The Typechecking Problem for XML Transformations: Methods and Formal Models

Wim Martens

Hasselt University and
Transnational University of Limburg,
Agoralaan, Gebouw D
B-3590 Diepenbeek, Belgium
wim.martens@uhasselt.be

Abstract

The typechecking problem for XML to XML transformations has recently attracted quite some attention in the literature. Given an input schema, an output schema, and an XML to XML transformation, the typechecking problem asks whether the output of the transformation is always conform to the output schema for input documents satisfying the input schema. The present paper takes a detailed look on the two main ingredients of the typechecking problem: formal models of the XML schemas and the XML to XML transformations. Finally, we give an overview of several formal models and techniques that can be used to tackle the typechecking problem.

1 Introduction

XML is a data format in which documents are essentially structured as labeled, ordered trees. It is intended to be a common data format for a very wide range of applications, and aims at being able to model the most diverse forms of data in an intuitive way. Due to this flexibility, XML has become increasingly popular in the last few years, and at the moment, it is quite safe to say that it has become the standard format for data exchange format on the Web.

However, for many applications, it is important to have restrictions on the structure of XML documents. Consider, for instance, automatic processing of XML documents, where prior knowledge of the document structure can greatly facilitate the implementation of the algorithms; or query optimization, which is in many cases not even possible when the structure of the document is not known in advance. These restrictions are imposed by *XML schemas*, which basically describe sets of XML documents.

On the Web, large user communities usually agree on representing their data using the same XML schema in order to facilitate the exchange of doc-

uments within the community. It does happen quite regularly, however, that documents need to be exchanged between two communities with different schemas. Suppose, for instance, that a certain community A with schema S_A, wants to query an XML database D from a certain community B with schema S_B, and publish the results on the world wide web. In order to answer the incoming query, B applies a certain transformation T to the database D, and sends the resulting document $T(D)$ back to A. Before A can automatically process the document $T(D)$ to publish it, the document has to be dynamically validated against the local schema S_A. If this validation fails, the automated processing of the incoming document is hardly possible. Indeed, validity against the local schema can, for example, be a necessary precondition of the algorithms that process the documents.

When many documents are exchanged between the same two communities using similar transformations, this dynamic validation of incoming documents can be a very time-consuming task. This motivates the need for *static typechecking*: given the schema S_B of the sending community, the schema S_A of the receiving community, and a transformation T, we are asked whether for *every* document D in S_B, $T(D)$ is valid with respect to S_A.

Suppose that a static type checker would decide that for every document D in S_A, we have that $T(D)$ is valid with respect to S_B. Then, B would not anymore have to validate the incoming documents coming from A which are transformed by transformation T. Even better, A can update its document D, and as long as D is valid with respect to S_A, B still does not have to validate the document $T(D)$ coming from A. It is clear that, in such as setting, a static type checker can save an important amount of time.
The static typechecking problem has been investigated in several varieties:

Complete Versus Incomplete Typechecking: Of course, the typechecking problem is undecidable when the transformation language is Turing complete. In this case, our best hope is to approximate the answers to the typechecking problem as well as possible. The usual approach that many typecheckers take, is that they are sound, but not complete. This means that the typechecker only passes transformations that actually typecheck, but it sometimes also unjustly rejects a transformation. The aim is of course to minimize these unjust rejections. We refer to [17] for an excellent tutorial on this area of XML typechecking.

Data Values Versus No Data Values: Sound and complete typechecking for XML transformations which have the ability to compare data values seems to be a very difficult problem. Indeed, even for very restricted cases, the typechecking problem becomes undecidable [1]. However, many XML schemas do not constrain the data values in the documents, and the behavior of many filtering or restructuring transformations does not depend on the actual data values.

In this paper, we focus on sound and complete typechecking algorithms

for abstractions of XML documents in which data values are not considered. The purpose of this paper is twofold. First, we visit the two components of the typechecking problem: the XML schema languages and the XML to XML transformations. We give a detailed overview on formal models for several XML schema languages, such as DTD [2] and XML Schema [23], and discuss some of their properties and alternative characterizations, which are meant to give the reader insight into their exact expressive power. We also take a detailed look at sevaral formalisms that have been used to model XML to XML transformations. Second, we want to to present several useful tools for obtaining complexity/decidability results on the typechecking problem for XML transformations.

The outline of this paper is as follows. In Section 2, we introduce some necessary notation and we formally define the typechecking problem. Section 3 discusses the first main ingredient of the typechecking problem: the schema languages. Among others, we provide formal models for the expressive power of DTDs [2] and XML Schema [23]. We also discuss several alternative characterizations of these formal models which give more insight in their exact expressive power. Finally, we discuss closure properties of these XML schema languages under Boolean operations. The second main ingredient of the typechecking problem is discussed in Section 4: the XML to XML transformations. We treat two formalisms in detail, and give an overview of other important formalisms that have been used in the past. In Section 5, we provide a high-level overview of some techniques that have been used to obtain complexity/decidability results on the typechecking problem. We treat complexity upper bounds as well as complexity lower bounds.

2 Preliminaries

2.1 Trees

In this section we provide the necessary background on trees, automata, and tree transducers. In the remainder of the paper, Σ always denotes a finite alphabet.

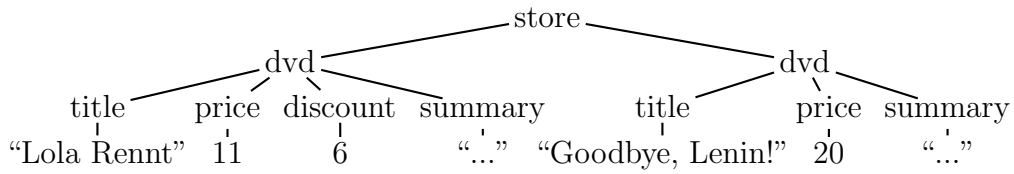
It is common to view XML documents as finite trees with labels from a finite alphabet Σ. There is no limit on the number of children of a node. Figure 1 gives an example of an XML document together with its tree representation. Of course, elements in XML documents can also contain references to nodes. But as XML schema languages usually do not constrain these nor the data values at leaves, it is safe to view schemas as simply defining tree languages over a finite alphabet.

We assume familiarity with DFAs (Deterministic Finite Automata), NFAs (Non-deterministic Finite Automata), and REs (Regular Expressions). Given a DFA, NFA, or RE A, we denote the language accepted by A by $L(A)$.

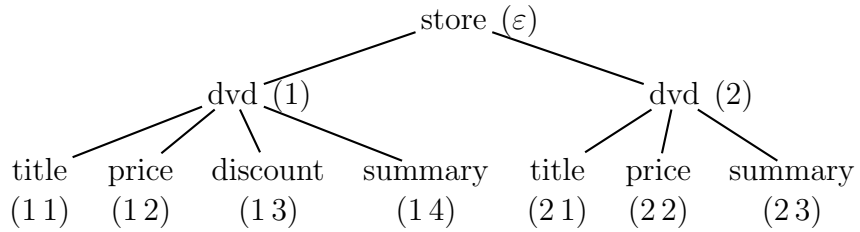
Throughout this paper, Σ always denotes a finite alphabet. The set of

```
<store>
  <dvd>
    <title> "Lola Rennt" </title>
    <price> 11 </price>
    <discount> 6 </discount>
    <summary> ... </summary>
  </dvd>
  <dvd>
    <title> "Goodbye, Lenin!" </title>
    <price> 20 </price>
    <summary> ... </summary>
  </dvd>
</store>
```

(a) An example XML document.



(b) Its tree representation with data values.



(c) The tree of Figure 1(b) without data values. The nodes are annotated next to the labels, between brackets.

Fig. 1. An example of an XML document and its tree representation.

unranked Σ*-trees*, denoted by $\mathcal{T}_\Sigma$, is the smallest set of strings over Σ and the parenthesis symbols "(" and ")" such that, for $a \in \Sigma$ and $w \in \mathcal{T}_\Sigma^*$, $\sigma(w)$ is in $\mathcal{T}_\Sigma$. So, a tree is either ε (empty) or is of the form $a(t_1 \cdots t_n)$ where each t_i is a tree. In the tree $a(t_1 \cdots t_n)$, the subtrees $t_1, \ldots, t_n$ are attached to the root labeled a. We write a rather than $a()$. Note that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked*. For every $t \in \mathcal{T}_\Sigma$, the *set of nodes* of t, denoted by $\mathrm{Dom}(t)$, is the set defined as follows:

(i) if $t = \varepsilon$, then $\mathrm{Dom}(t) = \emptyset$; and

(ii) if $t = a(t_1 \cdots t_n)$, where each $t_i \in \mathcal{T}_\Sigma$, then $\mathrm{Dom}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{iu \mid$

$u \in \mathrm{Dom}(t_i)\}$.

