Frontiers of Tractability for Typechecking Simple XML Transformations

Wim Martens
Limburgs Universitair Centrum
Universitaire Campus
B-3590 Diepenbeek, Belgium
wim.martens@luc.ac.be

Frank Neven
Limburgs Universitair Centrum
Universitaire Campus
B-3590 Diepenbeek, Belgium
frank.neven@luc.ac.be

ABSTRACT

Typechecking consists of statically verifying whether the output of an XML transformation is always conform to an output type for documents satisfying a given input type. We focus on complete algorithms which always produce the correct answer. We consider top-down XML transformations incorporating XPath expressions and abstract document types by grammars and tree automata. By restricting schema languages and transformations, we identify several practical settings for which typechecking is in polynomial time. Moreover, the resulting framework provides a rather complete picture as we show that most scenarios can not be enlarged without rendering the typechecking problem intractable. So, the present research sheds light on when to use fast complete algorithms and when to reside to sound but incomplete ones.

1. INTRODUCTION

In a typical XML data exchange scenario on the web, a user community creates a common schema and agrees on producing only XML 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 or document transformation applied to a valid input document, satisfies the output schema [24, 25].

The main goal of this paper is to determine relevant scenarios for which typechecking becomes tractable. Additionally, we also want to identify the frontier of tractability for these scenarios. As typechecking quickly becomes intractable [2, 16, 19], we focus on simple but practical XML transformations where only little restructuring is needed, like for instance in filtering of documents. Transformations that can for example be expressed by structural recursion [5] or by a top-down fragment of XSLT [3]. As is accustomed, we abstract such transformations by unranked tree transducers [14, 16]. As types we adopt the usual Document Type Definitions (DTDs) and their robust extension: regular tree languages [19, 13] or, equivalently, specialized DTDs [22, 23]. The latter serve as a formal model for XML Schema [28].

In earlier work, we identified three sources of complexity for the typechecking problem in the above setting: non-determinism in the regular expressions in the output DTD, the ability of the transformation to make arbitrary copies of subtrees, and the capability to delete (rather than rename or replace) nodes of the input document [16]. In fact, the only PTIME typechecking instance we obtained, was by disallowing all three parameters. As the latter scenario is overly restrictive, especially since it excludes every form of deletion, we investigate in this paper larger and more flexible classes for which the complexity of the typechecking problem remains in PTIME.

We first note that the scenario studied in [16] is very general: both the schemas and the transducer were determined to be part of the input. However, for some exchange scenarios it makes sense to fix the input and/or output schema: for instance, when considering a common schema within a community or when translating data from one community to another. Therefore, we first revisit the various instances of the typechecking problem considered in [16] and determine the complexity in the presence of fixed input and/or output schemas. The obtained results are summarized in Table 2 and explained in Section 3. In particular, we show that for non-deleting transducers and fixed input and output schemas, we can allow arbitrary copying and still have a tractable typechecking algorithm. Unfortunately, we also show that in all new settings the typechecking problem remains intractable when allowing deletion or using tree automata.

As illustrated by Example 3, deletion of an arbitrary number of interior nodes is quite typical for filtering transformations. Indeed, many simple transformations select specific parts of the input while deleting the non-interesting ones. We therefore explore ways to preserve tractability but admit restricted forms of deletion.

First, we investigate deletion in the setting where DTDs use DFAs to define right-hand sides of rules and transducers can only make a bounded number of copies of nodes in the input tree. By proving a general lemma which quantifies the combined effect of copying and deletion on the complexity of typechecking, we derive conditions under which typechecking becomes tractable. In particular, these conditions allow arbitrary deletion when no copying occurs (like in Example 3), but at the same time permit limited copying for those rules that only delete in a limited fashion. This result is relevant in practice as in common filtering transformations arbitrary deletion almost never occurs together with copying.

We then show that the present setting cannot be enlarged without increasing the complexity. In particular, we show that combining arbitrary deletion with the ability of copying the input only twice, or a slight relaxation of the limited deletion restriction makes typechecking intractable. Finally, we briefly examine tree automata to define schemas and show that in the case of deterministic tree automata, no copying but arbitrary deletion, we get a PTIME algorithm.

The first PTIME result still relies on a uniform bound on the number of copies a rule of the transducer can make. Although this number will always be fairly small in practice, it would still be more elegant to have an algorithm which is tractable for any transducer. Thereto, we have to restrict the schema languages. In fact, we show that only for very weak DTDs, those where all regular expressions are concatenations of symbols a and a^+, typechecking becomes tractable, and that obvious extensions of such expressions make the problem at least CONP-hard. So, the price for arbitrary deletion and copying is very high.

As an alternative to deletion, one can skip nodes in the input tree by by adding XPath expressions to the transformation language. In the case where DTDs use DFAs, we obtain a tractable fragment by translating the transformation language to transducers without XPath expressions. As XPath containment in the presence of DTDs [21, 27] can easily be reduced to the typechecking problem, lower bounds establishing intractability readily follow for XPath fragments containing filter and disjunction. We leave one case open. We only prove an initial result for the case where DTDs use RE$^+$ expressions.

Finally, we address how to generate counterexamples when an instance fails to typecheck and consider a slight adaptation of the typechecking problem: *almost always* typechecking. The latter problem was first discussed by Engelfriet and Maneth [12] and asks whether there exist only a finite number of counterexample trees for a given instance. We argue that the PTIME algorithms in Section 4 can also be used for almost always typechecking.

Complete vs. Incomplete. Our work studies sound and complete typechecking algorithms, an approach that should be contrasted with the work on general purpose XML programming languages like XDuce [11] and CDuce [8], for instance, where the main objective is fast and sound but sometimes incomplete typechecking. So, sometimes transformations are typesafe but are rejected by the typechecker. As we only consider very simple and by no means Turing-complete transformations, it makes sense to ask for complete algorithms. In fact, the present paper sheds light on precisely when we can get fast complete algorithms and when we should start looking for incomplete ones.

Related Work. The research on typechecking XML transformations is initiated by Milo, Suciu, and Vianu [19]. They obtained the decidability for typechecking of transformations realized by k-pebble transducers via a reduction to satisfiability of monadic second-order logic. Unfortunately, in this general setting, the latter non-elementary algorithm cannot be improved [19]. Interestingly, typechecking of k-pebble transducers has recently been related to typechecking of compositions of macro tree transducers [12]. Alon et al. [1, 2] investigated typechecking in the presence of data values and show that the problem quickly turns undecidable. A problem related to typechecking is type inference [18, 22]. 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 [22]. The transducers considered in the present paper are restricted versions of the ones studied by Maneth and Neven [14]. 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). Tozawa considered typechecking w.r.t. tree automata for a fragment of top-down XSLT [26]. His framework is more general but he only obtains a double exponential time algorithm. It is not clear whether that upper bound can be improved.

Organization. The remainder of the paper is organized as follows. In Section 2, we provide the necessary definitions. In Section 3, we discuss typechecking in the restricted settings of fixed output and/or input schemas. In Section 4, we consider deleting transducers. In Section 5, we discuss DTDs with RE$^+$ expressions. In Section 6, we discuss the addition of XPath. In Section 7, we present some observations. We conclude in Section 8. Complete proofs can be found in [15].

2. DEFINITIONS

In this section we provide the necessary background on trees, automata, and tree transducers. We fix a finite alphabet Σ.

2.1 Trees, Hedges, DTDs, and Tree Automata

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 $\mathcal{T}_\Sigma$. So, a tree is either ε (empty) or is of the form $\sigma(t_1 \cdots t_n)$ where each t_i is a tree. The latter denotes the tree where the subtrees $t_1, \ldots, t_n$ are attached to the root labeled σ. 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 *tree language* is a set of trees.

