Towards Types for Web Rule Languages

Włodzimierz Drabent[1,2]

[1] Institute of Computer Science, Polish Academy of Sciences,
ul. Ordona 21, Pl – 01-237 Warszawa, Poland
[2] Department of Computer and Information Science, Linköping University,
S – 581 83 Linköping, Sweden
drabent@ipipan.waw.pl

Abstract. Various schema languages have been introduced to describe (classes of) Web documents (DTD, XML Schema, Relax NG). We present mathematical treatment of their main features. We are interested in the sets of documents a schema defines; such sets will be called types. Using a mathematical formalism makes it possible to discuss chosen aspects of a schema language in a precise and simple way. Otherwise they are hidden among numerous details of a large and sophisticated schema language.

Our goal is typing of rule languages, more precisely approximately describing their semantics by means of types. Thus we are interested in formalisms for types that facilitate constructing (efficient) algorithms performing those operations on types that are needed in type checking and type inference for rules.

1 Introduction

Various schema languages have been introduced to describe (classes of) Web documents (DTD [10], XML Schema [11], Relax NG [6]). We present mathematical treatment of their main features. We are mainly interested in the sets of documents a schema defines; such sets are sometimes called types. Using a mathematical formalism makes it possible to discuss chosen aspects of a schema language in a precise and simple way. Otherwise they are hidden among numerous details of a large and sophisticated schema language.

Our main goal is typing of rule languages; by this we mean describing approximately their semantics by means of types. Such descriptions can be used for finding (certain kinds of) errors in the rules. Knowing that a rule is to be applied to data from a given type, we can compute the type of rule results. If the type is incompatible with the expectations of the programmer then there is an error in the rule. Assume that the expectations are formalized by providing a type description of the expected results. Then it can be automatically checked whether the actual results are in the given type. In other words, one can automatically check correctness of a rule with respect to an approximate specification (given by means of types). These ideas apply to sets of rules too. Sometimes the computed types are approximate and the checks are partial (answering "correct" or "maybe incorrect"); still it is useful to have a possibility of automatic checking certain properties of rule programs and of obtaining hints about possible errors.

So we are interested in formalisms for specifying types that facilitate constructing (efficient) algorithms performing those operations on types that are needed in type checking and type inference for rules. Often there is a trade-off between the expressive power of a formalism and the efficiency of the related algorithms.

A related subject is typechecking of XML queries and transformations: checking whether the results of queries (or transformations) applied to XML data from a given type are within an (another) given type. Substantial theoretical work in this area has been done (see [2, 17] and references therein). That work considers formalizations of (substantial subsets of) XQuery and XSLT. Various cases of such typechecking problem have been studied, some of them shown to be undecidable, many others of non-polynomial complexity. Similar difficulties can be expected with dealing with types for Web rule languages. This suggests that one should also look for approximate, but efficient, solutions.

In the next section we introduce an abstraction of XML data as data terms. Section 3 presents a standard formalism of tree automata. Tree automata define sets of trees, or equivalently of terms, where each symbol has a fixed arity. This is too restrictive for modelling Web data, where the number of children of a tree node is not fixed. In other words, we want to deal with unranked terms. In Section 4 we present a generalization of tree automata to sets of unranked terms. The general formalism is rather powerful. It can define sets which cannot be described by means of DTD and XML Schema. Also some related algorithmic problems, for instance inclusion check, are of high complexity. In section 5 we discuss some useful restrictions of the formalism.

2 Semistructured Data

The data on the Web are presented in the form of XML documents. By database researchers a term *semistructured data* is often used [1]. This term is related to the fact that the data format does not follow any database schema; instead the data are to a certain extent self-explanatory.

XML documents can be seen as trees. From our point of view it is convenient to abstract from syntactic details of XML. We define a formal language of data terms to model tree-structured data. Our notion of data terms has been influenced by Xcerpt, a query language for Web data (see e.g. [21]). Data terms are trees. The children of a node of a data term may be ordered or unordered. We will call such trees *mixed trees* to indicate their distinction from both ordered and unordered trees. Node labels of the trees correspond to XML tags and attribute names. An attribute list of an XML tag can be modelled as an unordered set of children, as exemplified later on.