Figure 1(c) contains a tree in which we annotated the nodes between brackets. Observe that the n child nodes of a node u are always $u1, \ldots, un$, from left to right. The *label* of a node u in the tree $t = a(t_1 \cdots t_n)$, denoted by $\mathrm{lab}^t(u)$, is defined as follows:

(i) if $u = \varepsilon$, then $\mathrm{lab}^t(u) = a$; and

(ii) if $u = iu'$, then $\mathrm{lab}^t(u) = \mathrm{lab}^{t_i}(u')$.

In the sequel, whenever we say tree, we always mean Σ-tree. A *tree language* is a set of trees. In the sequel, we will use the letters $t, t_1, t_2, \ldots$ to denote trees.

2.2 The Typechecking Problem

We define a *tree transformation* to be a mapping $T : \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$.

Definition 2.1 Let T be a tree transformation and S_{in} and S_{out} be two tree languages. We say that T *typechecks with respect to S_{in} and S_{out}*, if for every $t \in S_{\text{in}}, T(t) \in S_{\text{out}}$.

Definition 2.2 Given S_{in}, S_{out}, and T, the *typechecking problem* consists in deciding whether T typechecks with respect to S_{in} and S_{out}.

Of course, the typechecking problem is undecidable when arbitrary (that is, Turing complete) classes of tree transformations and arbitrary tree languages are used. We therefore restrict these notions in the following sections.

3 Formal Models for XML Schema Languages

We define several common restrictions on tree languages: DTDs, extended DTDs, single-type extended DTDs and restrained competition extended DTDs. Later in this section, we discuss alternate characterizations of these schema languages and their closure properties under Boolean operations.

3.1 The Schema Languages

DTD is the most widely used XML schema language in practice and only imposes *local* restrictions on trees [2]. That is, they only impose restrictions between the label of a node and the labels of its children. We abstract DTDs as extended context-free grammars:

Definition 3.1 A *DTD* is a pair (d, s_d) where d is a function that maps Σ-symbols to regular languages over Σ and $s_d \in \Sigma$ is the start symbol. A tree t *satisfies* d if its root is labeled by s_d and, for every node u with label a, the sequence $a_1 \cdots a_n$ of labels of its children is in $L(d(a))$. By $L(d)$ we denote the set of trees that satisfy d.

We usually denote (d, s_d) by d when the start symbol s_d is understood from the context. We parameterize the definition of DTDs by a class of representations $\mathcal{M}$ of the internal regular string languages such as, for instance, the class of DFAs (Deterministic Finite Automata), NFAs (Non-deterministic Finite Automata), or REs (Regular Expressions). For instance, DTD(DFA) is the class of DTDs, in which the regular languages in the definition of d are represented by DFAs. In examples, we denote the regular languages in the definition of a DTD by regular expressions. For clarity, we write $a \to r$ rather than $d(a) = L(r)$. We also do not list rules of the form $a \to \varepsilon$.

Example 3.2 In this notation, a simple example of a DTD(RE) defining the inventory of a store which sells DVDs is the following:

$$\begin{aligned} \text{store} &\to \text{dvd dvd}^* \\ \text{dvd} &\to \text{title price discount? summary} \end{aligned}$$

where the start symbol is "store". The DTD defines trees of depth 3, where the root is labeled with "store" and has one or more children labeled with "dvd". Every "dvd"-labeled node has three or four children. From left to right, these children are labeled "title", "price", "discount" (which is the optional child), and "summary", respectively. The tree in Figure 1(c) is in the language defined by this DTD.

In this paper, we assume that DTDs do not contain *useless* symbols, unless mentioned otherwise (such as in Propositions 5.1, 5.2, and 5.3). That is, we assume that, for every DTD d and for every $a \in \Sigma$, there exists a tree $t \in L(d)$ and a node $u \in \mathrm{Dom}(t)$ such that $\mathrm{lab}^t(u) = a$.

We now turn to a more expressive formalism for XML schema languages: extended DTDs [20]. [1] The class of tree languages defined by extended DTDs corresponds precisely to the regular (unranked) tree languages [3]. We recall the definition of extended DTDs:

Definition 3.3 ([20]) An *extended DTD* (EDTD) is a 4-tuple $\mathbf{d} = (\Sigma, \Sigma', d, \mu)$, where Σ' is an alphabet of *types*, d is a DTD over Σ', and μ is a mapping from Σ' to Σ. Note that μ can be extended to define a homomorphism on trees. A tree t then *satisfies* an extended DTD if $t = \mu(t')$ for some $t' \in L(d)$. Again, we denote by $L(\mathbf{d})$ the set of trees satisfying $\mathbf{d}$.

As with DTDs, we parametrize the definition of EDTDs by a class of representations $\mathcal{M}$ of the internal regular string languages. For example, EDTD(DFA) is the class of EDTDs $(\Sigma, \Sigma', d, \mu)$ in which d is a DTD(DFA).

For ease of exposition, we always take $\Sigma' = \{a^i \mid 1 \leq i \leq k_a, a \in \Sigma, i \in \mathbb{N}\}$ for some natural numbers k_a, and we set $\mu(a^i) = a$. If a node is labeled by

[1] Papakonstantinou and Vianu used the term *specialized DTD* as types *specialize* Σ-symbols. We prefer the term *extended DTD* as it expresses more clearly that the power of the schemas is amplified.

some $a^i \in \Sigma'$, we call a^i the *type* of the node. For simplicity, we also denote EDTDs in examples in a similar way as DTDs, that is, we write $a^i \to r$ rather than $d(a^i) = L(r)$.

Example 3.4 A simple example of an EDTD(RE) is the following:

$$\begin{aligned} \text{store}^1 &\to (\text{dvd}^1 + \text{dvd}^2)^* \text{dvd}^2 (\text{dvd}^1 + \text{dvd}^2)^* \\ \text{dvd}^1 &\to \text{title}^1\ \text{price}^1\ \text{summary}^1 \\ \text{dvd}^2 &\to \text{title}^1\ \text{price}^1\ \text{discount}^1\ \text{summary}^1 \end{aligned}$$

Here, dvd^1 defines ordinary DVDs, while dvd^2 defines DVDs on sale. The rule for store^1 specifies that there should be at least one DVD on sale.

We consider two restrictions on EDTDs in the remainder of this section, the first of which is the *single-type* EDTD [19]. Single-type EDTDs are an interesting subclass of EDTDs since they correspond to the expressive power of XML Schema [23], which is a widely used XML schema language in practice.

Definition 3.5 ([19]) A *single-type EDTD* (EDTD^{st}) is an EDTD $(\Sigma, \Sigma', d, \mu)$ with the property that for every $a \in \Sigma'$, in the regular expression $d(a)$ no two types b^i and b^j with $i \neq j$ occur.

The EDTD in Example 3.4 is not single-type as both dvd^1 and dvd^2 occur in the rule for store^1.

Example 3.6 A simple example of a single-type EDTD(RE) is the following:

$$\begin{aligned} \text{store}^1 &\to \text{regulars}^1\ \text{discounts}^1 \\ \text{regulars}^1 &\to (\text{dvd}^1)^* \\ \text{discounts}^1 &\to \text{dvd}^2\ (\text{dvd}^2)^* \\ \text{dvd}^1 &\to \text{title}^1\ \text{price}^1\ \text{summary}^1 \\ \text{dvd}^2 &\to \text{title}^1\ \text{price}^1\ \text{discount}^1\ \text{summary}^1 \end{aligned}$$

Although there are still two element definitions dvd^1 and dvd^2, they can only occur in different right hand sides.

The second restricted EDTD that we consider is called the *restrained competition* EDTD. It was originally defined because it is more expressive than a single-type EDTD, while still admitting a fast and simple top-down validation algorithm [19]. Interestingly, it has been shown recently that this restriction corresponds precisely to the semantic notion of *one-pass preorder typeable EDTDs*: it defines the largest class of EDTDs for which every node can be assigned the correct type at the first time it is met in a depth-first left-to-right traversal of the tree [14]. We elaborate on this in Section 3.2.