Later, in the right-hand side of transducer rules we will allow hedges: 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 Dom(h), is the subset of $\mathbf{N}^*$ defined as follows: (i) if $h = \varepsilon$, then Dom$(h) = \emptyset$; (ii) if $h = t_1 \cdots t_n$ where each $t_i \in \mathcal{T}_\Sigma$, then Dom$(h) = \bigcup_{i=1}^n \{iu \mid u \in \text{Dom}(t_i)\}$; and, (iii)

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. For a hedge $h = t_1 \cdots t_n$, $\mathrm{top}(h)$, is the string obtained by concatenating the root symbol of every t_i.

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

We denote by $\mathrm{DTD}(\mathcal{M})$ 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.

We recall the definition of non-deterministic tree automata from [4]. We refer the unfamiliar reader to [20] for a gentle introduction.

Definition 2. 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 $\delta : Q \times \Sigma \to 2^{Q^*}$ such that $\delta(q, a)$ is a regular string language over Q for every $a \in \Sigma$ and $q \in Q$. $\square$

A *run* of B on a tree t is a labeling $\lambda : \mathrm{Dom}(t) \to Q$ such that for every $v \in \mathrm{Dom}(t)$ with n children, $\lambda(v1) \cdots \lambda(vn) \in \delta(\lambda(v), \mathrm{lab}^t(v))$. Note that when v has no children, then the criterion reduces to $\varepsilon \in \delta(\lambda(v), \mathrm{lab}^t(v))$. 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 $L(B)$ and is called a *regular tree language*.

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 NTAs by DTA.

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

2.2 Transducers

We adhere to transducers as a formal model for simple transformations corresponding to structural recursion [5] and a fragment of top-down XSLT. Like in [19], the abstraction focuses on structure rather than on content. 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. Of course one can 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 3. A *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 $\mathcal{T}_\Sigma(Q) \setminus Q$. $\square$

The restriction on rules with the initial state ensures that the output is always a tree rather than a hedge. Transducers are required to be deterministic: for every pair (q, a) there is at most one rule in R.

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

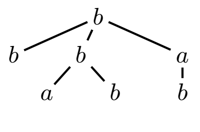
$$\begin{array}{ll} (p, a) \to d(e) & (p, b) \to d(q) \\ (q, a) \to c\ p & (q, b) \to c(p\ q) \end{array}$$

The XSLT program equivalent to the above transducer is given in Figure 1 (we assume the program is started in mode p). Note that the right-hand side of $(q, a) \to c\ p$ is a hedge, while the other right-hand sides are trees. $\square$

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 every node u in h labeled with state 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 $|Q| + |\Sigma| + \sum_{q \in Q, a \in \Sigma} |\mathrm{rhs}(q, a)|$. In the sequel, we always use $p, p_1, p_2, \ldots$ and $q, q_1, q_2, \ldots$ to denote states.

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



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

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

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

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

Figure 1: The XSLT program equivalent to the transducer of Example 1.

by the transducer of Example 1. In order to save space, we did not list $T^q(\varepsilon)$ and $T^p(\varepsilon)$. □

2.3 Copying and Deletion

We discuss two important features: *copying* and *deletion*. In Example 1, the rule $(q, b) \to c(p\,q)$ copies the children of the current node in the input tree two times: one copy is processed in state p and the other in state q. 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 $(q, a) \to c\,p$ copies the children of the current node only once. However, 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 that it is deleted. For instance, $T^q(a(b)) = c\,d$ where d corresponds to b and not to a.

We illustrate the functionality of copying and deleting by means of a typical filtering example.

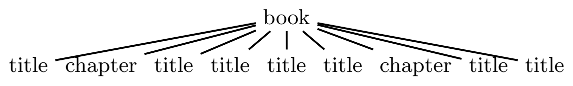
Example 3. The following DTD(DFA) defines a schema for books:

$$\begin{array}{lll} \texttt{book} & \to & \texttt{title}, \texttt{author}^+, \texttt{chapter}^+ \\ \texttt{chapter} & \to & \texttt{title}, \texttt{introduction}, \texttt{section}^+ \\ \texttt{section} & \to & \texttt{title}, \texttt{paragraph}^+, \texttt{section}^* \end{array}$$

Here, comma denotes concatenation. Figure 3 depicts a document conforming to the given schema. The following transducer generates a table of contents: that is, for every chapter of the book a list of its section titles.

$$\begin{array}{l} (q, \texttt{book}) \to \texttt{book}(q) \\ (q, \texttt{chapter}) \to \texttt{chapter}\ q \\ (q, \texttt{title}) \to \texttt{title} \\ (q, \texttt{section}) \to q \end{array}$$

The document in Figure 3 is transformed into the tree



The example illustrates the usefulness of deleting states: all intermediate sections are skipped. Further, the rule

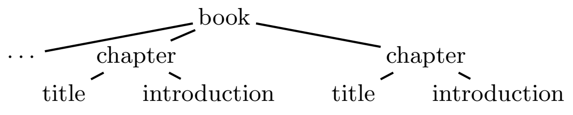
$$(q, \texttt{chapter}) \to \texttt{chapter}\ q$$

allows to list all section titles next to the chapter element rather than below.

Next, we illustrate copying. The following transducer extends the previous one by adding a summary of the book to the table of contents. The summary is given by listing the title and introduction of each chapter. By using the two states p and p', we make sure that the title of the book is not printed in the summary.

$$\begin{array}{l} (q, \texttt{book}) \to \texttt{book}(q\ p) \\ (q, \texttt{chapter}) \to \texttt{chapter}\ q \\ (q, \texttt{title}) \to \texttt{title} \\ (q, \texttt{section}) \to q \\ (p, \texttt{chapter}) \to \texttt{chapter}(p') \\ (p', \texttt{title}) \to \texttt{title} \\ (p', \texttt{introduction}) \to \texttt{introduction} \end{array}$$

The output of the transformation, applied to the document in Figure 3 is the following tree. Here, we replaced the part of the output that is also generated by the previous transformation with dots.



□

We define some relevant classes of transducers. A transducer is *non-deleting* if no states occur at the top-level of a rhs. We denote by $\mathcal{T}_{nd}$ the class of non-deleting transducers and by $\mathcal{T}_d$ the class of transducers where we allow deletion. Further, a transducer T has *copying width* k if there are at most k occurrences of states in every sequence of siblings in the right-hand sides of rules of T. For instance, the transducer in Example 1 has copying width two. By $\mathcal{T}_{bc}$ we denote the class of transducers for which there is a natural number k such that all transducers have copying width k. We leave k implicit. We denote the intersections of these classes by combining the indexes. For instance, $\mathcal{T}_{nd,bc}$ is the class of non-deleting transducers with bounded copying. To emphasize that we allow unbounded copying, we also write $\mathcal{T}_{nd,c}$ rather than $\mathcal{T}_{nd}$.

2.4 The Typechecking Problem

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 $t \in S_{\text{in}}$.

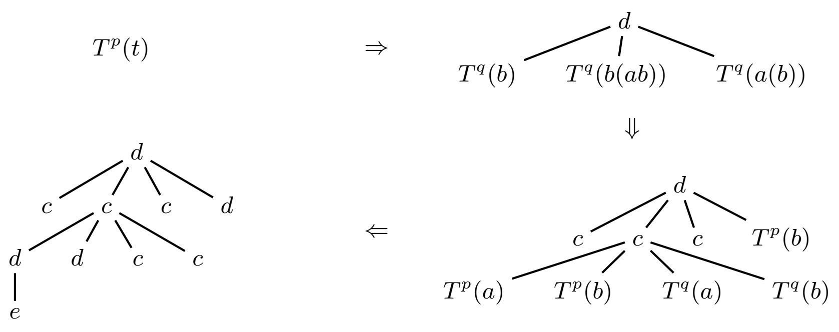


Figure 2: The translation of $t = b(b\,b(a\,b)a(b))$ by the transducer T of Example 1.

