Typechecking Top-Down Uniform Unranked Tree Transducers

Wim Martens and Frank Neven

University of Limburg
E-mail: {wim.martens,frank.neven}@luc.ac.be

Abstract. We investigate the typechecking problem for XML queries: statically verifying that every answer to a query conforms to a given output schema, for inputs satisfying a given input schema. As typechecking quickly turns undecidable for query languages capable of testing equality of data values, we return to the limited framework where we abstract XML documents as labeled ordered trees. We focus on simple top-down recursive transformations motivated by XSLT and structural recursion on trees. We parameterize the problem by several restrictions on the transformations (deleting, non-deleting, bounded width) and consider both tree automata and DTDs as output schemas. The complexity of the typechecking problems in this scenario range from PTIME to EXPTIME.

1 Introduction

The emergence of XML as the likely standard for representing and exchange of data on the Web confirmed the central role of semistructered data. However, it has also marked the return of the schema. In the context of the Web, schemas can be used to validate data exchange. In a typical scenario, a user community agrees on a common schema and on producing only data conforming to that schema. This raises the issue of typechecking: verifying at compile time that every XML document which is the result of a specified query applied to a valid input document, satisfies the output schema [23, 24].

Obviously, typechecking depends on the transformation language and the schema language at hand. As shown by Alon et al. [1, 2], when transformation languages have the ability to compare data values, the typechecking problem quickly turns undecidable. Milo, Suciu, and Vianu argued that the capability of most XML transformation languages can be encompassed by k-pebble transducers when data values are ignored and XML documents can be abstracted by labeled ordered trees [15]. Further, the authors showed that the typechecking problem in this context is decidable. More precisely, given two types τ_1 and τ_2, represented by tree automata, and a k-pebble transducer T, it is decidable whether $T(t) \in \tau_2$ for all $t \in \tau_1$. Here, $T(t)$ is the tree obtained by running T on input t. The complexity, however, is very bad: non-elementary.

In an attempt to lower the complexity, we consider much simpler tree transformations: those defined by deterministic top-down uniform tree transducers

Table 1. The presented results: the top row of the table shows the representation of the input and output schemas and the left column shows the type of tree transducer.

	NTA	DTA	DTD(NFA)	DTD(DFA)	DTD($\mathcal{SL}$)
general	EXPTIME	EXPTIME	EXPTIME	EXPTIME	EXPTIME
non-deleting	EXPTIME	EXPTIME	PSPACE	PSPACE	CONP
bounded width	EXPTIME	in EXPTIME/PSPACE-hard	PSPACE	PTIME	CONP

on unranked trees. Such transformations correspond to structural recursion on trees [5] and to simple top-down XSLT transformations [3, 6]. The transducers are called uniform as they cannot distinguish between the order of siblings. In brief, a transformation consists of a top-down traversal of the input tree where every node is replaced by a new tree (possibly the empty tree).

We show that the ability of transducers to delete interior nodes (i.e., replacing them by the empty tree) already renders the typechecking problem EXPTIME-hard for very simple DTDs (e.g. DTDs with deterministic regular expressions on their right hand side). To obtain a lower complexity, for the remainder of the paper, we focus on non-deleting transformations. Our inquiries reveal that the complexity of the typechecking problem of non-deleting transducers is determined by two features: (1) non-determinism in the schema languages; and, (2) unbounded copying of subtrees by the transducer. Only when we disallow both features, we get a PTIME-complexity for the typechecking problem. An overview of our results is given in Table 1. Unless specified otherwise, all complexities are both upper and lower bounds. The top row of the table shows the representation of the input and output schemas and the left column shows the type of tree transducer. NTA and DTA stand for non-deterministic and deterministic tree automata, respectively. Such automata abstract the expressiveness of XML Schema [7]. DTD(X) stands for DTDs whose right-hand sides consist of regular languages in X. The exact definitions are given in section 2.

Related Work. A problem related to typechecking is type inference [14, 19]. This problem consists in constructing a tight output schema, given an input schema and a transformation. Of course, solving the type inference problem implies a solution for the typechecking problem: check containment of the inferred schema into the given one. However, characterizing output languages of transformations is quite hard [19].

The transducers considered in the present paper are restricted versions of the ones studied by Maneth and Neven [13]. They already obtained a non-elementary upper bound on the complexity of typechecking (due to the use of monadic second-order logic in the definition of the transducers).

Although the structure of XML documents can be faithfully represented by unranked trees (these are trees without a bound on the number of children of nodes), Milo, Suciu, and Vianu chose to study k-pebble transducers over binary trees as there is an immediate encoding of unranked trees into binary ones. The top-down variants of k-pebble transducers are well-studied on binary trees [11]. However, these results do not aid in the quest to characterize precisely the complexity of typechecking transformations on unranked trees. Indeed, the class of unranked tree transductions can *not* be captured by ordinary transducers working on the binary encodings. Macro tree transducers can simulate our transducers on the binary encodings [13, 9], but as very little is known about their complexity this observation is not of much help. For these reasons, we chose to work directly with unranked tree transducers.

Tozawa considered typechecking w.r.t. tree automata for a fragment of top-down XSLT similar to ours [25]. He adapts the backward type inference technique of [15] and obtains a double exponential time algorithm.

Due to space limitations, we only provide sketches of proofs.

2 Definitions

In this section we provide the necessary background on trees, automata, and uniform tree transducers.

2.1 Trees and Hedges

We fix a finite alphabet Σ. The set of unranked Σ-trees, denoted by $\mathcal{T}_\Sigma$, is the smallest set of strings over Σ and the parenthesis symbols ')' and '(' such that for $\sigma \in \Sigma$ and $w \in \mathcal{T}_\Sigma^*$, $\sigma(w)$ is in $\mathcal{T}_\Sigma$. We write σ rather than $\sigma()$. 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*. In the following, whenever we say tree, we always mean Σ-tree. A *hedge* is a finite sequence of trees. The set of hedges, denoted by $\mathcal{H}_\Sigma$, is defined as $\mathcal{T}_\Sigma^*$.