Definition 3.7 ([19]) Let $\mathbf{d} = (\Sigma, \Sigma', d, \mu)$ be an EDTD. A regular expression r over Σ' *restrains competition* if there are no strings wa^iv and wa^jv' in $L(r)$

with $i \neq j$. We call $\mathbf{d}$ a *restrained competition* EDTD (EDTD$^{\text{rc}}$) if all regular expressions occurring in the definition of d restrain competition.

Example 3.8 An example of a restrained competition EDTD(RE) that is not single-type is given next:

$$\begin{aligned} \text{store}^1 &\to (\text{dvd}^1)^* \ \text{discounts}^1 \ (\text{dvd}^2)^* \\ \text{dvd}^1 &\to \text{title}^1 \ \text{price}^1 \ \text{summary}^1 \\ \text{dvd}^2 &\to \text{title}^1 \ \text{price}^1 \ \text{discount}^1 \ \text{summary}^1 \end{aligned}$$

Note that every single-type EDTD is also restrained competition. The classes of tree languages defined by the grammars introduced above are included as follows: DTD $\subsetneq$ EDTD$^{\text{st}}$ $\subsetneq$ EDTD$^{\text{rc}}$ $\subsetneq$ EDTD [19]. These strict inclusions can be understood by investigating Examples 3.4–3.8. Using the characterizations in Section 3.2, it is easy to see that every one of these examples defines a tree language that is not expressible in the weaker formalisms.

3.2 Alternative Characterizations of the Schema Languages

The expressive power of the above defined schema languages can be characterized in several manners: types of EDTDs that functionally depend on a part of the tree, DTDs that make use of contextual patterns, and closure properties of tree languages under under subtree exchange [14]. These alternative characterizations make use of the *ancestor-string* and the *ancestor-left-sibling string* of a node, which we define next.

Let t be a tree and v be a node. By ch-str$^t(v)$ we denote the string formed by the children of v, i.e., $\text{lab}^t(v1)\cdots\text{lab}^t(vn)$ if v has n children. By anc-str$^t(v)$ we denote the string formed by the labels on the path from the root to v, i.e., $\text{lab}^t(\varepsilon)\text{lab}^t(i_1)\text{lab}^t(i_1i_2)\cdots\text{lab}^t(i_1i_2\cdots i_k)$ where $v = i_1i_2\cdots i_k$. By l-sib-str(v) we denote the string formed by the labels of the left siblings of v, i.e., $\text{lab}^t(u1)\cdots\text{lab}^t(uk)$ where $v = uk$. By anc-sib-str(v) we denote the string

$$\text{l-sib-str}^t(\varepsilon)\#\text{l-sib-str}^t(i_1)\#\cdots\#\text{l-sib-str}^t(i_1i_2\cdots i_k)$$

formed by concatenating the left-sibling strings of all ancestors starting from the root. Here, we assume that "#" is a special symbol not occurring in Σ.

For a tree t and a node v we denote by preceding$^t(v)$ the tree resulting from t by removing everything below v, all right siblings of v's ancestors and of v, and their respective subtrees. We illustrate the just defined notions in Figure 2.

Definition 3.9 We say that an EDTD $\mathbf{d} = (\Sigma, \Sigma', d, \mu)$ *has ancestor-based types* if there is a (partial) function $f : (\Sigma \cup \{\#\})^* \to \Sigma'$ such that, for each tree $t \in L(\mathbf{d})$ the following holds:

(1) there is a unique tree $t' \in L(d)$ with $\mu(t') = t$; and

(2) for each node $v \in \text{Dom}(t)$, the label of v in t' is $f(\text{anc-str}^t(v))$.

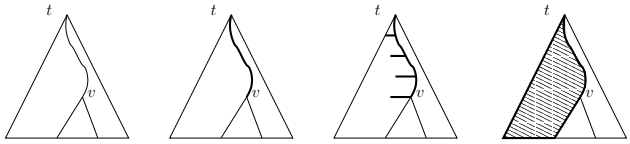


Fig. 2. From left to right: a tree t, the ancestor-string of v in t, the ancestor-sibling-string of v in t and the preceding of v in t.

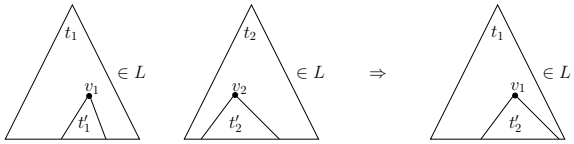


Fig. 3. Label-guarded subtree exchange.

We say that **d** *has ancestor-sibling based types* (respectively, *label-based types*) if the same holds with anc-str$^t(v)$ replaced by anc-sib-str$^t(v)$ (respectively, by lab$^t(v)$). We say that **d** *has preceding-based types* if there is a (partial) function $f : \mathcal{T}_\Sigma \to \Sigma'$ such that the above statement holds with preceding$^t(v)$ in place of anc-str$^t(v)$.

Definition 3.10 An *ancestor-guarded DTD* **d** is a pair (d, s_d) where $s_d \in \Sigma$ is the start symbol as in a DTD. But in contrast to a DTD, d is a finite set of triples (r, a, s), where $a \in \Sigma$ and r and s are regular expressions. A tree t satisfies **d** if for every node $v \in \mathrm{Dom}(t)$ the following holds. If lab$^t(v) = a$ then there must be a triple (r, a, s) in d such that anc-str$^t(v)$ matches r and ch-str$^t(v)$ matches s.

An *ancestor-sibling-guarded DTD* is defined in the same way with the difference that r has to be matched by anc-sib-str(v).

Note that it does not make much sense to define a *label-guarded DTD*, as this would simply be a DTD.

In the following definition, we denote by $t_1[u \leftarrow t_2]$ the tree obtained from a tree t_1 by replacing the subtree rooted at $u \in \mathrm{Dom}(t_1)$ by t_2. By subtree$^t(u)$ we denote the subtree of t rooted at u.

Definition 3.11 We say that a tree language L is *closed under ancestor-guarded subtree exchange* if the following holds. Whenever for two trees $t_1, t_2 \in T$ with nodes $u_1 \in \mathrm{Dom}(t_1)$ and $u_2 \in \mathrm{Dom}(t_2)$ it holds that anc-str$^{t_1}(u_1) =$ anc-str$^{t_2}(u_2)$, this implies that $t_1[u_1 \leftarrow \text{subtree}^{t_2}(u_2)] \in T$. We call it *closed under ancestor-sibling-guarded subtree exchange* (respectively, *label-guarded subtree exchange*) if the same holds with anc-sib-str$^{t_1}(u_1) =$ anc-sib-str$^{t_2}(u_2)$ (respectively, lab$^{t_1}(u_1) =$ lab$^{t_2}(u_2)$) as precondition of the implication.

These subtree exchange properties are illustrated in Figures 3, 4, and 5.

The following theorems characterize the expressive power of DTDs, single-type EDTDs, and restrained competition EDTDs. We assume in both theorems that L is a tree language in which every tree has the same root label.

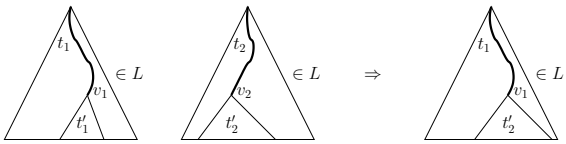


Fig. 4. Ancestor-guarded subtree exchange.

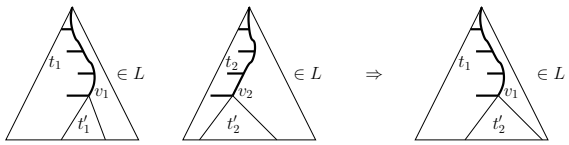


Fig. 5. Ancestor-sibling-guarded subtree exchange.

Theorem 3.12 ([14]) *For a regular tree language L the following are equivalent:*

(a) L is definable by a single-type EDTD;

(b) L is definable by an EDTD with ancestor-based types;

(c) L is closed under ancestor-guarded subtree exchange; and,

(d) L is definable by an ancestor-guarded DTD.

A corresponding theorem also holds for DTDs, by replacing *ancestor* by *label*. The equivalence between (c) and (a) is then already obtained in [20] (and the other equivalences are straightforward).

For restrained competition EDTDs, a similar theorem holds, but it has one extra characterization:

Theorem 3.13 ([14]) *For a regular tree language L the following are equivalent:*

(a) L is definable by a restrained competition EDTD;

(b) L is definable by an EDTD with ancestor-sibling-based types;

(c) L is definable by an EDTD with preceding-based types;

(d) L is closed under ancestor-sibling-guarded subtree exchange; and

(e) L is definable by an ancestor-sibling-guarded DTD.