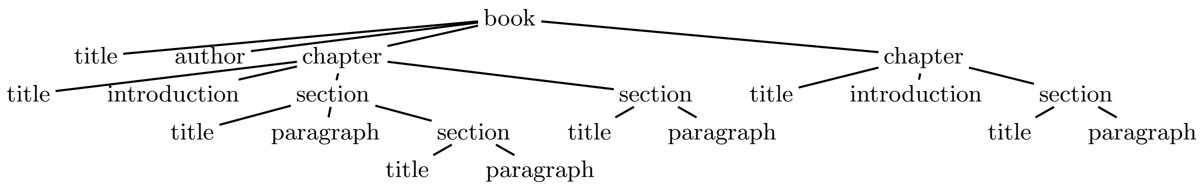


Figure 3: A document conforming to the schema of Example 3.

Example 4. The second transducer of Example 3 typechecks w.r.t. the input schema and the following DTD:

$$\texttt{book} \rightarrow \texttt{title}, (\texttt{chapter}, \texttt{title}^*)^*, \texttt{chapter}*$$
$$\texttt{chapter} \rightarrow \texttt{title}, \texttt{introduction} \mid \varepsilon$$

□

We define the problem central to this paper.

Definition 4. Given S_{in}, S_{out}, and T, the *typechecking problem* consists in verifying whether T typechecks w.r.t. S_{in} and S_{out}. □

The size of the input is the sum of the sizes of S_{in}, S_{out}, and T. 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 $S_{\text{in}}, S_{\text{out}} \in \mathcal{S}$. Examples of classes of tree languages, are those defined by tree automata or DTDs. Classes of transducers are discussed in the previous section. The complexity of the problem is measured in terms of the sum of the sizes of the input and output schemas and the transducer.

Table 1 summarizes the results obtained in [16]. All problems are complete for the mentioned complexity classes. In the setting of [16], typechecking is only tractable when restricting to non-deleting and bounded copying transducers in the presence of DTDs with DFAs. In the remainder of the paper, we obtain more general classes for which typechecking is in PTIME.

3. FIXING SCHEMA LANGUAGES

As argued in the introduction, for some scenarios it makes sense to consider the input and/or output schema not as part of the input. From a complexity theory point of view, it is important to note here that the input and/or output alphabet then also becomes fixed. In this section, we revisit the results of [16] in that respect. Surprisingly, the new settings do not result in a spectacular improvement of the complexity.

The results are summarized in Table 2. We explain the notation used in the table. The second column specifies the kind of tree transducer: *d* stands for deleting, *c* for copying, *nd* for non-deleting, and *bc* for bounded copying. The leftmost column lists which schema languages are fixed. In the case of deleting transformations, the different possibilities are grouped as all complexities coincide. The remaining columns show the allowed schema languages. As some results already follow from proofs in [16], we printed the new results in bold. The entries where the complexity was lowered are underlined.

We discuss the obtained results: for non-deleting transformations, we get two new tractable cases: (1) fixed output schema, bounded copying and DTD(NFA)s; and, (2) fixed input and output, *un*bounded copying and all DTDs. It is striking, however, that in the presence of deletion or tree automata (even deterministic ones) typechecking remains EXPTIME-hard for *all* scenarios. So, the relaxed setting still disallows to combine tractability with the desirable ability to delete. We therefore focus on deletion in the next section.

Mostly, we only needed to strengthen the lower bound proofs of [16]. A particularly interesting non-trivial case, is the PSPACE lower bound of TC^o[$\mathcal{T}_{nd,c}$,DTD(DFA)]. The rest of the proofs can be found in [15].

PROPOSITION 3.1. *$TC^o[\mathcal{T}_{nd,c}, DTD(DFA)]$ is* PSPACE*-hard.*

PROOF. We use a reduction from the corridor tiling prob-

TT	NTA	DTA	DTD(NFA)	DTD(DFA)
d,c	EXPTIME	EXPTIME	EXPTIME	EXPTIME
nd,c	EXPTIME	EXPTIME	PSPACE	PSPACE
nd,bc	EXPTIME	EXPTIME	PSPACE	PTIME

Table 1: Results of [16] (upper and lower bounds).

fixed	TT	NTA	DTA	DTD(NFA)	DTD(DFA)
in, out,	d,c	**EXP**	**EXP**	**EXP**	**EXP**
in+out	d,bc	**EXP**	**EXP**	**EXP**	**EXP**
in	nd,c	**EXP**	**EXP**	**PSPACE**	**PSPACE**
	nd,bc	**EXP**	**EXP**	**PSPACE**	**PTIME**
out	nd,c	**EXP**	**EXP**	**PSPACE**	**PSPACE**
	nd,bc	**EXP**	**EXP**	**PTIME**	PTIME
in+out	nd,c	**EXP**	**EXP**	**NL**	**NL**
	nd,bc	**EXP**	**EXP**	**NL**	**NL**

Table 2: Complexities of the typechecking problem in the new setting (upper and lower bounds).

lem [7]. Let $(D, V, H, \bar{t}, \bar{b})$ be a tiling system, where $D = \{t_1, \dots, t_k\}$ is the set of tiles, $V \subseteq D^2$ and $H \subseteq D^2$ are the sets of vertical and horizontal constraints respectively, and $\bar{t}$ and $\bar{b}$ are the top and bottom row, respectively. Let n be the width of $\bar{t}$ and $\bar{b}$. The tiling system has a solution if there is an $m \in \mathbf{N}$, such that the space $m \times n$ (m rows and n columns) can be correctly tiled (w.r.t. H and V) with the additional requirement that the bottom and top row are $\bar{b}$ and $\bar{t}$, respectively.

We define the input DTD d_{in} over the alphabet $\Sigma := \{(i, t_j) \mid j \in \{1, \dots, k\}, i \in \{1, \dots, n\}\} \cup \{r\}$; r is the start symbol. Define $d_{\text{in}}(r) = \#\bar{t}\#\big(\Sigma_1 \cdot \Sigma_2 \cdots \Sigma_n\#\big)^*\bar{b}\#$, where we denote by Σ_i the set $\{(i, t_j) \mid j \in \{1, \dots, k\}\}$. Here, # functions as a row separator. For all other alphabet symbols $a \in \Sigma$, $d_{\text{in}}(a) = \varepsilon$. So, d_{in} encodes all possible tilings that start and end with the bottom row $\bar{b}$ and the top row $\bar{t}$, respectively.

We now construct a tree transducer $T = (Q_T, \Sigma, q_T^0, R_T)$ and an output DTD d_{out} such that the tiling system has no correct corridor tiling if and only if T typechecks w.r.t. d_{in} and d_{out}. Intuitively, the transducer and the output DTD have to work together to determine errors in input tilings. There can only be two types of error: two tiles do not match horizontally or two tiles do not match vertically. The main difficulty is that the output DTD is fixed and can, therefore, *not* depend on the tiling system. The transducer is constructed in such a way that it prepares in parallel the verification for all horizontal and vertical constraints by the output schema. In particular, the transducer outputs specific symbols from a fixed set independent of the tiling system allowing the fixed output schema to determine whether an error occurred.

The state set Q_T is partitioned into two sets: (1) one for the horizontal constraints: for every $i \in \{1, \dots, n-1\}$ and $t \in D$, $q_{i,t} \in Q_T$ transforms the rows in the tiling such that it is possible to check that when position i carries a t, position $i+1$ carries a t' such that $(t, t') \in H$; and, (2) one for the vertical constraints: for every $i \in \{1, \dots, n\}$ and $t \in D$, $p_{i,t} \in Q_T$ transforms the rows in the tiling such that it is possible to check that when position i carries a t, the next row carries a t' on position i such that $(t, t') \in V$.