For every hedge $h \in \mathcal{H}_\Sigma$, the *set of nodes of* h, denoted by $\mathrm{Dom}(h)$, is the subset of $\mathbb{N}^*$ defined as follows:

- if $h = \varepsilon$, then $\mathrm{Dom}(h) = \emptyset$;
- if $h = t_1 \cdots t_n$ where each $t_i \in \mathcal{T}_\Sigma$, then $\mathrm{Dom}(h) = \bigcup_{i=1}^{n} \{iu \mid u \in \mathrm{Dom}(t_i)\}$; and,
- if $h = \sigma(w)$, then $\mathrm{Dom}(h) = \{\varepsilon\} \cup \mathrm{Dom}(w)$.

In the sequel we adopt the following convention: we use $t, t_1, t_2, \ldots$ to denote trees and $h, h_1, h_2, \ldots$ to denote hedges. Hence, when we write $h = t_1 \cdots t_n$ we tacitly assume that all t_i's are trees. For every $u \in \mathrm{Dom}(h)$, we denote by $\mathrm{lab}^h(u)$ the label of u in h. A *tree language* is a set of trees.

2.2 DTDs and Tree Automata

We use extended context-free grammars and tree automata to abstract from DTDs and the various proposals for XML schemas. Further, we parameterize the definition of DTDs by a class of representations $\mathcal{M}$ of regular string languages like, e.g., the class of DFAs or NFAs. For $M \in \mathcal{M}$, we denote by $L(M)$ the set of strings accepted by M.

Definition 1. Let $\mathcal{M}$ be a class of representations of regular string languages over Σ. A DTD is a tuple (d, s_d) where d is a function that maps Σ-symbols to elements of $\mathcal{M}$ and $s_d \in \Sigma$ is the start symbol. For simplicity, we usually denote (d, s_d) by d.

A tree t satisfies d if $\text{lab}^t(\varepsilon) = s_d$ and for every $u \in \text{Dom}(t)$ with n children $\text{lab}^t(u1) \cdots \text{lab}^t(un) \in L(d(\text{lab}^t(u)))$. By $L(d)$ we denote the tree language accepted by d.

We parameterize DTDs by the formalism used to represent the regular language $\mathcal{M}$. Therefore, we denote by DTD($\mathcal{M}$) the class of DTDs where the regular string languages are represented by elements of $\mathcal{M}$. The *size* of a DTD is the sum of the sizes of the elements of $\mathcal{M}$ used to represent the function d.

To define unordered languages we make use of the specification language $\mathcal{SL}$ inspired by [17] and also used in [1, 2]. The syntax of the language is as follows.

Definition 2. For every $\sigma \in \Sigma$ and natural number i, $\sigma^{=i}$ and $\sigma^{\geq i}$ are *atomic $\mathcal{SL}$-formulas*; true is also an atomic $\mathcal{SL}$-formula. Every atomic $\mathcal{SL}$-formula is an $\mathcal{SL}$-formula and the negation, conjunction, and disjunction of $\mathcal{SL}$-formulas are also $\mathcal{SL}$-formulas.

A string w over Σ satisfies an atomic formula $\sigma^{=i}$ if it has exactly i occurrences of σ; w satisfies $\sigma^{\geq i}$ if it has at least i occurrences of σ. Further, true is satisfied by every string.[1] Satisfaction of Boolean combinations of atomic formulas is defined in the obvious way. As an example, consider the $\mathcal{SL}$ formula $\text{co-producer}^{\geq 1} \rightarrow \text{producer}^{\geq 1}$. This expresses the constraint that a co-producer can only occur when a producer occurs. The *size* of an $\mathcal{SL}$-formula is the number of symbols that occur in it (every i in $\sigma^{=i}$ or $\sigma^{\geq i}$ is written in binary notation).

We recall the definition of non-deterministic tree automata from [4]. We refer the unfamiliar reader to [16] for a gentle introduction.

Definition 3. A *nondeterministic tree automaton (NTA)* is a tuple $B = (Q, \Sigma, \delta, F)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and δ is a function $Q \times \Sigma \rightarrow 2^{Q^*}$ such that $\delta(q, a)$ is a regular string language over Q for every $a \in \Sigma$ and $q \in Q$.

A *run* of B on a tree t is a labeling $\lambda : \text{Dom}(t) \rightarrow Q$ such that for every $v \in \text{Dom}(t)$ with n children, $\lambda(v1) \cdots \lambda(vn) \in \delta(\lambda(v), \text{lab}^t(v))$. Note that when v has no children, then the criterion reduces to $\varepsilon \in \delta(\lambda(v), \text{lab}^t(v))$. A run is *accepting* iff the root is labeled with an accepting state, that is, $\lambda(\varepsilon) \in F$. A tree is accepted if there is an accepting run. The set of all accepted trees is denoted by $L(B)$. We extend the definition of δ to trees and denote this by $\delta^*(t)$: if t consists of only one node labeled with a then $\delta^*(t) = \{q \mid \varepsilon \in \delta(q, a)\}$; if t is of the form $a(t_1 \cdots t_n)$, then $\delta^*(t) = \{q \mid \exists q_1 \in \delta^*(t_1), \ldots, \exists q_n \in \delta^*(t_n)$ and $q_1 \cdots q_n \in \delta(q, a)\}$. So, t is accepted if $\delta^*(t) \cap F \neq \emptyset$.