The extra characterization (c) is quite interesting indeed. It states that an EDTD with ancestor-sibling-based types does not become more expressive when we allow its types to depend on the *entire preceding* of a node. The importance of this characterization becomes apparent in the context of validating XML documents as SAX-streams. Intuitively, the SAX-stream of a tree is obtained from a tree by traversing it in a depth-first left-to-right manner and writing the opening tag `<a>` for every a-labeled node that is met for the first time (when coming from the node's parent) and the closing tag `</a>` for every a-labeled node that is met for the second time in the traversal (when coming from the node's children). Indeed, Theorem 3.13 shows that the class

of restrained competition EDTDs is precisely the class of EDTDs that allow to unambiguously assign a type to a node when its opening tag is met in the SAX-stream (or, in other words, when the node is visited for the first time in the depth-first left-to-right traversal of the tree).

3.3 Closure Properties

It is well-known that unranked regular tree languages (and, hence, EDTDs) are closed under union, difference, complement, and intersection. But does this also hold for DTDs, single-type EDTDs, and restrained competition EDTDs? Most of these questions are investigated by Murata et al. [19,18]. We summarize their results, along with closure properties under complement, in Table 1. Here, we define the complement of a DTD (d, s) to be the language $\{t \in \mathcal{T}_\Sigma \mid \text{lab}^t(\varepsilon) = s\} - L(d)$ to avoid problems with the start symbol. We define the complement of languages defined by EDTD$^{\text{st}}$s and EDTD$^{\text{rc}}$s analogously.

	union	difference	complement	intersection
DTD	not closed	not closed	not closed	closed
EDTD$^{\text{st}}$	not closed	not closed	not closed	closed
EDTD$^{\text{rc}}$	not closed	not closed	not closed	closed
EDTD	closed	closed	closed	closed

Table 1
Closure properties of DTDs, single-type EDTDs, restrained competition EDTDs, and EDTDs.

We give some insights about why these results hold. We start by showing that DTDs, EDTD$^{\text{st}}$s, and EDTD$^{\text{rc}}$s are not closed under complement, which is quite easy using the characterizations of Section 3.2:

Example 3.14 Consider the DTD (d, a) over the alphabet $\Sigma = \{a, b\}$ defined as

$$a \to b^*$$
$$b \to b^*.$$

This DTD defines the set of trees where the symbol a is only allowed to occur at the root. We show that the complement of this DTD is not definable by a EDTD$^{\text{rc}}$, which shows that DTDs, EDTD$^{\text{st}}$s, and EDTD$^{\text{rc}}$s are not closed under complement. Indeed, consider the two trees in Figure 6(a), which are in the complement of the DTD d. By applying ancestor-sibling-guarded subtree exchange on the circled nodes, we obtain the tree $a(b(b)b(b))$, which is not in the complement of d.

As the tree language $\{t \in \mathcal{T}_\Sigma \mid \text{lab}^t(\varepsilon) = a\}$ is definable by a DTD, this example shows that DTDs, EDTD$^{\text{st}}$s, and EDTD$^{\text{rc}}$s are also not closed under difference.

Fig. 6. Counterexample trees for Example 3.14 and Example 3.15.

We can use a similar argument to show that DTDs, EDTD$^{\text{st}}$s, and EDTD$^{\text{rc}}$s are not closed under union:

Example 3.15 Consider the DTD (d_1, a) over alphabet $\{a, b, c\}$ defined as

$$a \to b^*c^*$$

and the DTD (d_2, a) over alphabet $\{a, b, c\}$ defined as

$$a \to b^*$$
$$b \to c.$$

In the union of these DTDs, either every b-child of the root has a c-child, or, none of the b-children of the root has a c-child. We show that this union is not definable by a EDTD$^{\text{rc}}$. Indeed, consider the two trees in Figure 6(b). Note that the left tree is in $L(d_1)$ and the right tree is in $L(d_2)$. However, by applying ancestor-sibling-guarded subtree exchange on the circled nodes, we obtain the tree $a(bb(c))$, which is not in the union of d_1 and d_2.

Finally, the closure of DTDs, EDTD$^{\text{st}}$s, and EDTD$^{\text{rc}}$s under intersection can easily be shown by direct construction, using a straightforward product operation on the characterizations of Theorems 3.12(d) and 3.13(e). An alternative proof can be found in the appendix of [19].

4 Formal Models for XML to XML Transformations

The second main ingredient of the typechecking problem are the tree transformations. In this section, we revisit several formalisms that have been used to model XML to XML transformations.

4.1 Simple Tree Transducers

We first consider a class of tree transducers which was defined in order to investigate settings in which the typechecking problem becomes tractable [10,11]. These tree transducers are not very expressive (in fact, they are the least expressive transducers in this paper), but they are also not meant to model a full-fledged query language. Instead, they are intended to model simple document restructurings that occur often in practice.

For a set Q, denote by $\mathcal{T}_\Sigma(Q)$ the set of Σ-trees where leaf nodes are labeled with elements from $\Sigma \cup Q$ instead of only Σ.

Definition 4.1 A *simple tree transducer* is a 4-tuple $T = (Q, \Sigma, q^0, R)$, where Q is a finite set of states, Σ is the input and output alphabet, $q^0 \in Q$ is the initial state, and R is a finite set of rules of the form $(q, a) \to t$, where $a \in \Sigma$, $q \in Q$, and $t \in \mathcal{T}_\Sigma(Q)$. When $q = q^0$, t is restricted to be either empty, or to have a Σ-symbol as its root label. Simple tree transducers are required to be deterministic: for every pair (q, a), there is at most one rule in R.

The translation defined by $T = (Q, \Sigma, q^0, R)$ on a tree t in state q, denoted by $T^q[t]$, is inductively defined as follows: if $t = \varepsilon$ then $T^q[t] := \varepsilon$; if $t = a(t_1 \cdots t_n)$ and there is a rule $(q, a) \to t' \in R$ then $T^q[t]$ is obtained from t' by replacing every node u in t' labeled with state p by the sequence of trees $T^p[t_1] \cdots T^p[t_n]$. Note that such nodes u can only occur at leaves. So, t' is only extended downwards. If there is no rule $(q, a) \to t' \in R$ then $T^q[t] := \varepsilon$. Finally, the transformation of t by T, denoted by $T(t)$, is defined as $T^{q^0}[t]$.

For simplicity, we have only used *trees* on the right hand sides of rewrite rules in simple tree transducers. Usually, their right hand sides contain *hedges*, which are essentially sequences of trees. For more details, we refer to [10,11].

Example 4.2 We describe a simple tree transducer T that returns, for a "store"-document conforming to the DTD in Example 3.2, the "store"- document in which first the DVD titles occur together with their prices, and further in the document, every DVD title occurs with its summary. Here, we abbreviate "title" by "t", "price" by "p", "discount" by "d", and "summary" by "s".

Let $T = (Q, \Sigma, q_0, R)$ where $Q = \{q_0, q_{\text{tpd}}, q_{\text{ts}}\}$, $\Sigma = \{\text{store}, \text{dvd}, \text{t}, \text{p}, \text{d}, \text{s}\}$, and R contains the rules

$$(q_0, \text{store}) \to \begin{array}{c} \text{store} \\ \diagup \quad \diagdown \\ q_{\text{tpd}} \quad q_{\text{ts}} \end{array}$$

$$\begin{array}{lll} (q_{\text{tpd}}, \text{dvd}) & \to & q_{\text{tp}} \\ (q_{\text{tpd}}, \text{t}) & \to & \text{t} \\ (q_{\text{tpd}}, \text{p}) & \to & \text{p} \\ (q_{\text{tpd}}, \text{d}) & \to & \text{d} \end{array} \qquad \begin{array}{lll} (q_{\text{ts}}, \text{dvd}) & \to & q_{\text{ts}} \\ (q_{\text{ts}}, \text{t}) & \to & \text{t} \\ (q_{\text{ts}}, \text{s}) & \to & \text{s} \end{array}$$

Intuitively, q_{tpd} and q_{tp} select the "t", "p" and "d" descendants of the "dvd" node, or the "t" and "s" descendants of the "dvd"-node, respectively.

Example 4.3 In Figure 7 we give the translation of the tree t from Figure 1(c) by the transducer of Example 4.2. For brevity, we again abbreviate "title", "price", "discount", and "summary" by their initial letters, respectively. In order to keep the example simple, we abbreviated sequences of the form $T^q[a_1] \cdots T^q[a_n]$ by $T^q[a_1 \cdots a_n]$.