The tree transducer T always starts its transformation with the rule $(q_T^0, r) \to r(w)$, where w is the concatenation of all of the above states, separated by the delimiter \$. The other rules are of the following form:

- Horizontal constraints: for all $(j, t) \in \Sigma$ add the rule $(q_{i,t}, (j, t')) \to \alpha$ where

$$\alpha = \begin{cases} a & \text{if } j = i \text{ and } t = t' \\ e & \text{if } j = i \text{ and } t \neq t' \\ a & \text{if } j = i+1 \text{ and } (t, t') \in H \\ b & \text{if } j = i+1 \text{ and } (t, t') \notin H \\ e & \text{if } j \neq i \text{ and } j \neq i+1 \end{cases}$$

  Finally, $(q_{i,t}, \#) \to \texttt{hor}$. The intuition is as follows: if the i-th position in a row is labeled with t, then this position is transformed into a. Position $i+1$ is transformed to a when it carries a tile that matches t horizontally. Otherwise it is transformed to b. All other symbols are transformed into an e. So, a row, delimited by two `hor`-symbols, is wrong iff there is an a immediately followed by a b. When there is no a, then position i was not labeled with t. So, the label a acts as a trigger for the output automaton.

- Vertical constraints: for all $(j, t) \in \Sigma$, add the rule $(q_{i,t}, (j, t')) \to \alpha$ where

$$\alpha = \begin{cases} a & \text{if } (j, t') = (i, t) \text{ and } (t, t) \in V \\ b & \text{if } (j, t') = (i, t) \text{ and } (t, t) \notin V \\ c & \text{if } j = i,\ t \neq t', \text{ and } (t, t') \in V \\ d & \text{if } j = i,\ t \neq t', \text{ and } (t, t') \notin V \\ e & \text{if } j \neq i \end{cases}$$

  Finally, $(q_{i,t}, \#) \to \texttt{ver}$. The intuition is as follows: if the i-th position in a row is labeled with t, then this position is transformed into a when $(t, t) \in V$ and to b when $(t, t) \notin V$. Here, both a and b act as a trigger for the output automaton: they mean that position i was labeled with t. But no a and b can occur in the same transformed row. When position i is labeled with $t' \neq t$, then we transform this position into c when $(t, t') \in V$, and in d when $(t, t') \notin V$. All other positions are transformed into e. The output DFA works as follows. If a position is labeled a then it accepts if there is a d occurring after the next `ver`. If a position is labeled b, then it accepts if there is a b or a d occurring after the next `ver`. Otherwise, it rejects that row.

By making use of the delimiters `ver` and `hor`, both above described automata can be combined into one taking care of the vertical and the horizontal constraints. Note that the output automaton is defined over the fixed alphabet $\{a, b, c, d, e, \texttt{hor}, \texttt{ver}, \$\}$. $\square$

In the remainder of the paper, we denote by $\mathrm{TC}^i[\mathcal{T}, \mathcal{S}]$, $\mathrm{TC}^o[\mathcal{T}, \mathcal{S}]$ and $\mathrm{TC}^{i/o}[\mathcal{T}, \mathcal{S}]$ the typechecking problem where the input schema, output schema and both input and output schema are fixed respectively. So, 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, minus the size of the fixed parameter(s).

4. DELETION, BOUNDED COPYING, AND DFAS

Although deletion has an enormous impact on the complexity of typechecking, as is exemplified by the first two rows of Table 2, more often than not, the ability to skip nodes in the input tree is critical. Indeed, many simple transformations like the ones in Example 3 select specific parts of the input while deleting the non interesting ones. Moreover, such deletion can be unbounded. That is, the number of deleted nodes on a path depends only on the input tree and not on the schema.

In this section, we focus on DTD(DFA)s and on bounded copying transducers. We prove a general lemma which quantifies the combined effect of copying and deletion on the complexity of typechecking. From this lemma we then derive conditions under which typechecking becomes tractable. Interestingly, these conditions allow arbitrary deletion when no copying occurs, but at the same time permit bounded copying for those rules that only delete in a bounded fashion. We further show that these conditions cannot be relaxed without increasing the complexity. Finally, we discuss typechecking in the context of schemas represented by deterministic tree automata.

4.1 A Tractable Case

We start by introducing some terminology. Let $T = (Q, \Sigma, q_0, R)$ be a transducer. A *deletion path* is a sequence of states $q_1, \ldots, q_n$ such that q_i occurs in $\mathrm{top}(\mathrm{rhs}(q_{i-1}, a_{i-1}))$ for every $i = 2, \ldots, n$, where $a_1, \ldots, a_{n-1} \in \Sigma$. A state q is *recursively deleting* if it occurs twice in some deletion path; otherwise, q is said to be *non-recursively deleting*. The *deletion width* of q is the maximum number of occurrences of states in $\mathrm{top}(\mathrm{rhs}(q, a))$ for all $a \in \Sigma$. For instance, if R contains the rules $(q, a) \to aq_1bq_2q_3$ and $(q, b) \to q_1a$, then the deletion width of q is three. The *deletion width of a deletion path* $q_1, \ldots, q_n$ is the product of the deletion widths of $q_1, \ldots, q_{n-1}$ (q_n is not counted). A deleting state q has *deletion depth* k if all deletion paths starting with q contain at most $k + 1$ states. If there exists no such k, we say that q has *infinite deletion depth*. In particular, all recursively deleting states have infinite deletion depth.

Example 5. Suppose that T is a tree transducer with states $q_1, \ldots, q_8$ and the following rewrite rules:

$$\begin{array}{ll}
(q_1, a) \to q_2\, a\, q_2\, a & (q_5, a) \to q_6\, a\, a\, q_6 \\
(q_2, a) \to a\, q_3\, q_3\, a\, q_3 & (q_6, a) \to q_7\, q_7\, q_7 \\
(q_3, a) \to q_4 & (q_7, a) \to a\, q_8\, a \\
(q_4, a) \to a & (q_8, a) \to a\, a\, q_7
\end{array}$$

The deletion depths and widths are given as follows:

state	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
deletion depth	3	2	1	0	∞	∞	∞	∞
deletion width	2	3	1	0	2	3	1	1

The sequences q_1, q_2, q_3, q_4 and q_5, q_6, q_7, q_8, q_7 are examples of deletion paths in T. Both paths have deletion width six. Note that the deletion path $q_5, q_6, q_7, q_8, q_7, q_8, q_7, q_8$ also has deletion width six. The reason is that the deletion widths of q_7 and q_8 themselves are one. Would there be a rule $(q_7, b) \to q_8q_8$ then paths of arbitrary large deletion width could be constructed. $\square$

We are now ready to define the class of transducers that is of interest to us.

Definition 5. By $\mathcal{T}_{\mathrm{trac}}^{C,K}$, we denote the class of transducers that (i) have copying bound C, and (ii) for which every deletion path has deletion width at most K. $\square$

When C and K are not important, we simply write $\mathcal{T}_{\mathrm{trac}}$ instead of $\mathcal{T}_{\mathrm{trac}}^{C,K}$.

Note that the class $\mathcal{T}_{\mathrm{trac}}^{C,K}$ allows recursive deleting, but only for those states that do not copy at the same time. Otherwise the width of deletion paths cannot be bounded. So, if a state of a $\mathcal{T}_{\mathrm{trac}}^{C,K}$ transducer is recursively deleting then every right-hand side is of the form hqg where q is a state and h and g are hedges containing no states on their top level and whose copying width is at most C. When a state is non-recursively deleting, then simultaneous copying and deleting is allowed but only in a bounded fashion. That is, every deletion path containing that state is of deletion width at most K and $\mathrm{rhs}(q, a)$ has copying width at most C.