We begin with an alphabet $\mathcal{L}$ of **labels** and an alphabet $\mathcal{B}$ of **basic constants**. We assume that $\mathcal{L}$ and $\mathcal{B}$ are disjoint and countably infinite. Basic constants represent some basic values, such as numbers or strings, while labels are tree constructors, and will represent XML tags and attribute names. In contrast to function symbols of mathematical logic, the labels do not have fixed arities. The generalization of the notion of a term, allowing arbitrary number of arguments of a function symbol, is called *unranked term.* Data terms further generalize unranked terms, as in addition to argument sequences they also permit unordered sets of arguments.

From basic constants and labels we construct data terms for representing mixed trees. The linear ordering of children will be indicated by the brackets $[,]$, while unordered children are placed between 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 $t_1, \dots, t_n$ are $n \geq 0$ data terms, then $l[t_1 \cdots t_n]$ and $l\{t_1 \cdots t_n\}$ are data terms.

A data term which is not a basic constant is called **labelled**. Data terms not containing $\{,\}$ will be called **ordered**.

For a labelled data term $t = l[t_1 \cdots t_n]$ or $t = l\{t_1 \cdots t_n\}$, its **root**, denoted $root(t)$, is l. If t is a basic constant then $root(t) = t$.

Example 2. Consider the following XML element

```
<CD price="15.90" year="1994">
   Praetorius Mass
   <subtitle></subtitle>
   <artist>Gabrielli Consort and Players</artist>
</CD>
```

It can be represented as a data term

$$\begin{array}{l} CD[\, attributes\{\, price[\mathsf{15.90}]\ year[\mathsf{1994}]\,\} \\ \quad \mathsf{Praetorius\_Mass} \\ \quad subtitle[\,] \\ \quad artist[\mathsf{Gabrielli\_Consort\_and\_Players}] \\ \quad] \end{array}$$

where 15.90, 1994, Praetorius_Mass, Gabrielli_Consort_and_Players are basic constants and *attributes*, *price*, *year*, *subtitle*, *artist* are labels.

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. We have chosen to distinguish them to simplify the definition above, and some other definitions and algorithms.

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 $l[t_1 \cdots t_n]$ (or $l\{t_1 \cdots t_n\}$) is not an n-argument function symbol. It has rather a single argument which is a sequence (string) of data terms $t_1, \dots, t_n$ (where $n \geq 0$).

3 Tree Automata

Finite automata (FA) are a simple, important, and well known formalism for defining sets of strings. We present tree automata [13, 7], a generalization of FA for sets of terms.

A *run* of a finite automaton M on an input string x can be seen as an assignment of states to the suffixes of x. The first suffix is x, the last one is the empty string ϵ. The suffix can be understood as the part of x not yet read by M. To the longest suffix x the run assigns the initial state of M. If a state q is assigned to a suffix ay (where a is a single symbol) then the transition function of M determines (from q and a) the state assigned to suffix y. If the state assigned to suffix ϵ is a final state then the run is *accepting.*

If M is deterministic then for each input string there exist exactly one run. This is not the case for nondeterministic finite automata. Moreover, a run of such automaton may be a partial function, not assigning any state to some suffix y of the input string (and to all the suffixes of y). The *language* defined by a FA M is the set of those strings for which there exists an accepting run.

A tree automaton (TA) deals with terms instead of strings. A basic idea is that a run assigns states to subterms of the input term (instead of suffixes of the input string of a FA). We informally describe tree automata and an equivalent formalism of regular term grammars.

We begin with a finite set of function symbols Σ, each symbol $f \in \Sigma$ has its arity $arity(f) \geq 0$. A **bottom-up tree automaton** (buTA) over Σ is a tuple $M = (Q, \Sigma, F, \Delta)$, where Q is a finite set of states, $F \subseteq Q$ is a set of final states and Δ is a set of transition rules, of the form

$$f(q_1, \ldots, q_n) \to q,$$

where $f \in \Sigma,\ q, q_1, \ldots, q_n \in Q$, and $n = arity(f)$. In particular, if f is a constant then the rule is of the form $f \to q$. A run of M on an input term t is constructed by assigning states to subterms of t. (Formally, we have to deal with subterm occurrences, as a term t' may occur many times in t, e.g. when $t = f(t', t')$. For a full definition see e.g. [7].) A state q is assigned to a subterm $f(t_1, \ldots, t_n)$ only if some states $q_1, \ldots, q_n$ are assigned respectively to the terms $t_1, \ldots, t_n$ and Δ contains the rule $f(q_1, \ldots, q_n) \to q$. If there exists such a run assigning a final state $q \in F$ to t then t is accepted by M. The set of the terms accepted by M is called the **tree language** recognized (or defined, or accepted) by M, and denoted $L(M)$.

