Frontiers of Tractability for Typechecking Simple XML Transformations ⋆

Wim Martens *and Frank Neven

Hasselt University and
Transnational University of Limburg,
Agoralaan, Building D
B-3590 Diepenbeek, Belgium

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 can be done 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.

Key words: XML, XSLT, tree transformations, typechecking, unranked tree transducers, complexity

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

⋆ The present paper is the full version of reference [20], which appeared in the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 2004.

* Corresponding author.
Email addresses: wim.martens@uhasselt.be (Wim Martens), frank.neven@uhasselt.be (Frank Neven).

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 [35,36].

The main goal of this paper is to determine relevant scenarios for which typechecking becomes tractable. Additionally, we also want to identify the frontier where these scenarios become intractable. Previous research has shown that the latter frontier of intractability is easily reached [2,21,27]. We therefore 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 [6] or by a top-down fragment of XSLT [4]. We abstract such transformations by unranked tree transducers [19,21]. As types we adopt the usual Document Type Definitions (DTDs) and their robust extension of regular tree languages [18,27] or, equivalently, specialized DTDs [30,31]. The latter can serve as a formal model for XML Schema [33].

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 [21]. 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. As illustrated by Example 10, 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 ignoring the non-interesting ones.

In the present work, we investigate settings of the typechecking problem imposing the same restrictions on input and output schemas. In this respect, we first investigate deletion in the setting where DTDs use deterministic finite automata (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 10), 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 increas-

ing the complexity. In particular, we show that combining a slight relaxation of the limited deletion restriction with copying the input only twice 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.

As an alternative to deletion, one can skip nodes in the input tree 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 [29,40] can easily be reduced to the typechecking problem, lower bounds establishing intractability readily follow for XPath fragments containing filters and disjunction.

The first PTIME results still rely 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 in a specific class. 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^+ (which we call RE^+ expressions), 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.

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 [16] and asks whether there exist only a finite number of counterexample trees for a given instance. We argue that the PTIME algorithms in Section 3 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 [15] and CDuce [10], 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 [27]. 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 [27]. Interestingly, typechecking of k-pebble transducers has recently been related to typechecking of compositions of macro tree transducers [16]. 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 [26,30]. 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 [30]. The transducers considered in the present paper are restricted versions of the ones studied by Maneth and Neven [19]. 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 with respect to tree automata for a fragment of top-down XSLT [37]. 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. In [22], we reconsidered the typechecking scenarios of [21] and investigated the complexity in the presence of a fixed input and/or output schema. However, the complexity of typechecking did not lower very much. In particular, typechecking remained intractable in all settings allowing deletion or using tree automata.

Organization. The remainder of the paper is organized as follows. In Section 2, we provide the necessary definitions and illustrate them with some examples. In Section 3, we consider deleting transducers. In Section 4, we study the addition of XPath to skip nodes in the input. In Section 5, we discuss DTDs with RE^+ expressions. In Section 6, we present some observations on generating counterexamples and almost always typechecking. We conclude in Section 7. For readability, complete proofs are sometimes moved to the Appendix.

2 Preliminaries

In this section we provide the necessary background on trees, automata, and tree transducers. We define the typechecking problem and discuss copying and deletion. In the following, Σ always denotes a finite alphabet.

By $\mathbb{N}$ we denote the set of natural numbers. A *string* $w = a_1 \cdots a_n$ is a finite sequence of Σ-symbols. The set of positions, or the *domain*, of w is $\mathrm{Dom}(w) = \{1, \ldots, n\}$. The length of w, denoted by $|w|$, is the number of symbols occurring in it. The label a_i of position i in w is denoted by $\mathrm{lab}^w(i)$. The size of a set S

is denoted by $|S|$.

As usual, a *nondeterministic finite automaton* (NFA) over Σ is a tuple $N = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of final states. A *run* ρ of N on a string $w \in \Sigma^*$ is a mapping from $\mathrm{Dom}(w)$ to Q such that $\rho(1) \in \delta(q, \mathrm{lab}^w(1))$ for $q \in I$, and for $i = 1, \ldots, |w| - 1$, $\rho(i+1) \in \delta(\rho(i), \mathrm{lab}^w(i+1))$. A run is *accepting* if $\rho(|w|) \in F$. A string is *accepted* if there is an accepting run. The language accepted by N is denoted by $L(N)$. The *size* of N is defined as $|Q| + |\Sigma| + \sum_{q \in Q, a \in \Sigma} |\delta(q, a)|$.

A *deterministic finite automaton* (DFA) is an NFA where *(i)* I is a singleton and *(ii)* $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$.