Simple tree transducers can easily be extended with XPath expressions [11].

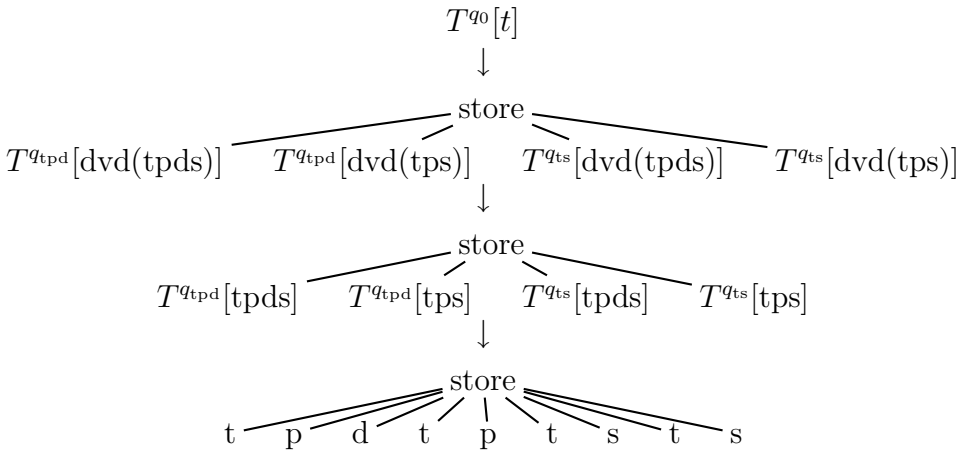


Fig. 7. The translation of the tree in Figure 1(c) by the transducer T of Example 4.2.

In that case, right hand sides of rules in the tree transducer can contain pairs of the form $\langle q, X\rangle$, where X is an XPath expression. The semantics of such a pair is that the transducer continues computation in state q in *all the nodes selected by* X (where the current node is the context node), rather than in *all children of the current node*. For more details, we refer to [11].

4.2 Macro Tree Transducers

We now turn to a much more powerful model of tree transducer for which the typechecking problem has been investigated: the macro tree transducer [5]. The reason why we treat macro tree transducers in this paper, is that they are not only a powerful transformation formalism, but they have also proven to be a useful tool to obtain upper bounds on the complexity of typechecking (cfr. Section 5.1.4).

As opposed to the XML schema languages from Section 3 and the simple tree transducers from Section 4.1, macro tree transducer are defined on *ranked* trees instead of unranked trees. However, it is well-known that unranked trees can be encoded to ranked trees in various ways, which provides a way to define the semantics of macro tree transducers over unranked trees. We will illustrate one such encoding later, when we explain the operation of a macro tree transducer.

Before we define macro tree transducers, we need to introduce some notions on ranked trees. A *ranked alphabet* (or ranked set) is a pair $(\Sigma, \text{rank}_\Sigma)$, where $\text{rank}_\Sigma : \Sigma \to \mathbb{N}$ is a function that maps each symbol a to the number of children that an a-labeled node is allowed to have in a ranked Σ-tree. For $k \geq 0$, we denote the set the set $\{a \in \Sigma \mid \text{rank}_\Sigma(a) = k\}$ by $\Sigma^{(k)}$. We also write $a^{(k)}$ to indicate that that $\text{rank}_\Sigma(a) = k$. For a set S, $\langle \Sigma, S\rangle$ is the ranked set $\Sigma \times S$ with $\text{rank}_{\langle\Sigma,S\rangle}(\langle a, s\rangle) = \text{rank}_\Sigma(a)$ for every $\langle a, s\rangle \in \langle \Sigma, S\rangle$.

Formally, the set $\mathcal{T}^R_\Sigma$ of *ranked* Σ*-trees* is the set of Σ-trees $a(t_1 \cdots t_k)$, where $a \in \Sigma^{(k)}$ and each t_i is a ranked Σ-tree.

We fix a *set of input variables* to be $X = \{x_1, x_2, \ldots\}$ and a *set of output*

variables to be $Y = \{y_1, y_2, \ldots\}$. For $k \geq 0$, $X_k = \{x_1, \ldots, x_k\}$ and $Y_k = \{y_1, \ldots, y_k\}$. We require that $\Sigma \cap X = \Sigma \cap Y = \emptyset$.

Definition 4.4 [[5]] A (non-deterministic) *macro tree transducer (MTT)* is a 5-tuple $M = (Q, \Sigma, \Delta, q_0, R)$, where Q is a ranked alphabet of states, Σ and Δ are the ranked alphabets of input and output symbols, respectively, $q_0 \in Q^{(0)}$ is the initial state, and R is a finite set of rules of the form

$$\langle q, a(x_1, \ldots, x_k)\rangle(y_1, \ldots, y_m) \to \zeta \qquad (*)$$

for $q \in Q^{(m)}$ and $a \in \Sigma^{(k)}$ with $m \geq 0, k \geq 0$. Here, $\zeta \in \mathcal{T}^R_{\langle Q, X_k\rangle \cup \Sigma}(Y_m)$. [2]

A macro tree transducer is called *deterministic* if, for every $q \in Q^{(m)}$ and $a \in \Sigma^{(k)}$, at most one rule of the form $(*)$ exists.

The rules of M can be viewed as term rewriting rules in the obvious way, with the input variables x_i ranging over $\mathcal{T}^R_\Sigma$ and the parameters y_j ranging over $\mathcal{T}^R_\Delta$. Then M induces a derivation relation $\Rightarrow_M$ on $\mathcal{T}^R_{\langle Q, \mathcal{T}^R_\Sigma\rangle \cup \Delta}$ and an input tree $s \in \mathcal{T}^R_\Sigma$ is translated by M into a set of possible output trees $t \in \mathcal{T}^R_\Delta$ with $\langle q_0, s\rangle \Rightarrow^*_M t$.

In the Example 4.5, we illustrate the derivation relation induced by a macro tree transducer in detail.

In order to operate a macro tree transducer over unranked trees, we consider unranked trees as *binary trees over the first-child and next-sibling relation*. That is, the left child of a node in the binary tree is the first child of the node in the unranked tree, and the second child of a node is the next sibling of the node in the unranked tree. We insert the special symbol $\perp$ when no first child or next sibling exists. For instance, the tree of Figure 8(a) then becomes the tree in Figure 8(b).

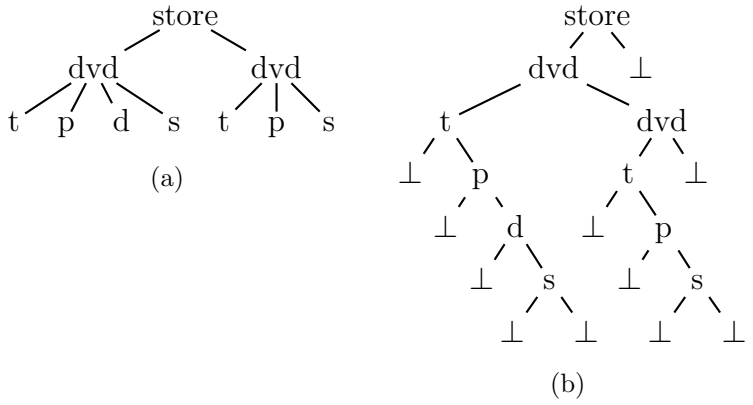


Fig. 8. An unranked tree (Figure 8(a)) as a binary tree over the first-child and next-sibling relation (Figure 8(b)).

[2] Recall that for a set S, we denote by $\mathcal{T}_\Sigma(S)$ the set of Σ-trees where leaf nodes are labeled with elements from $\Sigma \cup S$. Hence, $\mathcal{T}^R_{\langle Q, X_k\rangle \cup \Sigma}(Y_m)$ is the set $\mathcal{T}^R_{\langle Q, X_k\rangle \cup \Sigma \cup Y_m}$ where for every $y \in Y_m$, $\mathrm{rank}_Y(y) = 0$.

Example 4.5 We give an example of a MTT that removes all "dvd" nodes (and their subtrees) that are not on discount from documents satisfying the DTD from Example 3.2. Again, we abbreviate "title", "price", "discount", and "summary" by their initial letters, respectively.

The rules of this MTT are depicted in Figure 9. Intuitively, q_{id} is a state that performs the identity transformation, and $q_{\text{search-d}}$ is a state that searches the subtree for a "d"-labeled node and prunes its left subtree in the output unless the "d"-labeled node is found.