So a computation starts with assigning states to the constants in t, by applying rules of the form $a \to q$. Then iteratively: having assigned states to to the subterms $t_1, \ldots, t_n$ of a subterm $f(t_1, \ldots, t_n)$, a rule from Δ is applied, if possible, to assign a state to $f(t_1, \ldots, t_n)$.

A bottom-up tree automaton is **deterministic** if there are no two rules with the same left hand side. It turns out that any set defined by a bottom-up tree automaton is defined by a deterministic one. (A deterministic buTA M' equivalent to a given buTA M can be obtained by a construction similar to the standard one used for FA [7]; each state of M' is a set of states of M. This construction may result in exponential growth of the number of states.)

The sets of terms defined by bottom-up tree automata are sometimes called *regular tree languages* (or recognizable tree languages).

We can consider a different kind of tree automata. Instead of starting at the leaves of the tree, the computation may start at the root. By a **top-down tree automaton** (tdTA) we mean a tuple $M = (Q, \Sigma, I, \Delta)$, where Q, Σ are as above, $I \subseteq Q$ is a set of initial states and Δ is a set of transition rules of the form

$$q \to f(q_1, \ldots, q_n),$$

(where $f \in \Sigma,\ q, q_1, \ldots, q_n \in Q$ and $n = \mathit{arity}(f)$). Given an input term t, a run assigns an initial state $q_0 \in I$ to t, and if a state q is assigned to a subterm $f(t_1, \ldots, t_n)$ and a rule $q \to f(q_1, \ldots, q_n)$ is in Δ then states $q_1, \ldots, q_n$ can be respectively assigned to the subterms $t_1, \ldots, t_n$. A run for t is called accepting if it assigns a state to each subterm of t. A term t is accepted by M if there exists an accepting run for t.

A top-down tree automaton is **deterministic** if it has one initial state and has no two rules with the same left hand side and the same function symbol.

The top-down and bottom-up tree automata are equivalent, they define the same class of languages. (For a proof it is sufficient to reverse the rules, and exchange the sets of final and initial states.) Notice that this transformation applied to a deterministic bottom-up automaton does not necessarily produce a deterministic top-down one. Indeed, deterministic top-down tree automata define a proper subset of regular tree languages. For instance, the set $\{f(a, b), f(b, a)\}$ is not defined by any deterministic tdTA.

It is sometimes convenient to view top-down tree automata as grammars (called *regular term grammars* or *regular tree grammars*,[1] see e.g. [8, 7]). Let $M = (Q, \Sigma, I, \Delta)$ be such an automaton. In the corresponding grammar, the states of M become non-terminal (unary) symbols of the grammar, and the initial states become start symbols. We consider terms built out of $\Sigma \cup Q$ and a derivation relation $\Rightarrow$ on such terms: $t_1 \Rightarrow t_2$ iff t_2 is obtained from t_1 by replacing an occurrence of a nonterminal q by a term $f(q_1, \ldots, q_n)$ such that the rule $q \to f(q_1, \ldots, q_n)$ is in Δ. The language generated by the grammar is

$$\{\, t \mid q \Rightarrow^* t,\ q \in I,\ t \text{ does not contain symbols from } Q\,\}.$$

This set is equal to the language accepted by the automaton M. (We skip a proof, based on showing that an accepting run of M assigns a state q to an input term t iff $q \Rightarrow^* t$, for any term t over Σ).

The class of regular tree languages is closed under union, complement and intersection [7]. We briefly outline the proofs. To construct a buTA for the union of regular tree languages L_1, L_2, take two buTA $M_i = (Q_i, \Sigma, F_i, \Delta_i)$ $(i = 1, 2)$ respectively for L_1 and L_2, with disjoint sets of states. Automaton $(Q_1 \cup Q_2,$

[1] However [18] applies this name to a formalism defining sets of unranked trees.

Σ, $F_1 \cup F_2$, $\Delta_1 \cup \Delta_2$) accepts $L_1 \cup L_2$. A TA for $L_1 \cap L_2$ can be obtained by a constructing a product automaton from M_1 and M_2 (and the construction is polynomial). Exchanging the final and non final states in a deterministic buTA for a tree language L results in a buTA for the complement of L.