Example 6. The first transducer in Example 3 belongs to $\mathcal{T}_{\mathrm{trac}}^{1,1}$ while the second is in $\mathcal{T}_{\mathrm{trac}}^{2,1}$. The transducer of Example 5 is in $\mathcal{T}_{\mathrm{trac}}^{3,6}$. $\square$

The next lemma provides a detailed analysis of the complexity of typechecking in terms of copying and deletion power. Its proof is a non-trivial generalization from non-deleting to deleting transducers of the reduction in [16] from $TC[\mathcal{T}_{nd,c}, DTD(DFA)]$ to emptiness of unranked tree automata, followed by an analysis of the size of the obtained automaton.

LEMMA 4.1. *The complexity of* $TC[\mathcal{T}^{C,K}_{trac}, DTD(DFA)]$ *is* $\mathcal{O}\big((|d_{in}||T|^{CK}|d_{out}|^{CK})^{\alpha}\big)$, *where* $|d_{in}|$ *and* $|d_{out}|$ *are the sizes of the input and output schema, respectively;* $|T|$ *is the size of the tree transducer* T*; and* α *is a constant.*

PROOF SKETCH. For a transducer $T = (Q_T, \Sigma, q^0_T, R_T) \in \mathcal{T}^{C,K}_{\text{trac}}$, and input and output schemas d_{in} and d_{out}, we construct a nondeterministic unranked tree automaton A accepting all counterexample trees. That is, $L(A) = \{t \in L(d_{\text{in}}) \mid T(t) \notin L(d_{\text{out}})\}$. So, $L(A) = \emptyset$ iff T typechecks w.r.t. d_{in} and d_{out}. The size of A is $\mathcal{O}((|d_{\text{in}}||T|^{CK}|d_{\text{out}}|^{CK})^{\beta})$ for some constant β. As emptiness NTAs is in PTIME, there is a constant α such that the complexity of the typechecking problem is $\mathcal{O}\big((|d_{\text{in}}||T|^{CK}|d_{\text{out}}|^{CK})^{\alpha}\big)$.

Checking whether the input of A is conform to the input schema can be done by a simple product construction of tree automata. We therefore focus on verifying whether the output of the transformation is *not* conform to the output schema. Intuitively, the tree automaton non-deterministically locates a node v in the input tree that generates a subtree

$$\sigma(a'_1(t'_1)\cdots a'_m(t'_m))$$

in the output such that $a'_1 \cdots a'_m \notin d_{\text{out}}(\sigma)$. More specifically, A simulates T on the subtree rooted at v and runs the DFA D representing $d_{\text{out}}(\sigma)$ on $a'_1 \cdots a'_n$.

Let $a(t_1 \cdots t_n)$ be the tree rooted at v and suppose that T processes v in state q. Suppose that $\text{rhs}(q, a)$ contains the subtree $\sigma(z_0 q_1 z_1 \cdots q_k z_k)$, where $z_0, \ldots, z_k \in \Sigma^*$ and $q_1, \ldots, q_k \in Q_T$. Then, A needs to simulate D on

$$z_0\ \text{top}\big(T^{q_1}(t_1)\cdots T^{q_1}(t_n)\big)\ \cdots\ \text{top}\big(T^{q_k}(t_1)\cdots T^{q_k}(t_n)\big)\ z_k$$

and accept if D rejects. Note that k is bounded by C. For each t_i, the automaton A guesses k pairs of states of D, $(p^1_{i,1}, p^1_{i,2}), \ldots, (p^k_{i,1}, p^k_{i,2})$, so that $\text{top}(T^{q_j}(t_i))$ takes D from state $p^j_{i,1}$ to state $p^j_{i,2}$. We always make sure that

1. z_0 takes D from its initial state to $p^1_{1,1}$;
2. z_k takes D from $p^k_{n,2}$ to a rejecting state;
3. for each $j = 1, \ldots, k-1$, z_j takes D from $p^j_{n,2}$ to $p^{j+1}_{1,1}$; and
4. for each $i = 1, \ldots, n-1$ and $j = 1, \ldots, k$, we have $p^j_{i,2} = p^j_{i+1,1}$.

Note that for this step, A needs to remember at most $2C$ states of D for each subtree.

The most challenging part remains: testing whether for each t_i, and $j = 1, \ldots, k$, the string $\text{top}(T^{q_j}(t_i))$ takes D from state $p^j_{i,1}$ to state $p^j_{i,2}$. We only sketch the idea. If $\text{rhs}(q_j, \sigma_i)$, where σ_i is the root of t_i, contains no deleting states, then $\text{top}(T^{q_j}(t_i))$ only depends on $\text{rhs}(q_j, \sigma_i)$ and not on t_i and we are done. When $\text{rhs}(q_j, \sigma_i)$ contains only one deleting state, then we just need to guess k new pairs $(p_{i,1}, p_{i,2})$ and proceed as before. So, for recursively deleting states that do not copy we only need to remember k pairs of states. Otherwise, when $\text{rhs}(q_j, \sigma_i)$ contains say ℓ deleting states, then we need to guess $k \cdot \ell$ pairs of states. As long as the transducer deletes, each of these requires guessing new states. As K is an upper bound for this number, CK is the maximum number of pairs that need to be remembered at all time to check whether for every i, $\text{top}(T^{q_j}(t_i))$ takes D from state $p^j_{i,1}$ to state $p^j_{i,2}$. We refer the interested reader to [15] for a full proof. □

From Lemma 4.1 the following tractability result then readily follows.

THEOREM 4.2. $TC[\mathcal{T}_{trac}, DTD(DFA)]$ *is in* PTIME.

So, not only do we obtain a PTIME algorithm, Lemma 4.1 also provides a clear view on the concrete complexity in terms of the different parameters.

Link with practice. At first sight, Lemma 4.1 seems to be bad news as C and K occur in the exponent. Nevertheless, we believe these numbers to be small in practical transformations. The good news, hidden in the definition of K, is that there is no penalty for the recursive deletion without copying that occurs in many filtering transformations. In contrast to our previous results that abandoned deletion completely, the present result shows that transformations with small C and K but arbitrary deletion without copying can still be efficiently typechecked.

4.2 Lower Bounds for Extensions

We address the question whether there are obvious extensions of $\mathcal{T}_{\text{trac}}$ for which typechecking remains tractable. First of all, we cannot allow arbitrary copying as even without deletion typechecking is PSPACE-hard (see Table 1). However, the restriction on deletion for $\mathcal{T}_{\text{trac}}$ transformations is very severe: the number of consecutive deletions is fixed in advance and does not even depend on the transducer. As a generalization, we can therefore consider the class $\mathcal{T}_{nrd}$ of *non-recursively deleting* transducers for which no transducer is recursively deleting. Note that now the length of a deletion path is bounded by the number of states in the transducer. Unfortunately, the next theorem gives little hope for a tractable typechecking algorithm for that class.

THEOREM 4.3. $TC^i[\mathcal{T}_{nrd,bc}, DTD(DFA)]$ *is* PSPACE-*hard.*

In a $\mathcal{T}_{\text{trac}}$ transducer, a recursively deleting state can not copy. A legitimate question is whether that restriction is necessary. We show that even in the case of fixed input and output schema, an increase to deletion width two for recursively deleting states results in an EXPTIME lower bound for typechecking. We denote the class where every state can have at most deletion width k by $\mathcal{T}_{dw=k}$.