When executed on the tree t in Figure 8(b), we get the derivation in Figure 10. In this derivation, we denoted by t/u the subtree of t rooted at u.

$$\langle q_0, \text{store}(x_1, x_2)\rangle \to \text{store}\big(\langle q_{\text{sel-d}}, x_1\rangle, \bot\big)$$

$$\langle q_{\text{sel-d}}, \text{dvd}(x_1, x_2)\rangle \to \langle q_{\text{search-d}}, x_1\rangle\big(\langle q_{\text{id}}, x_1\rangle, \langle q_{\text{sel-d}}, x_2\rangle\big)$$

$$\langle q_{\text{search-d}}, \text{a}(x_1, x_2)\rangle(y_1, y_2) \to \langle q_{\text{search-d}}, x_2\rangle(y_1, y_2) \quad \text{for } a = \text{t,p}$$

$$\langle q_{\text{search-d}}, \text{d}(x_1, x_2)\rangle(y_1, y_2) \to \text{dvd}(y_1, y_2)$$

$$\langle q_{\text{search-d}}, \text{s}(x_1, x_2)\rangle(y_1, y_2) \to y_2$$

$$\langle q_{\text{id}}, a(x_1, x_2)\rangle \to a\big(\langle q_{\text{id}}, x_1\rangle, \langle q_{\text{id}}, x_2\rangle\big) \quad \text{for } a = \text{t,p,d,s}$$

$$\langle q_{\text{sel-d}}, \bot\rangle \to \bot$$

$$\langle q_{\text{id}}, \bot\rangle \to \bot$$

Fig. 9. Macro tree transducer from Example 4.5.

4.3 Further Tree Transducer Formalisms

4.3.1 k-Pebble Tree Transducers

The research on typechecking for XML transformations was initiated by Milo, Suciu, and Vianu [16]. As a formal model for tree transformations, they defined *k-pebble tree transducers*, which they designed to model many existing XML query languages (without data-value joins), and for which the typechecking problem w.r.t. extended DTDs is decidable.

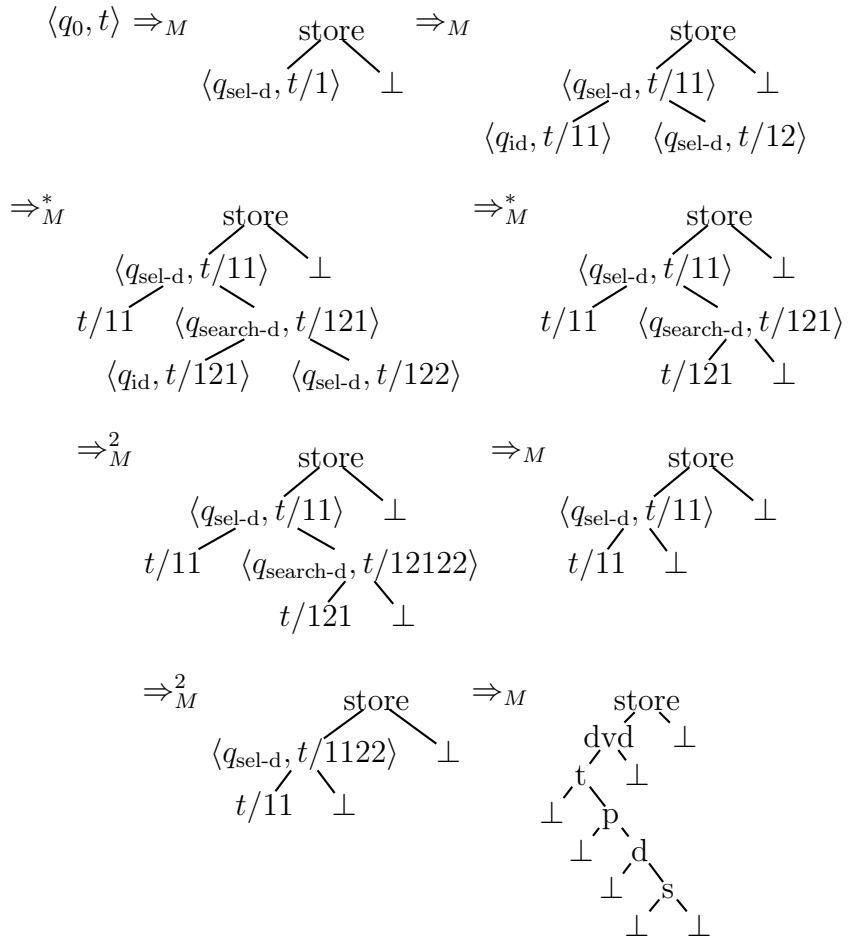


Fig. 10. Derivation of the execution of the MTT in Figure 9 on the tree t in Figure 8(b).

Intuitively, a k-pebble tree transducer is a finite state transducer which uses up to k heads (i.e., *pebbles*) that can be placed on the input. However, the transducer must place and remove the pebbles from the input in a stack-based fashion, and it can only move the highest ranked pebble that is placed on the input. In order to produce output, k-pebble tree transducers are equipped with *output transitions*. Such an output transition writes a symbol to the output, and creates a fresh copy of the k-pebble tree transducer for every child of the current node. These copies then run further in parallel, do not communicate to each other, and inherit the positions of all pebbles on the input.

Suciu presented the functional language *RecQL*, which is meant to characterize precisely the transformations that are expressible by k-pebble tree transducers [25].

Engelfriet and Maneth investigated the connection between macro tree transducers and k-pebble tree transducers [4]. They obtained that the transformation of a k-pebble tree transducer can be carried out by a composition of $k+1$ macro tree transducers. Conversely, they also showed that a macro tree

transducer can be simulated by a composition of four 1-pebble tree transducers (that is, a pebble tree transducer with only one head).

4.3.2 TL-*Transformers*

Recently, Maneth et al. defined TL-transformers, which are a much more powerful version of the simple tree transducers presented in Section 4.1. For instance, TL-transformers are not deterministic, and they can use unary or binary MSO formulas as *match* and *select* patterns, respectively. That is, their programs consist of rules of the form $f(m) \Rightarrow A$, where f is a function name, m is a unary formula that is determine to which nodes the function f is applicable (the *match* pattern), and A is the *action* associated to f. Here, the action A can contain binary MSO formulas to select the nodes in which the program can continue (*select* patterns).

A typechecking algorithm for TL transformations is obtained by decomposing TL transformations as a composition of macro tree transducers (See also Section 5.1.4).

4.3.3 Typechecking with Data Values

Sound and complete typechecking for transformations that have the power to test equality between data values has been investigated by Alon et al. [1]. They defined the query language QL, which defines queries that map trees *with* data values onto trees *without* data values. Queries in QL are tree templates in which each node is labeled with a formula, and a Σ-symbol or a variable. The formulas are existentially quantified conjunctions over *path expressions* and *comparison formulas*. Path expressions are formulas of the form XRY, where X and Y are variables (ranging over nodes in the input tree), and R is a regular expression. Nodes X and Y satisfy such a formula when there exists a path in the input tree from node X to node Y that matches the regular expression R. Comparison formulas are formulas of the form $X = V$ or $X \neq V$. Here, V can either be a variable ranging over nodes in the input tree, or a data value. The output of such a query on an input tree is defined in terms of possible matches of the free variables in the tree template on the input tree. The tree-structure of the output is inferred from the tree-structure of the query itself. For the precise details, we refer to [1].

Alon et al. characterize several fragments for which the typechecking problem for QL transformations is decidable, but they also show that the problem turns undecidable even in very restricted cases [1].

5 Methods for the Typechecking Problem

5.1 Methods for Proving Upper Bounds

In the past, a variety of methods for proving upper and lower bound for the typechecking problem have been devised. In the present section, we survey

some of these methods. With the word "schema", we refer to any of the schema languages presented in Section 3.

5.1.1 Emptiness

The emptiness test of tree automata or EDTDs seems to be a basic building stone of complete typechecking algorithms. Both the type inference and inverse type inference methods discussed in Sections 5.1.2 and 5.1.3, respectively, usually make use of the emptiness test of the schema languages at hand.

Formally, the emptiness test of a schema S is the following:

Emptiness: Given a schema S, is $L(S) = \emptyset$?

A direct reduction to the emptiness test of the schema language in the typechecking problem has been used in [10,11], to show tractability of several fragments of the typechecking problem. Here, it was used that testing emptiness of an EDTD in which the regular languages are represented by NFAs, is PTIME-complete [10].