It can be decided in time $O(|t| \cdot |M|)$ whether a term t is accepted by a TA M (where $|t|, |M|$ are respectively the sizes of t, M). If M is a deterministic buTA (or deterministic tdTA) then the membership can be tested in linear time. Checking whether $L(M) = \emptyset$ is linear, while checking emptiness of the complement of $L(M)$ is EXPTIME-complete. Also checking whether $L(M_1) \subseteq L(M_2)$ is EXPTIME-complete. For details and proofs see [7]. For deterministic tdTA polynomial algorithms for checking $L(M_1) \subseteq L(M_2)$ exist, see e.g. [12, 9].

4 Tree Automata Generalized

Tree automata are not directly applicable to semistructured data. They deal with terms in which each symbol has a fixed arity, while in semistructured data the number of arguments of a symbol is not fixed.

A straightforward solution is to apply a standard way of representing trees as binary trees [16]. In such a binary tree the first child of a node n represents the list of children of n in the original tree, while the second child represents the (tail of the list of) siblings of n.

$$\begin{tikzcd}
{} \arrow[d] & & \\
n \arrow[r] \arrow[dd] & \text{sister of } n \arrow[r] \arrow[d] & \cdots \\
 & \cdots & \\
\text{daughter of } n \arrow[r] \arrow[d] & \text{daughter of } n \arrow[r] \arrow[d] & \cdots \\
\cdots & \cdots &
\end{tikzcd}$$

Such representation is used for instance by [15]. A disadvantage is that the representation obscures the structure of the original tree; the (next) sibling of n is treated in the same way as its (first) child, while the children of n are treated differently. It seems more elegant and clear to provide a formalism to directly describe semistructured data. It turns out that such approach has some actual technical advantages.

There exist various equivalent generalizations of tree automata to unranked terms [18, 3, 15]. They follow a common main idea. In tree automata (or regular tree grammars) the children of a node are described by a single sequence (of states or nonterminals). The generalizations replace a single sequence by a regular language. In this way the formalism is able to specify a set of tree sequences, which are allowed as the children of a given tree node.

Some of the formalisms are formulated as defining sets of trees (e.g. [18]), some other as defining sets of sequences of trees (in other words of ordered forests, or of hedges, e.g. [3, 15]). This difference is inessential, as we may express a sequence $t_1, \ldots, t_n$ of trees as a tree $f(t_1, \ldots, t_n)$, where f is a selected new symbol. The grammatical formalism described below defines sets of trees.

As our abstraction of semistructured data we choose data terms (cf. Section 2). The purpose of this paper is to discuss defining sets of data terms, so we do not consider a way of specifying sets of basic constants. Instead we assume that we have an alphabet $\mathcal{C}$ of **type constants**, and for each $C \in \mathcal{C}$ a corresponding set $[\![C]\!]$ of basic constants is given. The formalism also employs an alphabet $\mathcal{V}$ of **type variables**. The symbols from $\mathcal{V} \cup \mathcal{C}$ will play the role of grammar nonterminals, they will be called **type names**.

A regular language (of strings) over $\mathcal{V} \cup \mathcal{C}$ will be called a *regular type language*. As a way of specifying regular type languages we choose regular expressions; they may be replaced by other formalisms, like (deterministic or nondeterministic) finite automata. By a **regular type expression** we mean a regular expression 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 τ denotes a *regular language* $L(\tau)$ over the alphabet $\mathcal{V} \cup \mathcal{C}$: $L(\varepsilon) = \{\epsilon\}$, $L(\phi) = \emptyset$, $L(T) = \{T\}$, $L((\tau_1\tau_2)) = L(\tau_1)L(\tau_2)$, $L((\tau_1|\tau_2)) = L(\tau_1) \cup L(\tau_2)$, and $L((\tau^*)) = L(\tau)^*$. We adopt the usual notational conventions [14], where the parentheses are suppressed by assuming the following priorities of operators: $*$, concatenation, $|$.

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

- $\tau(n : m)$, or $\tau^{(n:m)}$, where $n \leq m$, 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 3. A **type definition** D is a finite set of rules of the form

$$T \to l[\tau] \quad \text{or} \quad T \to l\{\tau\}$$

where T is a type variable, l is a label, τ a regular expression over $\mathcal{V} \cup \mathcal{C}$ (i.e. a regular type expression), and no two rules with the same T and l occur in D.