A tree automaton is *bottom-up deterministic* if for all $q, q' \in Q$ with $q \neq q'$ and $a \in \Sigma$, $\delta(q, a) \cap \delta(q', a) = \emptyset$. We denote the set of bottom-up deterministic

[1] The empty string is obtained by $\bigwedge_{\sigma \in \Sigma} \sigma^{=0}$ and the empty set by $\neg$ true.

NTAs by DTA. A tree automaton is *top-down deterministic* if for all $q, q' \in Q$ with $q \neq q'$, $a \in \Sigma$, and $n \geq 0$, $\delta(q,a)$ contains at most one string of length n.

Like for DTDs, we parameterize NTAs by the formalism used to represent the regular languages in the transition functions $\delta(q,a)$. So, for $\mathcal{M}$ a class of representations of regular languages, we denote by NTA($\mathcal{M}$) the class of NTAs where all transition functions are represented by elements of $\mathcal{M}$. The *size* of an automaton B is then $|Q| + |\Sigma| + \sum_{q,a} |\delta(q,a)| + |F|$. Here, by $|\delta(q,a)|$ we denote the size of the automaton accepting $\delta(q,a)$. Unless explicitly specified otherwise, $\delta(q,a)$ is always represented by an NFA.

Let 2AFA be the class of two-way alternating finite automata [12]. We give without proof the following theorem which will be a useful tool for obtaining upper bounds.

Theorem 1. *1. Emptiness of NTA(NFA) is in* PTIME*;*
2. Emptiness of NTA(2AFA) is in PSPACE.

2.3 Transducers

We next define the tree transducers used in this paper. To simplify notation, we restrict to one alphabet. That is, we consider transductions mapping Σ-trees to Σ-trees. It is, however, possible to define transductions where the input alphabet differs from the output alphabet.

For a set Q, denote by $\mathcal{H}_\Sigma(Q)$ (resp. $\mathcal{T}_\Sigma(Q)$) the set of Σ-hedges (resp. trees) where leaf nodes can be labeled with elements from Q.

Definition 4. A *uniform tree transducer* is a tuple (Q, Σ, 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 $(q,a) \to h$, where $a \in \Sigma$, $q \in Q$, and $h \in \mathcal{H}_\Sigma(Q)$. When $q = q_0$, h is restricted to $\mathcal{T}_\Sigma(Q) \setminus Q$.

The restriction on rules with the initial state ensures that the output is always a tree rather than a hedge. For the remainder of this paper, when we say tree transducer, we always mean *uniform* tree transducer.

Example 1. Let $T = (Q, \Sigma, p, R)$ where $Q = \{p, q\}$ and R contains the rules

$$\begin{array}{ll} (p,a) \to d(e) & (p,b) \to c(q\ p) \\ (q,a) \to c\ q & (q,b) \to d(q) \end{array} .$$

Our definition of tree transducers corresponds to structural recursion [5] and a fragment of top-down XSLT. For instance, the XSLT program equivalent to the above transducer is given in Figure 1 (we assume the program is started in mode p). □

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 $(q,a) \to h \in R$ then $T^q(t)$ is obtained from h by replacing

```
<xsl:template match="a" mode ="p">
  <d>
     <e/>
  </d>
</xsl:template>
```

```
<xsl:template match="b" mode ="p">
  <c>
      <xsl:apply-templates mode="q"/>
      <xsl:apply-templates mode="p"/>
  </c>
</xsl:template>
```

```
<xsl:template match="a" mode ="q">
  <c/>
  <xsl:apply-templates mode="q"/>
</xsl:template>
```

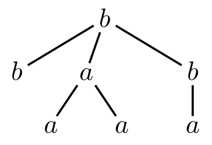
```
<xsl:template match="b" mode ="q">
  <d>
     <xsl:apply-templates mode="q"/>
  </d>
</xsl:template>
```

Fig. 1. The XSLT program equivalent to the transducer of Example 1.

every node u in h labeled with p by the hedge $T^p(t_1)\cdots T^p(t_n)$. Note that such nodes u can only occur at leaves. So, h is only extended downwards. If there is no rule $(q,a) \to h \in R$ then $T^q(t) := \varepsilon$. Finally, define the transformation of t by T, denoted by $T(t)$, as $T^{q_0}(t)$.

For $a \in \Sigma$, $q \in Q$ and $(q,a) \to h \in R$, we denote h by $\mathrm{rhs}(q,a)$. If q and a are not important, we say that h is a rhs. The *size* of T is $|Q|+|\Sigma|+\sum_{(q,a)} |\mathrm{rhs}(q,a)|$.

Example 2. In Figure 2 we give the translation of the tree t defined as



by the transducer of Example 1. □

We discuss two features which are of importance in the remainder of the paper: copying and deleting. The rule $(p,b) \to c(q\,p)$ in the above example copies the children of the current node in the input tree two times: one copy is processed in state q and the other in state p. The symbol c is the parent node of the two copies. So the current node in the input tree corresponds to the latter node. The rule $(q,a) \to c\,q$ also copies the children of the current node two times. However, in this case, one copy is replaced by the single symbol tree c, the other copy is obtained by processing the children in state q. No parent node is given for this copy. So, there is no corresponding node for the current node in the input tree. We, therefore, say it is deleted. For instance, $T^q(a(b)) = c\,d$ where d corresponds to b and not to a.

2.4 The Typechecking Problem

We define the problem central to this paper.

Definition 5. A tree transducer T *typechecks* w.r.t. to an input tree language S_{in} and an output tree language S_{out}, if $T(t) \in S_{\text{out}}$ for every $t \in S_{\text{in}}$.