5.1.2 Type Inference

Type inference might intuitively be the most straightforward technique to obtain a typechecking algorithm. Intuitively, we want to characterize the set of documents that can result from the transformation, when the input is conform to the input schema. Finally, we should test whether this set is contained in the output schema. This idea is used in several incomplete type checkers such as, for instance, in the programming language XDuce [6].

The type inference approach can be summarized more formally as follows:

Type Inference: Given input and output schemas S_{in} and S_{out} and tree transducer T, compute the tree language $L_{\text{out}} = T(L(S_{\text{in}}))$. Then verify whether $L_{\text{out}} \subseteq L(S_{\text{out}})$.

The last inclusion test usually involves computing an automaton A for the language $L_{\text{out}} \cap \overline{L(S_{\text{out}})}$, and testing whether $L(A) = \emptyset$, which is why this approach is usually more involved than the emptiness test. Here, $\overline{L(S_{\text{out}})}$ denotes the complement of $L(S_{\text{out}})$.

The main difficulty in this approach lies in precisely characterizing L_{out}. Indeed, even when S_{in} is a DTD and T is a simple tree transducer, L_{out} can be non-regular. For example, consider the DTD $s \to a^*$ defining S_{in} and the simple tree transducer T with the rules

$$\begin{aligned}(q_0, s) &\to s(q_1 q_2 q_3)\\ (q_1, a) &\to a\\ (q_2, a) &\to b\\ (q_3, a) &\to c.\end{aligned}$$

Then, the language $L_{\text{out}} = \{s(w) \mid w = a^n b^n c^n \text{ for } n \in \mathbb{N}\}$, which is clearly not regular. Note that it is even not possible to approximate L_{out}: there is no smallest regular tree language containing L_{out}. Indeed, for every regular tree

language L that contains L_{out} and for every tree $t \in L_{\text{out}} - L$, $L - \{t\}$ is a better regular approximation for L_{out} than L.

A sound and complete typechecking algorithm based on type inference is presented in [11], to show that arbitrary simple tree transducers can be typechecked w.r.t. DTD(RE$^+$)s in PTIME.[3] However, the main difficulty in this approach was showing the correctness of the algorithm, as it only infers an approximation of the language L_{out}.

5.1.3 Inverse Type Inference

Inverse type inference is inspired on type inference, but here we want to infer the set of input trees t for which $T(t)$ is in the ouput schema (or, *not* in the output schema, depending on the preferred variant).

More formally, inverse type inference can be described as follows:

Inverse Type Inference I: Given input and output schemas S_{in} and S_{out} and tree transducer T, first compute the pre-image $L_{\text{in}} = \{t \in \mathcal{T}_\Sigma \mid T(t) \in L(S_{\text{out}})\}$. Then, test whether $L(S_{\text{in}}) \subseteq L_{\text{in}}$.

Of course, one could also try another variant of inverse type inverence:

Inverse Type Inference II: Given input and output schemas S_{in} and S_{out} and tree transducer T, compute the complement $\overline{L(S_{\text{out}})}$ of $L(S_{\text{out}})$. Then, compute the pre-image of $\overline{L(S_{\text{out}})}$ through T, that is, $L_{\text{in}} = \{t \in \mathcal{T}_\Sigma \mid T(t) \in \overline{L(S_{\text{out}})}\}$. Then verify whether $L(S_{\text{in}}) \cap L_{\text{in}} = \emptyset$.

The preferred variant of the inverse typechecking method depends on the complexities of computing the complement of the output schema, and computing the pre-image of a tree language.

The inverse type inference technique was used for typechecking k-pebble tree transducers [16], macro tree transducers [4], and TL-programs [9]. Even though the transformation languages were quite powerful in all the three cases, it turned out that the inferred set L_{in} is regular when the output schema defines a regular language.

5.1.4 Compositions of Macro Tree Transducers or k-Pebble Tree Transducers

Of course, one can always reduce to instances of the typechecking problem which are known to be decidable, such as for macro tree transducers or for k-pebble tree transducers. Maneth et al., for example, reduced the typechecking problem for TL-transformations to the typechecking problem for a composition of macro tree transducers [9]. More specifically, they showed that a TL program can be rewritten as a composition of *(i)* three deterministic MTTs when the TL program is deterministic, or *(ii)* two MTTs and one *stay* MTT otherwise. Here, a stay MTT is a MTT which does not have to go to the children of the current node in a computation step, but is also allowed to stay at the current node.

[3] Here, RE$^+$ is a restricted fragment of regular expressions.

The time complexity for the inferred typechecking algorithm will be rather high with these approaches. In the case of macro tree transducers, it is hyperexponential in the number of compositions, and in the case of k-pebble tree transducers it is hyperexponential in the number of pebbles [4,16].

5.2 Methods for Proving Lower Bounds

5.2.1 Inclusion, Emptiness, Universality

The most obvious way to obtain lower bounds on the complexity of the typechecking problem is to use a (trivial) reduction from the inclusion problem of the schema language at hand. Indeed, the inclusion problem is simply an instance of the typechecking problem in which the tree transducer performs the identity transformation.

Inclusion: Given schemas S_{in} and S_{out}, is $L(S_{\text{in}}) \subseteq L(S_{\text{out}})$?

Note that, when considering settings of the typechecking problem where the input and/or output schema is fixed, this straightforward approach does not work anymore. When only the output schema is fixed, the emptiness test of the input schema is a lower bound: test whether $L(S_{\text{in}}) \subseteq \emptyset$. When only the input schema is fixed, the universality test for the output schema can be a lower bound: test whether $\mathcal{T}_\Sigma \subseteq L(S_{\text{out}})$. When considering a setting for typechecking in which both schemas are fixed, however, one of the simulation methods in Section 5.2.2 might be more interesting.

The following propositions give an overview on the complexity of the inclusion, emptiness and universality problems for the the schema languages from Section 3:

Proposition 5.1 *The inclusion problem is*

- PTIME-*complete for* ***D****(DFA)s [12,13];*
- PSPACE-*complete for* ***D****(NFA)s and* ***D****(RE)s [7,13,24]; and,*
- EXPTIME-*complete for EDTD(DFA)s, EDTD(NFA)s, and EDTD(RE)s [21],*

where ***D*** *stands for DTD, EDTDst, or EDTD^{rc}.*

Proposition 5.2 *The universality problem is*

- NLOGSPACE-*complete for* ***D****(DFA)s;*
- PSPACE-*complete for* ***D****(NFA)s and* ***D****(RE)s [7,13,15]; and,*
- EXPTIME-*complete for EDTD(DFA)s, EDTD(NFA)s, and EDTD(RE)s [21].*

where ***D*** *stands for DTD, EDTDst, or EDTD^{rc}.*

Proposition 5.3 *The emptiness problem is*

- PTIME-*hard for DTD(DFA) [12]; and,*
- *in* PTIME *for EDTD(NFA)s and EDTD(RE)s [10].*

As opposed to the rest of the paper, we do *not* assume here that DTDs do not contain useless symbols (otherwise, the emptiness problem would be trivial).

Maybe a note on the mentioned references is appropriate: PSPACE completeness of the universality and equivalence problems for regular expressions and NFAs is shown in [7,15,24]. For DFAs, these problems are in PTIME. In the full version of [13], it is shown that these upper bounds carry over to DTDs, EDTD^{st}s, and EDTD^{rc}s (lower bounds carry over trivially).

Proposition 5.2 states that the universality problem is NLOGSPACE-hard for DTD(DFA)s and in NLOGSPACE for EDTD^{rc}(DFA)s. The former is immediate, since graph reachability is NLOGSPACE-hard. The latter follows from the fact that universality for EDTD^{rc}(DFA)s can be decided using a reachability algorithm. Indeed, for a given EDTD^{rc} $\mathbf{d} = (\Sigma, \Sigma', d, \mu)$, we have to test whether the extended context free grammar d contains a symbol s that is *(i)* reachable from d's start symbol, and *(ii)* for which $\mu(d(s)) \neq \Sigma^*$. If this test succeeds, the EDTD^{rc} is non-universal. The result follows, as NLOGSPACE is closed under complement.

Seidl showed that the inclusion and equivalence problem are EXPTIME-complete for standard non-deterministic tree automata [21]. The EXPTIME upper bounds for EDTD(NFA)s and EDTD(RE)s can then be obtained through unranked-to-ranked encodings (which we also used in Section 4.2). Finally, the PTIME-hardness for emptiness of DTD(DFA)s is shown in [12], and PTIME membership for emptiness of EDTD(NFA)s and EDTD(RE)s follows from results in [10].