2.1 Trees and Hedges

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

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

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

(i) if $u = \varepsilon$, then $\mathrm{lab}^t_T(u) = a$; and,
(ii) if $u = iu'$, then $\mathrm{lab}^t_T(u) = \mathrm{lab}^{t_i}_T(u')$.

We define the *depth* of a tree t, denoted by $\mathrm{depth}(t)$, as follows: if $t = \varepsilon$, then $\mathrm{depth}(t) = 0$; and if $t = a(t_1 \cdots t_n)$, then $\mathrm{depth}(t) = \max\{\mathrm{depth}(t_i) \mid 1 \leq i \leq n\} + 1$. The *depth* of a node $u = i_1 \cdots i_n \in \mathrm{Dom}_T(t)$ is $n + 1$, where every $i_1, \ldots, i_n \in \mathbb{N}$. Hence, a tree t only consisting of a root has depth one, and the root node itself has depth one in t. In the sequel, whenever we say tree, we always mean Σ-tree. A *tree language* is a set of trees.

A *hedge* is a finite sequence of trees. Hence, the set of hedges, denoted by $\mathcal{H}_\Sigma$, equals $\mathcal{T}_\Sigma^*$. For every hedge $h \in \mathcal{H}_\Sigma$, the *set of hedge-nodes of* h, denoted by $\mathrm{Dom}_H(h)$, is the subset of $\mathbb{N}^*$ defined as follows:

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

The *label* of a node $u = iu'$ in the hedge $h = t_1 \cdots t_n$, denoted by $\mathrm{lab}_H^h(u)$, is defined as $\mathrm{lab}_H^h(u) = \mathrm{lab}_T^{t_i}(u')$. Note that the set of hedge-nodes of a hedge consisting of one tree is different from the set of tree-nodes of this tree. For instance, a hedge consisting of one tree can have $\{1, 11, 12\}$ as its set of hedge-nodes, whereas the set of tree-nodes of the tree occurring in this hedge is $\{\varepsilon, 1, 2\}$. The *depth* of the hedge $h = t_1 \cdots t_n$, denoted by $\mathrm{depth}(h)$, is defined as $\max\{\mathrm{depth}(t_i) \mid i = 1, \ldots, n\}$. For a hedge $h = t_1 \cdots t_n$, we denote by $\mathrm{top}(h)$ the string obtained by concatenating the root symbols of all t_is, that is, $\mathrm{lab}_H^{t_1}(1) \cdots \mathrm{lab}_H^{t_n}(n)$. The *depth* of a node $u = iv$ in the hedge $h = t_1 \cdots t_n$ is the depth of v in t_i.

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. We denote Dom_T and Dom_H simply by Dom, and we denote lab_T and lab_H by lab when it is understood from the context whether we are working with trees or hedges.

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, for example, the class of DFAs, NFAs, or regular expressions. For $M \in \mathcal{M}$, we denote by $L(M)$ the set of strings accepted by M. We then abstract DTDs as follows.

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 convenience of notation, we denote (d, s_d) by d and leave the start symbol s_d implicit whenever this cannot give rise to confusion. A tree t *satisfies* d if *(i)* $\mathrm{lab}^t(\varepsilon) = s_d$ and, *(ii)* 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 set of trees satisfying d.

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.

We recall the definition of non-deterministic tree automata from [5]. We refer the unfamiliar reader to [28] for a gentle introduction. Notice that we use tree automata on *unranked trees*, rather than the traditional tree automata.

Definition 2 A *nondeterministic tree automaton (NTA)* is a 4-tuple $B = (Q, \Sigma, \delta, F)$, where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^{Q^*}$ is a function 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 $\lambda : \mathrm{Dom}(t) \to Q$ 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, the criterion reduces to $\varepsilon \in \delta(\lambda(v), \mathrm{lab}^t(v))$. A run is *accepting* if 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 $a \in \Sigma$ we have that $\delta(q, a) \cap \delta(q', a) = \emptyset$. We denote the set of bottom-up deterministic NTAs by DTA. A tree automaton is *complete* when, for every $a \in \Sigma$, $\bigcup_{q \in Q} \delta(q, a) = Q^*$. We denote the set of bottom-up deterministic complete tree automata by DTA^c.

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 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.

We mention some basic results about decision problems for tree automata. The emptiness problem for NTAs asks, given an NTA B, whether $L(B) = \emptyset$. The proofs of the following results are in Appendix A.

Lemma 3 *The emptiness problem is* PTIME*-complete for DTA^c(DFA).*

The finiteness problem for an NTA B asks whether $L(B)$ is a finite set.