The regular expression τ is called the *content model* of the rule. A rule beginning with a type name T is said to be a *rule for* T. The form of the content models in rules of the form $T \to l\{\tau\}$ is restricted, as explained below.

The two kinds of rules are used to distinguish ordered and unordered arguments of a label. A rule $T \to l[\tau]$ describes a family of data terms where the children of the root l are ordered and their sequence is described by the regular expression τ. In the second case the children of l are unordered and we abstract from the order of symbols in the strings from $L(\tau)$. Thus the full power of regular expressions is not needed here. We will usually require that the regular expressions in the rules of the form $T \to l\{\tau\}$ are **multiplicity lists**, i.e. they are 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. A different kind of restrictions is considered in [20, 2].

A type definition defines a set of data terms by means of rewriting of data patterns.

Definition 4. 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, and data patterns may be seen as data terms with some subterms replaced by type names. Now we are ready to define the rewriting relation of a type definition.

Definition 5 (of $\rightarrow_D$). Let D be a type definition and 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, 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 $[\![S]\!]$.

As usually, a sequence $d_1 \rightarrow_D \cdots \rightarrow_D d_n$ is called a derivation of D. Derivation may end with a data term. This gives the semantics for type definitions:

Definition 6. Let D be a type definition. The **type** $[\![T]\!]_D$ associated with a type name T by D is defined as the set of all data terms t that can be obtained from T:

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

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

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

A set S of data terms is called a **type** or a **regular** set if $S = [\![T]\!]_D$ for some type definition D and type name T.

Notice that type definitions generalize regular term grammars. Assuming a fixed arity $arity(l)$ for each label l, a type definition containing only rules of the form $T \rightarrow l[T_1 \cdots T_{arity(l)}]$ is a regular term grammar.

Example 7. Assume that $\#name \in \mathcal{C}$ and consider the following type definition D:

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

Let $\mathsf{john}, \mathsf{mary}, \mathsf{bob} \in [\![\#name]\!]$. 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 $[\![Person]\!]$

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

5 Useful Restrictions of Type Definitions

The formalism of type definitions introduced in the previous section is rather general. This has some disadvantages. For instance, inclusion checking for sets defined by type definitions is EXPTIME-hard. It is interesting to find out classes of type definitions for which some problems can be solved more efficiently. This section presents a few such classes; it is mainly based on the classification proposed by Murata, Lee, and Mani [18]. (A newer version of that paper is [19].) That classification is made from the point of view of membership checking, but it is also useful when other problems are considered. The work [18, 19] dealt only with ordered trees (in our formalism this means ordered data terms). We provide a straightforward generalization of the classification of [18, 19] to mixed trees (i.e. arbitrary data terms).

In what follows we also explain briefly the relation between the discussed classes of definitions, and DTD and XML Schema. It should also be mentioned that the schema language Relax NG [6] is able to define any regular set of ordered data terms (formally, any set of XML documents corresponding to such data term set).

There is already a restriction imposed in the Definition 3, namely that there are no two rules for the same type variable T and with the same label l. This restriction is not severe, as any two rules $T \rightarrow l\alpha\tau_1\beta$, $T \rightarrow l\alpha\tau_2\beta$ with the same T, l and the same parentheses $\alpha\beta$ are equivalent to one rule $T \rightarrow l\alpha\tau_1|\tau_2\beta$. However the restriction implies that in a type $[\![T]\!]_D$ defined by a type definition there cannot occur data terms of the form $l\{\tau\}$ and of the form $l[\tau]$ (with the same l). We expect that that this restriction is not important from a practical point of view.

A natural question arises whether we need multiple rules for one type name.

Definition 8. A type definition D will be called **single-label** if D contains at most one rule for each type name T.

We show that the sets defined by type definitions are finite unions of sets defined by single-label type definitions.

Proposition 9. Let D be a type definition. There exists a single-label definition D' such that for each type variable T occurring in D we have $[\![T]\!]_D = [\![T_1|\cdots|T_n]\!]_{D'}$ for some type names $T_1, \ldots, T_n$.