5.2.2 Simulations with the Tree Transducer

Perhaps the technically most interesting way to obtain lower bounds on the complexity of typechecking, is through simulation of a certain automata model (or, simulation of acceptance for classes of regular expressions) by the tree transducer.

Here, we focus on simulation of finite automata.[4] To this end, let $\mathcal{A}$ be a class of finite automata. The overall idea is to generate all possible inputs of $\mathcal{A}$ with the input schema. Then, we use the tree transducer to simulate one or more automata in $\mathcal{A}$ on the input tree and write to the output whether the input is accepted or not. The ouput schema can then verify whether some property holds for the simulated automata.

In this manner, we can test various decision problems of automata such as emptiness, inclusion, equivalence, and universality. For the inclusion problem of tree automata A_1 and A_2, for instance, we can do the following reduction. Our goal is to construct an instance of the typechecking problem that typechecks if and only if $L(A_1) \subseteq L(A_2)$. The input schema defines all possible

[4] Of course, the techniques we present can also be used to test e.g. satisfiability or validity of logical formulas.

input trees for A_1 or A_2. Given an input tree t, the tree transducer copies the t twice and simulates A_1 and A_2 in parallel on the left and right copy of t, respectively. At the end of this simulation, it can write to the output whether A_1, respectively A_2, accept t or not. Finally, the output schema can verify whether the tree transducer's output always encodes a situation in which A_1 had an accepting computation only if A_2 had an accepting computation. Note that, for this reduction, the tree transducer only has to be able to simulate the automata in $\mathcal{A}$, and copy a subtree of the input twice. Also, when the alphabets of the automata A_1 and A_2 are fixed, the constructed input and output schemas in the reduction do not depend on the given automata. Therefore, it is possible to obtain complexity hardness results using this reduction even in settings where *both the input and output schemas are fixed*.

When the tree transducer has the power to copy a part of its input tree an arbitrary number of times, the following problem can be interesting:

Intersection Emptiness: Given finite automata $A_1, \ldots, A_n$, is

$$L(A_1) \cap \cdots \cap L(A_n) = \emptyset?$$

The reduction in this case is analogous to the reduction that we just sketched. Typically, the tree transducer copies its input tree n times and simulates all the automata $A_1, \ldots, A_n$ in parallel. Finally, the output schema should verify whether the tree transducer's output always encodes a situation in which at least one simulated automaton rejects.

The following proposition gives the complexities of the intersection emptiness problem for various kinds of automata (or schema languages):

Proposition 5.4 *The intersection emptiness problem is*

- PSPACE-*complete for DFAs, NFAs, and REs [8];*
- EXPTIME-*complete for top-down deterministic (standard) tree automata [22];*
- PSPACE-*complete for DTD(DFA)s, DTD(NFA)s, and DTD(RE)s [13];*
- EXPTIME-*hard for EDTDst(DFA)s [13]; and*
- *in* EXPTIME *for EDTD(NFA)s and EDTD(RE)s.*

In the last item, membership in EXPTIME is immediate, as a product automaton for the intersection can be constructed in EXPTIME, and we can test emptiness of this product automaton in polynomial time in the size of the automaton [10].

This technique has been used many times in the literature. Milo, Suciu, and Vianu used a simulation of acceptance by *star-free generalized regular expressions* to obtain a non-elementary lower bound on the complexity of typechecking k-pebble tree transducers, even for fixed input and output schemas [16]. Martens, Neven, and Gyssens used a simulation of top-down deterministic standard tree automata by the simple tree transducers of Section 4.1 to obtain EXPTIME lowerbounds in several settings of the typechecking problem, also for fixed input and output schemas [12].

Acknowledgments

The author wishes to thank Frank Neven for his comments on a previous version of this paper.

References

[1] Alon, N., T. Milo, F. Neven, D. Suciu and V. Vianu, *XML with data values: typechecking revisited*, Journal of Computer and System Sciences **66** (2003), pp. 688–727.

[2] Bray, T., J. Paoli, C. Sperberg-McQueen, E. Maler and F. Yergeau, *Extensible Markup Language (XML)*, Technical report, World Wide Web Consortium (2004), http://www.w3.org/TR/REC-xml/.

[3] Brüggemann-Klein, A., M. Murata and D. Wood, *Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001*, Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology (2001).

[4] Engelfriet, J. and S. Maneth, *A comparison of pebble tree transducers with macro tree transducers*, Acta Informatica **39** (2003), pp. 613–698.

[5] Engelfriet, J. and H. Vogler, *Macro tree transducers*, Journal of Computer and System Sciences **31** (1985), pp. 71–146.

[6] Hosoya, H. and B. C. Pierce, *XDuce: A statically typed XML processing language*, ACM Transactions on Internet Technology **3** (2003), pp. 117–148.

[7] Hunt III, H. B., D. J. Rosenkrantz and T. G. Szymanski, *On the equivalence, containment, and covering problems for the regular and context-free languages*, Journal of Computer and System Sciences **12** (1976), pp. 222–268.

[8] Kozen, D., *Lower bounds for natural proof systems*, in: *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, 1977, pp. 254–266.

[9] Maneth, S., T. Perst, A. Berlea and H. Seidl, *XML type checking with macro tree transducers*, in: *Proc. 24th Symposium on Principles of Database Systems (PODS 2005)*, 2005, pp. 283–294.

[10] Martens, W. and F. Neven, *On the complexity of typechecking top-down XML transformations*, Theoretical Computer Science **336** (2005), pp. 153–180.

[11] Martens, W. and F. Neven, *Frontiers of tractability for typechecking simple XML transformations*, Journal of Computer and System Sciences (2006), to Appear.

[12] Martens, W., F. Neven and M. Gyssens, *On typechecking top-down XML tranformations: Fixed input or output schemas* (2005), submitted, available at http://alpha.uhasselt.be/wim.martens.

[13] Martens, W., F. Neven and T. Schwentick, *Complexity of decision problems for simple regular expressions*, in: *Proc. 29th Symposium on Mathematical Foundations of Computer Science (MFCS 2004)*, 2004, pp. 889–900.

[14] Martens, W., F. Neven and T. Schwentick, *Which XML schemas admit 1-pass preorder typing?*, in: *Proc. 10th International Conference on Database Theory (ICDT 2005)*, 2005, pp. 68–82.

[15] Meyer, A. R. and L. J. Stockmeyer, *The equivalence problem for regular expressions with squaring requires exponential space*, in: *Proc. 13th Annual Symposium on Foundations of Computer Science (FOCS 1972)*, 1972, pp. 125–129.

[16] Milo, T., D. Suciu and V. Vianu, *Typechecking for XML transformers*, Journal of Computer and System Sciences **66** (2003), pp. 66–97.

[17] Møller, A. and M. I. Schwartzbach, *The design space of type checkers for XML transformation languages*, in: *Proc. 10th International Conference on Database Theory (ICDT 2005)*, 2005, pp. 17–36.

[18] Murata, M., D. Lee and M. Mani, *Taxonomy of XML schema languages using formal language theory*, in: *Extreme Markup Languages*, Montreal, Canada, 2001.

[19] Murata, M., D. Lee, M. Mani and K. Kawaguchi, *Taxonomy of XML schema languages using formal language theory*, ACM Transactions on Internet Technology **5** (2005), to Appear, Full version of [18].

[20] Papakonstantinou, Y. and V. Vianu, *DTD inference for views of XML data*, in: *Proc. 19th ACM Symposium on Principles of Database Systems (PODS 2000)*, 2000, pp. 35–46.

[21] Seidl, H., *Deciding equivalence of finite tree automata*, SIAM Journal on Computing **19** (1990), pp. 424–437.

[22] Seidl, H., *Haskell overloading is DEXPTIME-complete*, Information Processing Letters **52** (1994), pp. 57–60.

[23] Sperberg-McQueen, C. and H. Thompson, *XML Schema* (2005), http://www.w3.org/XML/Schema.

[24] Stockmeyer, L. J. and A. R. Meyer, *Word problems requiring exponential time: Preliminary report*, in: *Proc. 5th Annual ACM Symposium on Theory of Computing (STOC 1973)*, 1973, pp. 1–9.

[25] Suciu, D., *Typechecking for semistructured data*, in: *Proc. 8th International Workshop on Database Programming Languages (DBPL 2001)*, 2001, pp. 1–20.