THEOREM 4.4. *$TC^{i/o}[\mathcal{T}_{dw=2,bc}, DTD(DFA)]$ is* EXPTIME-*hard.*

4.3 Tree Automata

In the last part of this section, we turn to schemas defined by unranked tree automata. We show that when we fix the copying width to one, denoted by $cw = 1$, then recursively deleting of width one remains tractable in the presence of DTA(DFA)s but not when DTA(NFA)s are used. Such transformations are mild generalizations of relabelings. It is hence not surprising that the output type of a transducer in $\mathcal{T}_{dw=1,cw=1}$ can be captured by a tree automaton. The latter observation is a generalization of the corresponding result for ranked tree transducers [9] (Proposition 7.8(b)). We only have to show that the construction of the unranked tree automaton can be done in PTIME. Typechecking then reduces to containment checking of NTA(NFA)s in DTA(DFA)s. For completeness, we also mention here that typechecking is EXPTIME-hard when we extend the copying width to two.

THEOREM 4.5. *1. $TC[\mathcal{T}_{dw=1,cw=1}, DTA(DFA)]$ is* PTIME-*complete;*

2. $TC^{i}[\mathcal{T}_{dw=1,cw=1}, DTA(NFA)]$ is PSPACE-*hard; and*

3. $TC^{i/o}[\mathcal{T}_{nd,cw=2}, DTA(DFA)]$ is EXPTIME-*hard.*

5. DELETION, UNBOUNDED COPYING, AND RE$^+$

All tractable fragments of the previous setting assume a uniform bound on the copying and deletion width of a transducer. Although in practice these bounds will usually be small and Lemma 4.1 provides a detailed account of their effect, the restrictions remain somewhat artificial. In the present section we therefore investigate fragments in which there are no restrictions on the copying or deletion power of the transducer. This implies that we have to restrict schemas, e.g., by restricting the regular expressions in rules.

We consider the following regular expressions. Let RE$^+$ be the set of regular expressions of the form $\alpha_1 \cdots \alpha_k$ where every α_i is ε, a, or a^+ for some $a \in \Sigma$. An example is `title author`$^+$ `chapter`$^+$. In this section, we show that typechecking for arbitrary tree transducers w.r.t. DTD(RE$^+$) is in PTIME. We note that every DTD(RE$^+$) is either non-recursive (i.e. an a-labeled node has no a-labeled descendants) or defines the empty language. However, the tractability of typechecking remains non-trivial, as in general typechecking is already PSPACE-complete when using DTD(DFA)s only defining trees of depth one [16].

Let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a tree transducer, and denote the input and output DTD by d_{in} and d_{out}, respectively. We'll present a sketch of the proof. To this end, we introduce some terminology. For an RE$^+$ expression e and DTD d, we denote by d^e the hedge language $\{a_1(h_1)\cdots a_n(h_n) \mid a_1 \cdots a_n \in L(e)$ and for every $i = 1, \ldots, n$, $a_i(h_i) \in (d, a_i)\}$. So, if $t_1 \cdots t_n \in d^e$ then $\text{top}(t_1)\cdots\text{top}(t_n) \in L(e)$ and every t_i is a derivation tree of $(d, \text{top}(t_i))$. Recall that (d, a_i) denotes DTD d with start symbol a_i. For a state $q \in Q_T$ and an alphabet symbol $a \in \Sigma$, we say that the pair (q, a) is *reachable* if there exists a tree t in d_{in} such that T processes at least one node of t labeled with a in state q. The latter set can be computed in PTIME.

To verify that the instance typechecks, we have to check that for every reachable pair (q, a) and for every node u in rhs(q, a) that

$$\{z_0\text{top}(T^{q_1}(h))z_1 \cdots z_{k-1}\text{top}(T^{q_k}(h))z_k \mid h \in d_{\text{in}}^e\} \subseteq d_{\text{out}}(\sigma),$$

where $e = d_{\text{in}}(a)$, $z_0 q_1 z_1 \cdots q_k z_k$ is the concatenation of u's children, and σ is the label of u. In the above, for $h = t_1 \cdots t_n$, we denote by $T^q(h)$ the hedge $T^q(t_1)\cdots T^q(t_n)$.

We denote the above language occurring to the left of $\subseteq$ by $L_{q,a,u}$. Note that the latter is not necessarily regular, or even context-free. We construct an extended context-free grammar $G_{q,a,u}$ such that $L(G_{q,a,u}) \subseteq d_{\text{out}}(\sigma)$ iff $L_{q,a,u} \subseteq d_{\text{out}}(\sigma)$. More specifically, $G_{q,a,u} = (V, \Sigma, P, S)$, where $V = \{\langle p, b\rangle \mid p \in Q_T, b \in \Sigma\}$ is the set of non-terminals, Σ is the set of terminals, P is the set of production rules and S is the start symbol. Intuitively, each non-terminal $\langle p, b\rangle$ corresponds to the string language $\{\text{top}(T^p(t)) \mid t \in (d_{\text{in}}, b)\}$. It remains to define the production rules P. For the start symbol S, we have the rule

$$S \to z_0\langle q_1, e_1\rangle^{\theta_1} \cdots \langle q_1, e_n\rangle^{\theta_n} z_1 \cdots \\ \cdots z_{k-1}\langle q_k, e_1\rangle^{\theta_1} \cdots \langle q_k, e_n\rangle^{\theta_n} z_k,$$

where $d_{\text{out}}(\sigma) = e_1^{\theta_1} \cdots e_n^{\theta_n}$, every $e_i \in \Sigma$ and θ_i is either $+$ or ε. For a non-terminal $\langle p, b\rangle$ let $d_{\text{in}}(b) = b_1^{\alpha_1} \cdots b_m^{\alpha_m}$ and let $\text{top}(\text{rhs}(p, b)) = s_0 p_1 s_1 \cdots p_\ell s_\ell$. Then we add the rule

$$\langle p, b\rangle \to s_0\langle p_1, b_1\rangle^{\alpha_1} \cdots \langle p_1, b_m\rangle^{\alpha_m} s_1 \cdots \\ \cdots s_{\ell-1}\langle p_\ell, b_1\rangle^{\alpha_1} \cdots \langle p_\ell, b_m\rangle^{\alpha_m} s_\ell$$

to P. If there is no rhs(p, b) in R_T, we add $\langle p, b\rangle \to \varepsilon$ to P. Note that $G_{q,a,u}$ is an extended context-free grammar, polynomial in the size of d_{in} and T. It is easy to see that since d_{in} is not recursive, $G_{q,a,u}$ is also non-recursive.

It follows from the next lemma that $L_{q,a,u} \subseteq d_{\text{out}}(\sigma)$ iff $L(G_{q,a,u}) \subseteq d_{\text{out}}(\sigma)$. For a non-terminal $\langle p, b\rangle \in V$, we denote by $L(\langle p, b\rangle)$ the language that is generated by $(V, \Sigma, P, \langle p, b\rangle)$, i.e. the grammar $G_{q,a,u}$ with start symbol $\langle p, b\rangle$.