Proof. Let $T \to l_i\alpha_i\tau_i\beta_i$ $(i = 1, \ldots, n)$ be the rules from D for a type variable T (where $\alpha_i\beta_i$ are parentheses [] or {}). By Definition 3 the labels $l_1, \ldots, l_n$ are distinct. Introduce new type variables $T_1, \ldots, T_n$. Replace the i-th rule above by $T_i \to l_i\alpha_i\tau_i\beta_i$. Replace each occurrence of T in the content model of a rule by $(T_1|\cdots|T_n)$. For the resulting type definition D_T we have $[\![T]\!]_D = [\![T_1|\cdots|T_n]\!]_{D_T}$ and $[\![U]\!]_D = [\![U]\!]_{D_T}$ for all the other type names occurring in D. D' is obtained by repeating this transformation for all the type variables for which rules in D exist.

Consider a type definition D. Following [18] we define a notion of competing type names. Distinct type variables T_1, T_2 are **competing** (w.r.t. D) if D contains rules with T_1 and T_2 as the left hand sides and with the same label l. Distinct type constants C_1, C_2 are *competing* if $[\![C_1]\!] \cap [\![C_2]\!] \neq \emptyset$.

Definition 10. A type definition is called **local** if it does not contain competing type names.

Example 11. Consider a type definition

$$D = \{\, Book \to book[Author^*],\ Author \to man[\#],\ Author \to woman[\#]\,\},$$

where $Book, Author \in \mathcal{V}$, $\# \in \mathcal{C}$, and $book, man, woman \in \mathcal{L}$. No two type names of D are competing, thus D is local. D is not single-label; removing one of the rules for *Author* results in a single-label definition.

The intention for introducing local definitions is simplifying the membership check. If D is local then for any data term t there is at most one type name T_t (occurring in D) such that $t \in [\![T_t]\!]_D$. Thus to check whether $l[t_1\cdots t_n] \in [\![T]\!]_D$ it is sufficient to check, for a single sequence $T_{t_1}, \ldots, T_{t_n}$, whether $t_i \in [\![T_{t_i}]\!]_D$ for $i = 1, \ldots, n$, whether a rule $T \to l[\tau]$ exists in D, and whether $T_{t_1}\cdots T_{t_n} \in L(\tau)$. Checking if $l\{t_1\cdots t_n\} \in [\![T]\!]_D$ is similar.

Sections 3.2 and 5.1 of [18] point out correspondence between DTD's [10] and local type definitions. Indeed, any DTD represented as a data definition is local. This is due to not distinguishing between type variables and labels; each rule of the data definition is of the form $l \to l[\tau]$. On the other hand local definitions are more general than DTD, as they allow different labels for the same type variable, like in Ex. 11. (Papers [18, 19] do not discuss this issue and all the example definitions they use are single-label). So it is more accurate to state that DTD's correspond to data definitions which are local, single-label and do not contain rules with {}.

Example 12. Consider the type definition D from Example 11 and assume that $[\![\#]\!]$ is the set of character strings. Removing the last rule from D results in a type definition D' corresponding to the DTD: `<!ELEMENT book (man*)>` `<!ELEMENT man (#PCDATA)>` .

Definition D does not correspond to any DTD, as $[\![Author]\!]$ contains data terms with two roots, *man* and *woman*. Transforming D into a single-label definition as in the proof of Proposition 9 results in a definition which is not local.

The conditions on local type definitions can be weakened without requiring any substantial modifications of the outlined membership checking algorithm. Namely it is sufficient that, for a given t, in any content model of D there is at most one T such that $t \in [\![T]\!]_D$.

Definition 13. A type definition D is called **single-type** if no content model in a rule of D contains competing type names [18]. A type definition D is **proper** if it is single-type and single-label [22, 5].

Example 14. Consider the type definition D from Example 11 and rules

$$D' = \{\, Library \rightarrow lib\{Reader^*\},\ Reader \rightarrow man[\#],\ Reader \rightarrow woman[\#]\,\}.$$

Definition $D \cup D'$ is single-type but not proper and not local. Removing from $D \cup D'$ the two rules with label *man* results in a proper definition.

Paper [18, 19] explains that the sets defined by XML Schema [11], with exclusion of a few constructs, can be defined by single-type type definitions. One of the excluded constructs is the mechanism of *xsi*:*type*. Actually, "single-type" can be replaced here by "proper", as all the elements of a set defined by an XML Schema have the same main tag.