We parameterize the typechecking problem by the kind of tree transducers and tree languages we allow. Let $\mathcal{T}$ be a class of transducers and $\mathcal{S}$ be a class of tree languages. Then TC$[\mathcal{T}, \mathcal{S}]$ denotes the typechecking problem where $T \in \mathcal{T}$ and $S_{\text{in}}, S_{\text{out}} \in \mathcal{S}$. The size of the input of the typechecking problem is the sum of the sizes of the input and output schema and the tree transducer.

A transducer T has *width* k if there are at most k occurrences of states in every rhs of T. By $\mathcal{BW}_k$ we denote the class of transducers of width k. A transducer is *non-deleting* if no states occur at the top-level of a rhs. We denote by $\mathcal{T}_g$ the class of all transducers and by $\mathcal{T}_{nd}$ the class of non-deleting transducers. For a class of representations of regular string languages $\mathcal{M}$, we write TC$[\mathcal{T}$,$\mathcal{M}]$ rather than TC$[\mathcal{T}$,DTD$(\mathcal{M})]$.

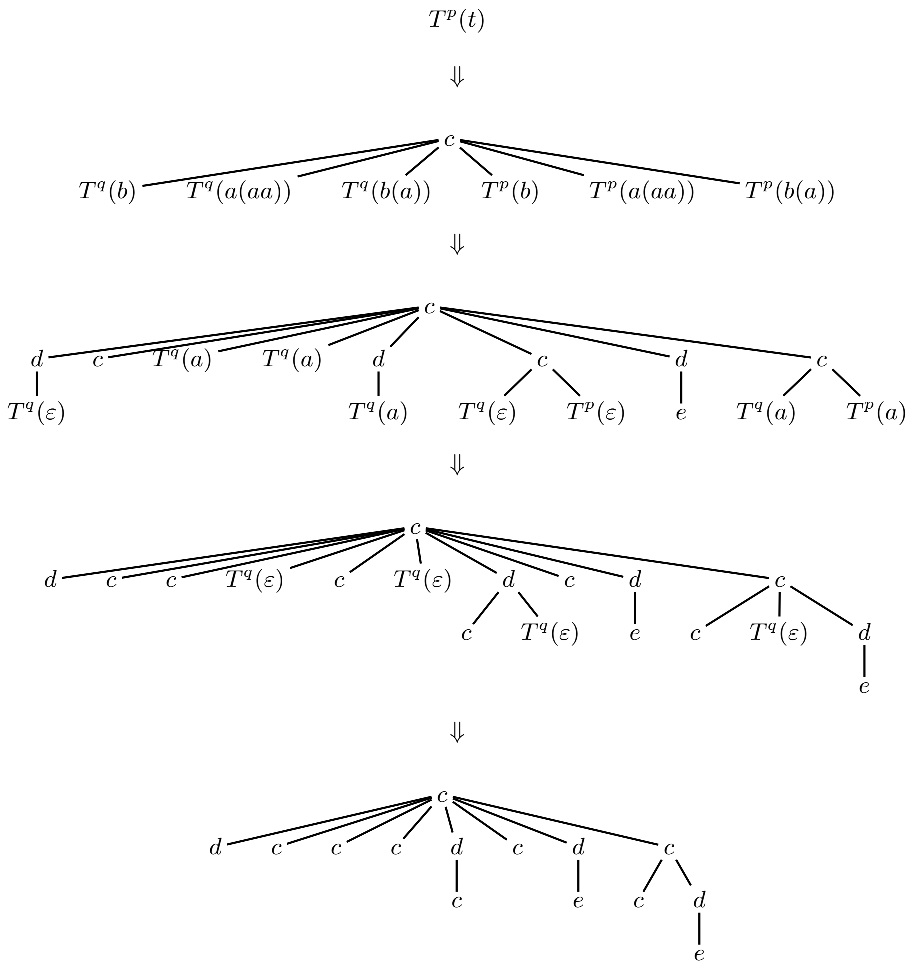
3 The General Case

When we do not restrict our transducers in any way, the typechecking problem is in EXPTIME and is EXPTIME-hard for even the simplest DTDs: those where the right-hand sides are specified with $\mathcal{SL}$-formulas or with DFAs. The main reason is that the deleting states allow the transducer to simulate deterministic top-down tree automata in such a way that the transducer produces no output besides acceptance information. In this way, even a very simple DTD can check whether the output was rejected or not. By copying the input several times, we can execute several deterministic tree automata in parallel. These are all the ingredients we need for a reduction from non-emptiness of the intersection of an arbitrary number of deterministic tree automata which is known to be EXPTIME-hard.

Theorem 2. *1. TC[$\mathcal{T}_g$,NTA] is in* EXPTIME*;*
2. TC[$\mathcal{T}_g$,$\mathcal{SL}$] is EXPTIME*-hard;*
3. TC[$\mathcal{T}_g$,DFA] is EXPTIME*-hard.*

Proof. (Sketch) (1) Let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a transducer and let A_{in} and $A_{\text{out}} = (Q_A, \Sigma, \delta_A, F_A)$ be two NTAs representing the input and output schema, respectively. We next describe a *non*-deleting transducer S and an NTA B_{out} which can be constructed in LOGSPACE, such that T typechecks w.r.t. A_{in} and A_{out} iff S typechecks w.r.t. A_{in} and B_{out}. From Theorem 3(1) it then follows that TC$[\mathcal{T}_g$,NTA$]$ is in EXPTIME.

Intuitively, S puts a # whenever T would process a deleting state. For instance, the rule $(q, a) \to c\, q$ is replaced by $(q, a) \to c\, \#(q)$. We introduce some notation to characterize the behavior of B_{out}. Define the #-eliminating function γ as follows: $\gamma(\sigma(h))$ is $\gamma(h)$ when $\sigma = \#$ and $\sigma(\gamma(h))$ otherwise; further, $\gamma(t_1 \cdots t_n) := \gamma(t_1) \cdots \gamma(t_n)$. Then, clearly, for all $t \in \mathcal{T}_\Sigma$, $T(t) = \gamma(S(t))$. B_{out} then accepts a tree t iff $\gamma(t) \in L(A_{\text{out}})$.