LEMMA 5.1. *Let e and f be RE$^+$ expressions, with $e = e_1^{\theta_1} \cdots e_n^{\theta_n}$, d a DTD(RE$^+$) and $T = (Q_T, \Sigma, \delta_T, R_T)$ a tree transducer. For $z_0, \ldots, z_k \in \Sigma^*$ and $q_1, \ldots, q_k \in Q_T$, define*

$$L_e = \{z_0 \mathit{top}(T^{q_1}(h))z_1 \cdots z_{k-1}\mathit{top}(T^{q_k}(h))z_k \mid h \in L(d^e)\},$$

$$L'_e = \{z_0 \mathit{top}(T^{q_1}(h_1))z_1 \cdots z_{k-1}\mathit{top}(T^{q_k}(h_k))z_k \mid \\ h_1, \ldots, h_k \in L(d^e)\},$$

and

$$L''_e = \{z_0 s_1 z_1 \cdots z_{k-1} s_k z_k \mid \\ s_i \in L(\langle q_i, e_1\rangle)^{\theta_1} \cdots L(\langle q_i, e_n\rangle)^{\theta_n}, i \in \{1, \ldots, k\}\}.$$

Then,

$$L_e \subseteq L(f) \Leftrightarrow L'_e \subseteq L(f) \Leftrightarrow L''_e \subseteq L(f).$$

Finally, testing whether $L(G_{q,a,u}) \subseteq L(d_{\text{out}}(\sigma))$ can then be done in PTIME using standard techniques. We have thus obtained the following theorem.

THEOREM 5.2. *$TC[\mathcal{T}_{d,c}, DTD(RE^+)]$ is in* PTIME.

The simplicity of RE$^+$-expressions seems to be the price to pay for a tractable algorithm for arbitrary transducers. Indeed, the inclusion problem for a class of regular expressions $\mathcal{C}$ can readily be reduced to typechecking with DTD($\mathcal{C}$)s. As it is shown in [17] that inclusion of obvious extensions of RE$^+$-expressions is CONP-hard, typechecking for the corresponding fragment is CONP-hard. In particular, [17] discusses expressions of the form $\alpha_1 \cdots \alpha_n$ where all α_i belong to classes (1) a or $a?$, and (2) a or a^*. By using similar techniques as in [17], it can also be shown that inclusion is CONP-hard for expressions where all α_i belong to classes (3) a or $(a_1^+ + \cdots + a_n^+)$, (4) a or $(a_1 \cdots a_n)^+$ (5) a or $(a_1 + \cdots + a_n)^+$ and (6) $(a_1 + \cdots + a_n)$ or a^+.

An interesting question is whether we can also obtain a PTIME typechecking algorithm if we allow expressions of the form α and $\alpha + \varepsilon$ where α is an RE$^+$ expression. This problem remains open. The following simple example shows why Lemma 5.1 does not hold anymore for such expressions.

Example 7. Consider the DTD d_{in} with rules $r \to a + \varepsilon$ and $a \to \varepsilon$, and let T be a tree transducer with start state q_0 and rules

$$(q_0, r) \to r(q_1\ q_2) \qquad (q_1, a) \to a \qquad (q_2, a) \to b.$$

Then $L_r = \{\varepsilon, ab\}$ and $L'_r = \{\text{top}(T^{q_1}(t_1)T^{q_2}(t_2)) \mid t_1, t_2 \in d_{\text{in}}^{a+\varepsilon}\} = \{\varepsilon, a, b, ab\}$. But $L_r \subseteq L(a^+b^+ + \varepsilon)$, while $L'_r \not\subseteq L(a^+b^+ + \varepsilon)$. □

6. XPATH EXPRESSIONS

An approach complementary to deletion, is the use of XPath expressions to skip nodes of the input tree. We only consider XPath expressions for downward navigation and therefore restrict attention to the following axes and predicates: child (/), descendant (//), wildcard (∗), disjunction (|), and filter ([ ]). We allow node tests and either the child or descendant axis in every fragment of XPath we consider. We use the following notational convention: for a sequence X of axes and predicates, we denote by XPath$\{X\}$ the XPath expressions that only use the axes and predicates in $\{X\}$. We assume that the semantics of XPath is known (see, e.g., [6]). Recall that an XPath pattern defines a function $t \times \text{Dom}(t) \to 2^{\text{Dom}(t)}$.

Let P be a set of patterns. We explain how the syntax and the semantics of transducers is extended to patterns in P. We denote the latter fragment by $\mathcal{T}^P$. Rules are now of the form $(q, a) \to h$ where $h \in \mathcal{H}_\Sigma((Q \times P) \cup Q)$. That is, state-pattern pairs $\langle q, \psi \rangle$ can now also occur at leaves. Previously, all children of the current node were processed; now, only the nodes selected by ψ starting from the current node (in document order). We denote state-pattern pairs with angled parentheses to avoid confusion in the string representation of trees. In our framework, we only use XPath expressions that start with $\cdot$, i.e. always start from the context node.

So, if T is a tree transducer, $t = a(t_1 \cdots t_n)$ and there is a rule $(q, a) \to h \in R_T$ then $T^q(t)$ is obtained from h by replacing every node u in h labeled with $\langle p, \psi \rangle$ by the hedge $T^p(t/u_1) \cdots T^p(t/u_n)$ where $\psi(t, \varepsilon) = \{u_1, \ldots, u_n\}$ and the sequence $u_1, \ldots, u_n$ occurs in document order. Here, we denote by t/u the subtree of t rooted at u. Note that the context node is always set to the root of the subtree that is to be processed by T.

Example 8. When making use of XPath expressions, we can write the first document transformation in Example 3 more succinctly as follows:

$(q, \texttt{book}) \to \texttt{book}(q)$
$(q, \texttt{chapter}) \to \texttt{chapter}\ \langle q, \cdot//\texttt{title} \rangle$
$(q, \texttt{title}) \to \texttt{title}$

□

Via a reduction to Theorem 4.2, we show that for very simple XPath expressions added to the formalism typechecking remains in PTIME.

THEOREM 6.1. *$TC[\mathcal{T}_{trac}^{XPath\{/,*\}}, DTD(DFA)]$ is in* PTIME.

PROOF. We will show that for any tree transducer $T \in \mathcal{T}_{\text{trac}}^{\text{XPath}\{/,*\}}$, we can construct an equivalent tree transducer $T' \in \mathcal{T}_{\text{trac}}$ such that size(T') is $\mathcal{O}$(size(T)) and T and T' have the same copying width and deletion path width.

Intuitively, we convert every XPath-expression x occurring in T to a DFA, which we simulate by using deleting states in T'. The simulation of such DFAs only introduces non-recursively deleting states of deleting width one.

Formally, let $T = (Q_T, \Sigma, q_T^0, R_T)$ and let X_T be the XPath expressions occurring in T. For each XPath-expression $x \in X_T$, let $A_x = (Q_x, \Sigma, \delta_x, \{q_x^I\}, \{q_x^F\})$ be the DFA representing it. According to [10], each A_x is linear in the size of x. Further, A_x is acyclic, only accepts a finite language, and all strings in $L(A_x)$ are of the same length. Without loss of generality, we assume that the sets Q_x are pairwise disjoint and disjoint from Q_T.

We construct $T' = (Q_{T'}, \Sigma, q_T^0, R_{T'})$ as follows. Its state set is $Q_T \cup \bigcup_{x \in X_T} Q_X$. For every rule $(q, a) \to h$ in R_T, and for every $\langle p, x \rangle$ occurring in h we have the following set of rules in $R_{T'}$

- $(q, a) \to h'$ where h' is the hedge obtained from h by replacing every occurrence of $\langle p, x \rangle$ by q_x^I;
- $(p_x, b) \to \delta_x(p_x, b)$ for every $p_x \in Q_x$ and $b \in \Sigma$ such that $\delta_x(p_x, b) \neq q_x^F$; and
- $(p_x, b) \to \text{rhs}(p, b)$ for every $p_x \in Q_x$ and $b \in \Sigma$ such that $\delta_x(p_x, b) = q_x^F$.