An important property is that inclusion of sets defined by proper type definitions can be checked in polynomial time. More precisely, [5] presents an algorithm which checks whether $[\![T_1]\!]_{D_1} \subseteq [\![T_2]\!]_{D_2}$, where D_2 is proper and D_1 is arbitrary. (Definition D_1 is required to be single-label, but this restriction can be abandoned.) The algorithm works in time polynomial w.r.t. the sizes of D_1, D_2 and the sizes of deterministic finite automata equivalent to the content models of D_1, D_2. Unfortunately, the latter are exponential w.r.t. the sizes of regular expressions. It is known that construction of deterministic FA is of linear time for *1-unambiguous* regular expressions [4]. Thus inclusion can be checked in polynomial time for type definitions D_1, D_2 with 1-unambiguous content models, where D_2 is proper.

The restrictions above can be further weakened for rules with parentheses [ ], by considering the positions on which type names occur in the strings from $L(\tau)$. We do not discuss this issue here.

The classes discussed in this section can be parameterized by the way the regular languages in the content models are specified. As the discussion above on inclusion checking for proper definitions suggests, an important class of type definitions is that with content models given by deterministic FA (or by regular expressions which can be transformed to such automata in linear or polynomial time). This class is also distinguished in the work on complexity of XML transformations (cf. [17] and the references therein). That work also introduces a class of *bottom-up deterministic* (unranked) tree automata, which in our approach correspond to type definitions such that whenever a definition contains rules $T_1 \to l\alpha_1\tau_1\beta_1$ and $T_2 \to l\alpha_2\tau_2\beta_2$ for distinct T_1, T_2 then $L(\tau_1) \cap L(\tau_2) = \emptyset$.

The class of regular sets of data terms is closed under intersection, union and complementation. The classes of the sets defined by local, single-type and proper type definitions are closed under intersection but not under union (hence not under complementation) [18, 19, 5].

Conclusions

The intention of this text is to introduce the reader to formalisms for defining sets of trees; the intended application is typechecking of rule languages for Web applications. First we presented tree automata as a generalization of finite automata for strings. Tree automata define sets of terms where each function symbol has a fixed arity. This is too restrictive from the point of view of modelling Web data. For this task unranked terms are needed, where arity of symbols is not fixed. We deal with a slightly more general concept of data terms, where the arguments of a symbol can be ordered or unordered. A generalization (called type definitions) of tree automata for data terms is shown in Section 4. Some algorithmic problems, like inclusion check, are of non polynomial complexity already for tree automata. We outlined some restrictions of the formalism; for such restrictions more efficient algorithms exist. We briefly discussed the correspondence of these restrictions to DTD and XML Schema.

Acknowledgement. This research has been partially funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.
2. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. *J. Comput. Syst. Sci.,* 66(4):688–727, 2003.
3. A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, April 2001.
4. A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, May 1998.
5. François Bry, Włodzimierz Drabent, and Jan Małuszyński. On subtyping of tree-structured data: A polynomial approach. In Hans Jürgen Ohlbach and Sebastian Schaffert, editors, *Principles and Practice of Semantic Web Reasoning, Second International Workshop (PPSWR 2004)*, volume 3208 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2004.

6. J. Clark and M. Murata (editors). RELAX NG specification, December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.
7. 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/`, 2002.
8. P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. The MIT Press, 1992.
9. W. Drabent, J. Maluszynski, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4–5):549–610, 2002.
10. Extensible markup language (XML) 1.0 (second edition), W3C recommendation 6 October 2000. `http://www.w3.org/TR/REC-xml`.
11. D. C. Fallside (ed.). XML Schema part 0: Primer. W3C Recommendation, `http://www.w3.org/TR/xmlschema-0/`, 2001.
12. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proc. of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.
13. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer-Verlag, 1997.
14. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 2nd edition, 2001.
15. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *ICFP 2000*, pages 11–22, 2000.
16. Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, second edition, 1973.
17. W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. In *PODS 2004,* pages 23–34, 2004.
18. M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Langages*, Montreal, Canada, 2001. `http://www.cs.ucla.edu/~dongwon/paper/`.
19. M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. Submitted, 2003.
20. F. Neven and T. Schwentick. XML schemas without order. Unpublished, 1999.
21. Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6th August 2004)*, 2004.
22. 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.