$T^p(t)$
$\Downarrow$
c
$T^q(b)$
$T^q(a(aa))$
$T^q(b(a))$
$T^p(b)$
$T^p(a(aa))$
$T^p(b(a))$
$\Downarrow$
d c $T^q(a)$ $T^q(a)$ d c d c
$T^q(\varepsilon)$ $T^q(a)$ $T^q(\varepsilon)$ $T^p(\varepsilon)$ e $T^q(a)$ $T^p(a)$
$\Downarrow$
d c c $T^q(\varepsilon)$ c $T^q(\varepsilon)$ d c d c
c $T^q(\varepsilon)$ e c $T^q(\varepsilon)$ d
e
$\Downarrow$
d c c c d c d c
c e c d
e

Fig. 2. The translation of $t = b(b\,a(a\,a)b(a))$ by the transducer T of Example 1.

(2) We use a reduction from the intersection problem of deterministic binary top-down tree automata A_i ($i = 1 \ldots n$), which is known to be hard for EXPTIME [21]. The problem is stated as follows, given deterministic binary top-down automata $A_1, \ldots, A_n$, is $\bigcap_{i=1}^n L(A_i) = \emptyset$? We define a transducer T and two DTDs d_{in} and d_{out} such that $\bigcap_{i=1}^n L(A_i) \neq \emptyset$ iff T does not typecheck w.r.t. d_{in} and d_{out}. In the construction, we exploit the copying power of transducers to make n copies of the input tree: one for each A_i. By using deleting states, we can execute each A_i on its copy of the input tree without producing output. When an A_i does not accept, we output an *error* symbol under the root of the output tree. The output DTD should then only check that an *error* symbol always appears.

The proof of (3) is similar to the one for (2). □

4 Non-deleting Transformations

In an attempt to lower the complexity, we consider, in the present section, non-deleting transformations w.r.t. various schema formalisms. We observe that when schemas are represented by tree automata, the complexity remains EXPTIME-hard. When tree languages are represented by DTDs, the complexity of the typechecking problem drops to PSPACE and is hard for PSPACE even when right-hand sides of rules are represented by DFAs. The main reason for this is that the tree transducers can still make an unbounded number of copies of the input tree. This allows to simulate in parallel an unbounded number of DFAs and makes it possible to reduce the intersection emptiness problem of DFAs to the typechecking problem. In the next section, we therefore constrain this copying power. In summary, we prove the following results:

Theorem 3. *1. TC[$\mathcal{T}_{nd}$,NTA] is* EXPTIME-*complete;*
2. TC[$\mathcal{T}_{nd}$,DTA] is EXPTIME-*complete;*
3. TC[$\mathcal{T}_{nd}$,NFA] is PSPACE-*complete;*
4. TC[$\mathcal{T}_{nd}$,DFA] is PSPACE-*complete;*
5. TC[$\mathcal{T}_{nd}$,$\mathcal{SL}$] is CONP-*complete.*

4.1 Tree Automata

Consider the case where the input and output schemas are represented by NTA(NFA)s. One way to obtain an appropriate typechecking algorithm, would be to build a composite automaton A_{co} that on input t, runs A_{out} on $T(t)$ without actually constructing $T(t)$. The given instance typechecks iff $L(A_{\text{in}}) \subseteq L(A_{\text{co}})$. However, constructing A_{co} would lead to an exponential blowup because the state set of A_{co} would be $2^{Q_T} \times 2^{Q_{out}}$. Since A_{co} is nondeterministic because A_{out} is nondeterministic, solving the inclusion problem on this instance would lead to a double exponential time algorithm. Thus, to show that TC[$\mathcal{T}_{nd}$,NTA] can be solved in EXPTIME, we need a slightly more sophisticated approach.

Theorem 3(1). *TC[$\mathcal{T}_{nd}$,NTA] is* EXPTIME-*complete.*

Proof. (Sketch) Hardness is immediate as containment of NTAs is already hard for EXPTIME [20]. We, therefore, only prove membership in EXPTIME. The proof is similar in spirit to a proof in [18], which shows that containment of Query Automata is in EXPTIME. Let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a non-deleting tree transducer and let $A_{\text{in}} = (Q_{in}, \Sigma, \delta_{in}, F_{in})$ and $A_{\text{out}} = (Q_{out}, \Sigma, \delta_{out}, F_{out})$ be the NTAs representing the input and output schema, respectively.

For ease of exposition, we restrict hedges in the rhs of T to be trees. In brief, our algorithm computes the set

$$P = \{(S, f) \mid S \subseteq Q_{in}, f : Q_T \to 2^{Q_{out}}, \exists t \text{ such that} \\ S = \delta_{in}^*(t) \text{ and } \forall q \in Q_T, f(q) = \delta_{out}^*(T^q(t))\}.$$

