules** $\{R_1, \ldots, R_n\}$ where each rule R_i is of the form

$$T_i \to E_i$$

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

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 $T \to l[r]$ 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 $T \to l\{q\}$ describe a family T of trees where the children of the root are unordered. The ordering of the children is thus irrelevant and the full power of regular expressions is not needed. 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 6. 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.

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

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

Definition 5 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 $\mathit{Person}, \mathit{Father}$ 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 [12]).

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 7. The type definition D of Definition 5 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 6 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 8. 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 $n \geq 0$ 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 9 (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 10. Let D be a type definition for $T_1, \ldots, T_n$. 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 11 (of $label_D(T)$ and $type_D(t, r)$). Notice that:

- Every type variable T in D has only one rule defining it, the label of this rule will be denoted $label_D(T)$.
- Assume that D is proper. Thus given a term $l[t_1 \cdots t_n]$ and a rule $T \rightarrow l[r] \in D$, or a term $l\{t_1 \cdots t_n\}$ and a rule $T \rightarrow l\{r\} \in D$, for each t_i the root of t_i determines at most one type name S occurring in r such that
  - S is a type variable and $label_D(S) = root(t_i)$, or
  - S is a type constant and $t_i \in [\![S]\!]$.

  (In the first case t_i is not a basic constant, in the second it is.) Such type name S will be denoted $type_D(t_i, r)$. If such S does not exist, we assume that $type_D(t_i, r) = S_0$, 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 12. Consider the following type definition D (which is proper):

$$\begin{aligned} &Person \rightarrow person[Name\ (M|F)\ Person(0:2)] \\ &Name \rightarrow name[\#name] \\ &M \rightarrow m[\,] \\ &F \rightarrow f[\,] \end{aligned}$$

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

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

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

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

Our type definitions are similar to those of [8]. 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 [8] 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 [8] 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 [8].

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 [10]. 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 [10].

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 $T \rightarrow \{r\}$). 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 $O(|r| \cdot |x|)$ [1]. 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 there is no rule for T in D THEN return *false*

ELSE IF $root(d) \neq label(T)$ THEN return *false*
ELSE IF the rule for T in D is of the form $T \rightarrow l\{r\}$ 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 $T \rightarrow l[r]$ 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 [11].

A natural concept of subtyping is based on set inclusion.

Definition 13. *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