We only need to argue that the XPath expressions in T are evaluated correctly in T'. To this end, it easy to see that we only use deleting states for nodes that are skipped in the input tree by the XPath expressions, and that we continue in the correct state in Q_T in the nodes that are selected by the XPath expressions. Further, only deleting states of width one are introduced. So, $T' \in \mathcal{T}^{C,K}_{\text{trac}}$ whenever $T \in \mathcal{T}^{C,K}_{\text{trac}}$. $\square$

Although the fragment XPath$\{/, *\}$ is very limited, the next theorem shows that there is not much room for improvement. The lower bounds in the first bullet follow from a reduction from XPath containment in the presence of DTDs [21, 27]. The lower bound in the second bullet follows from a reduction from the intersection emptiness problem for DFAs over a unary alphabet.

THEOREM 6.2. *The following problems are* CONP*-hard.*

1. $TC[\mathcal{T}^{X}_{nd,bc}, DTD(DFA)]$*, for* X *among* XPath$\{/, |\}$, XPath$\{//, |\}$, XPath$\{/, []\}$ *and* XPath$\{//, []\}$*; and*
2. $TC[\mathcal{T}^{XPath\{//\}}_{trac}, DTD(DFA)]$.

We denote by $\mathcal{T}^{DFA}$ the fragment where patterns are specified by DFAs (every node that is reached in a final state is selected). When we completely disallow deletion, we still have tractability when patterns are specified by DFAs.

THEOREM 6.3. $TC[\mathcal{T}^{DFA}_{nd,bc}, DTD(DFA)]$ *is in* PTIME.

Using the link between DFAs and XPath expressions that was laid in [10], we immediately obtain that typechecking is in PTIME for $\mathcal{T}^{\text{XPath}\{/,//,*\}}_{nd,bc}$ where patterns are such that the number of wildcards occurring between two descendant axes is bounded by a constant. It remains open whether typechecking for $\mathcal{T}^{\text{XPath}\{/,//,*\}}_{nd,bc}$ is in PTIME in general.

Finally, we discuss XPath in connection with the RE$^+$ expressions of the previous section. As DFA-patterns can be rather directly simulated by deleting states, we obtain that typechecking is also in PTIME when we allow the transducer to use such expressions.

COROLLARY 6.4. $TC[\mathcal{T}^{DFA}_{d,c}, DTD(RE^+)]$ *is in* PTIME.

7. REMARKS

In practice it is relevant that typechecking algorithms can generate counterexample trees (or a description of them) for instances that it rejects. As our main upper bound theorem reduces the typechecking problem to the emptiness problem for a NTA(NFA) of polynomial size, and since it is possible to generate a description of a tree in the language of an NTA(NFA) in polynomial time, we can also generate a counterexample tree for the typechecking algorithm in polynomial time. Further, the algorithm for $TC[\mathcal{T}_{d,c}, DTD(\text{RE}^+)]$ can also be adapted to generate a description of a counterexample tree.

COROLLARY 7.1. *If an instance of* $TC[\mathcal{T}_{trac}, DTD(DFA)]$ *or* $TC[\mathcal{T}_{d,c}, DTD(RE^+)]$ *does not typecheck, we can generate a counterexample in* PTIME.

We say that an instance of the typechecking problem typechecks *almost always* iff the set $\{t \in d_{\text{in}} \mid T(t) \notin d_{\text{out}}\}$ is finite. The latter notion is introduced by Engelfriet and Maneth [12]. Since the finiteness problem of NTA(NFA) is decidable in PTIME, we have obtained the following.

COROLLARY 7.2. *Almost always typechecking of* $\mathcal{T}_{trac}$ *transducers w.r.t. DTD(DFA)s is in* PTIME.

8. CONCLUSION

We provided a rather complete overview of how the different parameters influence the complexity of the typechecking problem. As the main focus of the paper is on tractable scenarios, we did not investigate upper bounds for intractable cases.

First, we considered the complexity of typechecking in the presence of fixed input and/or output schemas. In comparison with the results in [16], fixing input and/or output schemas only lowers the complexity in the presence of DTDs and when deletion is disallowed.

In the remainder of the paper we identified several interesting practical tractable cases that can be classified depending on the strength of the schema languages. The most liberal setting is where RE$^+$ expressions suffice to define schema languages: we have PTIME typechecking for all transducers in our framework. In fact, any fragment of XPath whose patterns can be translated in polynomial time to DFAs can be added to the transformations. Sometimes, however, one needs more expressive regular expressions in schema languages. For instance, to express choice like in $(\texttt{section} + \texttt{table} + \texttt{figure})^*$. Our results show that there is still a PTIME algorithm when those expressions can be translated in PTIME to DFAs and when one can bound simultaneous copying and deletion. Interestingly, arbitrary deletion without copying *can* be allowed. As copying is usually fairly limited in the simple transformations for which XSLT is used, but unbounded deleting without copying is required for so-called filtering transformations, our result identifies a tractable fragment with potential in practice. Further, we obtained that the XPath axes / and $*$ can be added without increasing the complexity. Finally, when deterministic tree automata are required, no copying can be allowed but arbitrary deletion is permitted.

Though we left some questions open, we also showed that none of the above restrictions can be severely relaxed without rendering the typechecking problem intractable. So, for these larger classes of transformations or schema languages, it is more appropriate to develop incomplete or approximate algorithms.

In future work we will try to settle the remaining questions concerning the XPath fragments, look at how fixed input and/or output schemas influence the complexity of typechecking w.r.t. DTD(RE$^+$)s, and consider data values.

Acknowledgment

We thank Giorgio Ghelli for raising the question about the complexity of typechecking in the setting of a fixed output schema. We thank Dirk Leinders, Thomas Schwentick, and

Stijn Vansummeren for their comments on a previous version of this paper.

9. REFERENCES

[1] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003.

[2] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. *Journal of Computer and System Sciences*, 66(4):688–727, 2003.

[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] J. Clark and S. DeRose. XML Path Language (XPath). `http://www.w3.org/TR/xpath`.

[7] B. S. Chlebus. Domino-tiling games. *Journal of Computer and System Sciences*, 32(3):374–392, 1986.

[8] A. Frisch, G. Castagna, and V. Benzaken. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional Programming*, pages 51–63. ACM Press, 2003.

[9] 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.

[10] T.J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proc. 9th International Conference on Database Theory (ICDT 2003)*, pages 173–189, 2003.

[11] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

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

[13] D. Lee, M. Mani, and M. Murata. Reasoning about XML schema languages using formal language theory. Technical report, IBM Almaden Research Center, 2000. Log# 95071.

[14] 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.

[15] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations: Full version. `http://alpha.luc.ac.be/~lucp1436/pubs.html`.

[16] W. Martens and F. Neven. Typechecking top-down uniform unranked tree transducers. In *Proc. 9th International Conference on Database Theory (ICDT 2003)*, pages 64–78.

[17] W. Martens, F. Neven, and T. Schwentick. Complexity of Simple Regular Expressions with Applications to XML Schema Languages. Submitted.

[18] 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.

[19] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66–97, 2003.

[20] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.

[21] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proc. 9th International Conference on Database Theory (ICDT 2003)*, pages 315–329.

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

[23] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proc. 9th International Conference on Database Theory (ICDT 2003)*, pages 47–63. Springer, 2003.

[24] D. Suciu. Typechecking for semistructured data. In *Proceedings of the 8th Workshop on Data Bases and Programming Languages (DBPL 2001)*, 2001.

[25] D. Suciu. The XML typechecking problem. *SIGMOD Record*, 31(1):89–96, 2002.

[26] A. Tozawa. Towards static type checking for XSLT. In *Proceedings of the ACM Symposium on Document Engineering*, pages 18–27, 2001.

[27] P. T. Wood. Containment for XPath fragments under DTD constraints. In *Proc. 9th International Conference on Database Theory (ICDT 2003)*, pages 300–314, 2003.

[28] World Wide Web Consortium. XML Schema. `http://www.w3.org/XML/Schema`.