Intuitively, in the definition of P, t can be seen as a witness of (S, f). That is, S is the set of states reachable by A_{in} at the root of t, while for each state q of the transducer, $f(q)$ is the set of states reachable by A_{out} at the root of $T^q(t)$ (recall that this is the translation of t started in state q). So, the given instance does *not* typecheck iff there exists an $(S, f) \in P$ such that $F_{in} \cap S \neq \emptyset$ and $F_{out} \cap f(q_T^0) = \emptyset$. In Figure 3, an algorithm for computing P is depicted. By $\text{rhs}(q, a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T]$, we denote the tree obtained from $\text{rhs}(q, a)$ by replacing every occurrence of a state p by the sequence $f_1(p) \cdots f_n(p)$. For $c \in \{in, out\}$, $\delta_c^* : \mathcal{T}_\Sigma(2^{Q_c}) \to 2^{Q_c}$ is the transition function extended to trees in $\mathcal{T}_\Sigma(2^{Q_c})$. To be precise, for $a \in \Sigma$, $\delta_c^*(a) := \{q \mid \varepsilon \in \delta_c(q, a)\}$; for $P \subseteq Q_c$, $\delta_c^*(P) := P$; and, $\delta_c^*(a(t_1 \cdots t_n)) := \{q \mid \exists q_i \in \delta_c^*(t_i) : q_1 \cdots q_n \in \delta_c^*(q, a)\}$. The correctness of the algorithm follows from the following lemma which can be easily proved by induction.

Lemma 1. *A pair (S, f) has a witness tree of depth i iff $(S, f) \in P_i$.*

It remains to show that the algorithm is in EXPTIME. The set P_1 can be computed in time polynomial in the sizes of A_{in}, A_{out}, and T. As $P_i \subseteq P_{i+1}$ for all i, the loop can only make an exponential number of iterations. It, hence, suffices to show that each iteration can be done in EXPTIME. Actually, we argue that it can be checked in PSPACE whether a tuple $(S, f) \in P_i$. Indeed, the question whether there are $(S_1, f_1) \cdots (S_n, f_n) \in P_{i-1}^*$ can be reduced to the emptiness test of a 2AFA A which works on strings over the alphabet P_{i-1}. On input $(S_1, f_1) \cdots (S_n, f_n)$ the 2AFA A operates as follows: for every $p \in S$ it checks whether there are $r_i \in S_i$ such that $r_1 \ldots r_n \in \delta(p, a)$. This can be done by $|S|$ traversals through the input string. Next, A checks for every $q \in Q_{in} \setminus S$ whether for all $r_i \in S_i$, $r_1 \ldots r_n \notin \delta(q, a)$. This can be done by $|Q_{in} \setminus S|$ traversals through the input string while using alternation. In a similar way $f(q)$ is checked. The automaton A is exponential in the input. However, we can construct A on the fly when executing the PSPACE algorithm for non-emptiness. The latter algorithm is an adaptation of the technique used by Vardi [26]. As there are only exponentially many tuples (S, f), the overall algorithm is in EXPTIME. □

In the remainder of this section, we examine what happens when tree automata are restricted to be deterministic. From the above result, it is immediate

```
$P_0 := \emptyset$;
$i := 1$;
$P_1 := \{(S, f) \mid \exists a : \forall r \in S : \varepsilon \in \delta_{in}(r, a), \forall q \in Q_T : f(q) = \delta^*_{out}(T^q(a))\}$;
**while** $P_i \neq P_{i-1}$ **do**
  $P_i := \big\{(S, f) \mid \exists (S_1, f_1) \cdots (S_n, f_n) \in P^*_{i-1}, \exists a \in \Sigma :$
              $S = \{p \mid \exists r_k \in S_k, k = 1 \ldots n, r_1 \cdots r_n \in \delta_{in}(p, a)\}$
              $\forall q \in Q_T : f(q) = \delta^*_{out}\big(\mathrm{rhs}(q, a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T]\big)\big\}$;
  $i := i + 1$;
**end while**
$P := P_i$;
```

Fig. 3. The algorithm of Theorem 3(1) computing P.

that TC[$\mathcal{T}_{nd}$,DTA] is in EXPTIME. To show that it is hard, we can use a reduction from the intersection problem of deterministic binary top-down tree automata like in the proof of Theorem 2(2). The reduction is almost identical to the one in Theorem 2(2): A_{in} defines the same set of trees as d_{in} does with the exception that A_{in} enforces an ordering of the children. The transducer in the proof of Theorem 2(2) starts the in parallel simulation of the n automata, but then, using deleting states, delays the output until it has reached the leaves of the input tree. In the present setting, we can not use deleting states. Instead, we copy the input tree and attach error-symbols to the leaves when an automaton rejects. The output automaton then checks whether at least one error occurred. We obtain the following theorem.

Theorem 3(2) *TC[$\mathcal{T}_{nd}$,DTA] is* EXPTIME*-complete.*

4.2 DTDs

When we consider DTDs as input schemas the complexity drops to PSPACE and CONP.

Theorem 3(3) *TC[$\mathcal{T}_{nd}$,NFA] is* PSPACE*-complete.*

Proof. (Sketch) The hardness result is immediate as containment of regular expressions is known to be PSPACE-hard [22]. For the other direction, let T be a non-deleting tree transducer. Let d_{in} and d_{out} be the input and output DTDs, respectively. We construct an NTA(2AFA) B such that $L(B) = \{t \in L(d_{\text{in}}) \mid T(t) \notin d_{\text{out}}\}$. Moreover, the size of B is polynomial in the size of T, d_{in}, and d_{out}. Thus, $L(B) = \emptyset$ iff T typechecks w.r.t. d_{in} and d_{out}. By Theorem 1(2), the latter is in PSPACE. To explain the operation of the automaton we introduce the following notion: let q be a state of T and $a \in \Sigma$ then define $q(a) := z$ where z is the concatenation of the labels of the top most nodes of rhs(q, a). For a string $w := a_1 \cdots a_n$, we define $p(w) := p(a_1) \cdots p(a_n)$. Intuitively, the automaton B now works bottom-up as follows: (1) B checks that $t \in L(d_{\text{in}})$; (2) at the same time, B guesses a node v labeled σ with n children and picks a state q in which

v is processed: B then accepts if h does not satisfy d_{out}, where h is obtained from $\text{rhs}(\sigma, q)$ by replacing every state p by $p(\text{lab}^t(u1) \cdots \text{lab}^t(un))$. As d_{out} is specified by NFAs and we have to check that d_{out} is *not* satisfied, we need alternation to specify the transition function of B. Additionally, as T can copy its input, we need two-way automata.

Formally, let $T = (Q_T, \Sigma, q_0^T, \delta_T)$. Define $B = (Q^B, \Sigma, F^B, \delta^B)$ as follows. The set of states Q^B is the union of the following sets: Σ, $\{(\sigma, q) \mid q \in Q_T, \sigma \in \Sigma\}$, and $\{(\sigma, q, \text{check}) \mid q \in Q_T, \sigma \in \Sigma\}$. If there is an accepting run on a tree t, then a node v labeled with a state of the form σ, (σ, q), $(\sigma, q, \text{check})$ has the following meaning:

σ: the current node is labeled with σ and the subtree rooted at this node satisfies d_{in}.

(σ, q): same as in previous case with the following two additions: (1) v is processed by T in state q; and, (2) a descendant of v will produce a tree that will not satisfy d_{out}.

$(\sigma, q, \text{check})$: same as the previous case only now v itself will produce a tree that does not satisfy d_{out}.

The set of final states is defined as follows: $F^B := \{(\sigma, q_0^T) \mid \sigma \in \Sigma\}$. The transition function is defined as follows:

1. $\delta^B(a, b) = \delta^B((a, q), b) = \delta^B((a, q, \text{check}), b) = \emptyset$ for all $a \neq b$;
2. $\delta^B(a, a) = d_{\text{in}}(a)$ and

   $$\delta^B((a, q), a) = \{\Sigma^*(b, p)\Sigma^* + \Sigma^*(b, p, \text{check})\Sigma^* \mid p \text{ occurs in } \text{rhs}(q, a), b \in \Sigma\},$$

   for all $a \in \Sigma$ and $q \in Q_T$.
3. Finally, $\delta^B((a, q, \text{check}), a) = \{a_1 \cdots a_n \mid h \notin L(d_{\text{out}}) \text{ and } a_1 \cdots a_n \in d_{\text{in}}(a)\}$. Here, h is obtained from $\text{rhs}(q, a)$ by replacing every q by $q(a_1 \cdots a_n)$.

We are left with the proof that $\delta^B((a, q, \text{check}), a)$ can be computed by a 2AFA A with only a polynomial blowup. Before we define A, we define some other automata. First, for every $b \in \Sigma$, let A_b be the NFA accepting $d_{out}(b)$.

For every v in $\text{rhs}(q, a)$, let w be the largest string in $(\Sigma \cup Q_T)^*$ such that $\text{lab}^h(v)(w)$ is a subtree rooted at v in h. Define the 2NFA B_v as follows: suppose w is of the form $z_0 p_1 z_1 \cdots p_\ell z_\ell$, then $a_1 \cdots a_n \in L(B_v)$ if and only if $z_0 p_1(a_1 \cdots a_n) z_1 \cdots p_\ell(a_1 \cdots a_n) z_\ell \in L(A_{\text{lab}^h(v)})$. As w is fixed, B_v can recognize this language by reading $a_1 \cdots a_n$ ℓ times while simulating $A_{\text{lab}^h(v)}$. Intuitively, the automaton simulates $A_{\text{lab}^h(v)}$ on $z_{i-1} p_i(a_1 \cdots a_n)$ on the ith pass.

It remains to describe the construction of A. To this end, let A^a_{in} be the NFA such that $d_{\text{in}}(a) = L(A^a_{in})$. On input $a_1 \cdots a_n$, A first checks whether $a_1 \cdots a_n \in L(A^a_{in})$ by simulating A^a_{in}. After this, A goes back to the beginning of the input string, guesses an internal node u in $\text{rhs}(q, a)$ and simulates the negation of B_u. As B_u is a 2NFA, A is a 2AFA. $\square$

The intersection problem of deterministic finite automata is known to be PSPACE-hard [10] and is easily reduced to TC[$\mathcal{T}_{nd}$,DFA]. This implies the following result:

Theorem 3(4) *TC[$\mathcal{T}_{nd}$,DFA] is* PSPACE*-complete.*

Using $\mathcal{SL}$-expressions to define right-hand sides of DTDs reduces the complexity of typechecking to CONP.

Theorem 3(5) *TC[$\mathcal{T}_{nd}$,$\mathcal{SL}$] is* CONP*-complete.*

Proof. (Sketch) First, we prove the hardness result. Let ϕ be a SAT-formula and let $v_1, \ldots, v_n$ be the variables occurring in ϕ. We define the typechecking instance as follows. $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$. We only define d_{in} and d_{out} for σ_1, since this is all we require. $d_{\text{in}}(\sigma_1) = \phi'$, where ϕ' is the formula ϕ with every occurrence of v_i replaced by $\sigma_i^{=1}$ for $i = 1 \ldots n$. The transducer T is the identity, and $d_{out}(\sigma_1) = \emptyset$. Hence, this instance typechecks iff ϕ is *not* satisfiable.

To prove the upper bound, let $T = (Q_T, \Sigma, q_T^0, R_T)$ and let $(d_{\text{in}}, s_{\text{in}})$ and $(d_{\text{out}}, s_{\text{out}})$ be the input and output DTD respectively. We describe an NP algorithm that accepts iff the given instance does *not* typecheck.

We introduce some notation. For a DTD (d, s_d) and $\sigma \in \Sigma$, we denote by d^σ the DTD d with start symbol σ, that is, (d, σ). Let k be the largest number occurring in an $\mathcal{SL}$-formula in d_{in}. Set $r = (k+1) \times |\Sigma|$.

The algorithm consists of three main parts:

1. First, we sequentially guess a subset L of the derivable symbols $\{b \in \Sigma \mid L(d_{\text{in}}^b) \neq \emptyset\}$.
2. Next, we guess a path of a tree in d_{in}. In particular, we guess a sequence of pairs $(a_i, q_i) \in L \times Q_T$, $i = 0, \ldots, m$, with $m \leq |\Sigma| \times |Q_T|$, such that
   (a) $a_0 = s_{\text{in}}$ and $q_0 = q_T^0$;
   (b) $\exists t, \exists u \in \text{Dom}(t)$ such that $a_0 \cdots a_m$ is the concatenation of the labels of the nodes on the path from the root to u; and,
   (c) $\forall i = 1, \ldots, m$: T visits a_i in state q_i.
3. Finally, we guess a string $w \in L^*$ of length at most r such that $T^{q_m}(a_m(w)) \notin L(d_{\text{out}}^\sigma)$ with σ the root symbol of $T^{q_m}(a_m(w))$.

All guesses can be done at once and can be checked by a polynomial verifier. This completes the description of the algorithm. □

5 Transducers of Bounded Width

When we put a bound on the width (or copying power, recall the discussion at the end of sections 2.3 and 2.4) of transducers we get a PTIME algorithm for typechecking when the right-hand sides of DTDs are represented by DFAs. All other results have the same complexity as in the case of unrestricted copying.

Theorem 4. *1. TC[$\mathcal{BW}_k$,NTA] is* EXPTIME*-complete;*
2. TC[$\mathcal{BW}_k$,RE] is PSPACE*-complete;*
3. TC[$\mathcal{BW}_k$,DFA] is PTIME*-complete;*
4. TC[$\mathcal{BW}_k$,$\mathcal{SL}$] is CONP*-complete.*

The lower bounds of (1), (2), and (4) follow immediately from the construction in the proofs of Theorem 3(1), (3), and (5).

Theorem 4(3) *TC[$\mathcal{BW}_k$,DFA] is* PTIME*-complete.*

Proof. (Sketch) In the proof of Theorem 3(3), TC[$\mathcal{T}_{nd}$,NFA] is reduced to the emptiness of NTA(2AFA)s. Alternation was needed to express negation of NFAs; two-wayness was needed because T could make arbitrary copies of the input tree. However, when transducers can make only a bounded number of copies and DFAs are used, TC[$\mathcal{BW}_k$,DFA] can be LOGSPACE-reduced to emptiness of NTA(NFA)s. From Theorem 1(1), it then follows that TC[$\mathcal{BW}_k$,DFA] is in PTIME. A PTIME lower bound is obtained by a reduction from PATH SYSTEMS [8]. □

6 Conclusion

Motivated by structural recursion and XSLT, we studied typechecking for top-down XML transformers in the presence of both DTDs and tree automata. In this setting the complexity of the typechecking problem ranges from PTIME to EXPTIME. In particular, the PTIME algorithm is obtained by restricting to non-deleting tree transducers of bounded width and DTD(DFA)s. The main open question for future research is how these restrictions can be relaxed while still having a PTIME algorithm. Another question we left open is the exact complexity of TC[$\mathcal{BW}_k$,DTA] which is in EXPTIME and PSPACE-hard.

References

1. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. In *Proc. 16th IEEE Symposium on Logic in Computer Science (LICS 2001)*, pages 421–130, 2001.
2. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *Proc. 20th Symposium on Principles of Database Systems (PODS 2001)*, pages 560–572, 2001.
3. G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.
4. A. Brüggemann-Klein, 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.
5. P. Buneman, M. Fernandez, and D. Suciu. UnQl: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.
6. James Clark. XSL transformations version 1.0. http://www.w3.org/TR/WD-xslt, august 1999.
7. World Wide Web Consortium. XML Schema. http://www.w3.org/XML/Schema.
8. S.A. Cook. An observation on time-storage trade-off. *Journal of Computer and System Sciences*, 9(3):308–316, 1974.
9. J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 1985.

10. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
11. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 1, pages 1–68. Springer, 1997.
12. R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer. Alternating pushdown and stack automata. *SIAM Journal on Computing*, 13(1):135–155, 1984.
13. S. Maneth and F. Neven. Structured document transformations based on XSL. In R. Connor and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming (DBPL'99)*, volume 1949 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2000.
14. T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*, pages 215–226. ACM Press, 1999.
15. T. Milo, D. Suciu, and V. Vianu. Type checking for XML transformers. In *Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems*, pages 11–22. ACM Press, 2000.
16. F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.
17. F. Neven and T. Schwentick. XML schemas without order. Unpublished manuscript, 1999.
18. F. Neven and T. Schwentick. Query automata on finite trees. *Theoretical Computer Science*, 275:633–674, 2002.
19. Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. 20th Symposium on Principles of Database Systems (PODS 2001)*, pages 35–46. ACM Press, 2001.
20. H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.
21. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.
22. L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *Conference Record of Fifth Annual ACM Symposium on Theory of Computing*, pages 1–9, Austin, Texas, 30 April–2 May 1973.
23. D. Suciu. Typechecking for semistructured data. In *Proceedings of the 8th Workshop on Data Bases and Programming Languages (DBPL 2001)*, 2001.
24. D. Suciu. The XML typechecking problem. *SIGMOD Record*, 31(1):89–96, 2002.
25. A. Tozawa. Towards static type checking for XSLT. In Proceedings of ACM Symposium on Document Engineering, 2001.
26. M. Y. Vardi. A note on the reduction of two-way automata to one-way automata. *Information Processing Letters*, 30:261–264, March 1989.