Proposition 4 *(1) Finiteness of NTA(NFA) is in* PTIME*.*
(2) Emptiness of NTA(NFA) is in PTIME*.*
(3) For a NTA(NFA) N, we can generate a description of some tree $t \in L(N)$ in PTIME*.*

2.3 Transducers

We adhere to transducers as a formal model for simple transformations corresponding to structural recursion [6] and a fragment of top-down XSLT. Like in [27], 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. [1]

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

Definition 5 A *tree transducer* is a tuple $T = (Q, \Sigma, q^0, R)$, where Q is a finite set of states, Σ is the input and output alphabet, $q^0 \in Q$ is the initial state, and R is a finite set of rules of the form $(q, a) \to h$, where $a \in \Sigma$, $q \in Q$, and $h \in \mathcal{H}_\Sigma(Q)$. When $q = q^0$, h is restricted to $\mathcal{T}_\Sigma(Q) \setminus Q$. Transducers are required to be deterministic: for every pair (q, a) there is at most one rule in R.

The restriction on rules with the initial state ensures that the output is always a tree rather than a hedge.

The translation defined by a tree transducer $T = (Q, \Sigma, q^0, R)$ 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, the transformation of t by T, denoted by $T(t)$, is defined as $T^{q^0}(t)$, interpreted as a tree.

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 an rhs. The *size* of T is $|Q| + |\Sigma| + \sum_{q \in Q, a \in \Sigma} |\mathrm{rhs}(q, a)|$, where $|\mathrm{rhs}(q, a)|$ denotes the number of nodes in $\mathrm{rhs}(q, a)$. In the sequel, we always use $p, p_1, p_2, \ldots$ and $q, q_1, q_2, \ldots$ to denote states.

We give an example of a tree transducer:

Example 6 Let $T = (Q, \Sigma, p, R)$ where $Q = \{p, q\}$, $\Sigma = \{a, b, c, d, e\}$, and R

[1] In general, of course, one can define transducers where the input alphabet differs from the output alphabet.

```
<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>
```

Fig. 1. The XSLT program equivalent to the transducer of Example 6.

contains the rules

$$(p, a) \to d(e) \qquad (p, b) \to d(q)$$
$$(q, a) \to c\ p \qquad (q, b) \to c(p\ q)$$

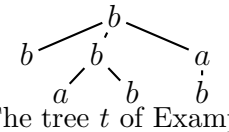
Note that the right-hand side of $(q, a) \to c\ p$ is a hedge consisting of two trees, while the other right-hand sides consist of only one tree. □

Our tree transducers can be implemented as XSLT programs in a straightforward way. For instance, the XSLT program equivalent to the above transducer is given in Figure 1 (we assume the program is started in mode p).

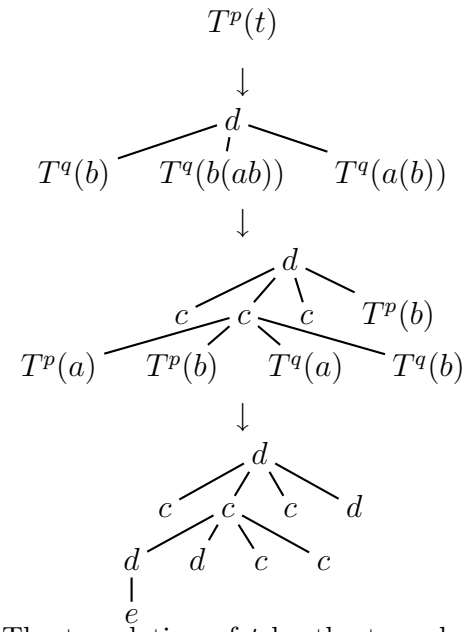
Example 7 Consider the tree t shown in Figure 2(a). In Figure 2(b) we give the translation of t by the transducer of Example 6. In order to keep the example simple, we did not list $T^q(\varepsilon)$ and $T^p(\varepsilon)$ explicitly in the process of translation. □

2.4 The Typechecking Problem

Definition 8 *A tree transducer T* typechecks *with respect to to an input tree language S_{in} and an output tree language S_{out}, if $T(t) \in S_{out}$ for every $t \in S_{in}$.*



(a) The tree t of Example 7.



(b) The translation of t by the transducer T of Example 6.

Fig. 2. A tree and its translation.

An example of a tree transducer that typechecks with respect to its input and output schema can be found in Example 11.

We define the problem central to this paper.

Definition 9 Given two tree languages S_{in}, S_{out}, and a transducer T, the *typechecking problem* consists in verifying whether T typechecks with respect to S_{in} and S_{out}.

The size of the input of the typechecking problem 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 $\text{TC}[\mathcal{T}, \mathcal{S}]$ 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 next 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.

2.5 Copying and Deletion

We discuss two important features of tree transducers: *copying* and *deletion*. In Example 6, the rule $(q, b) \to c(p\, q)$ copies the children of the current node in the input tree twice: one copy is processed in state p and the other in state

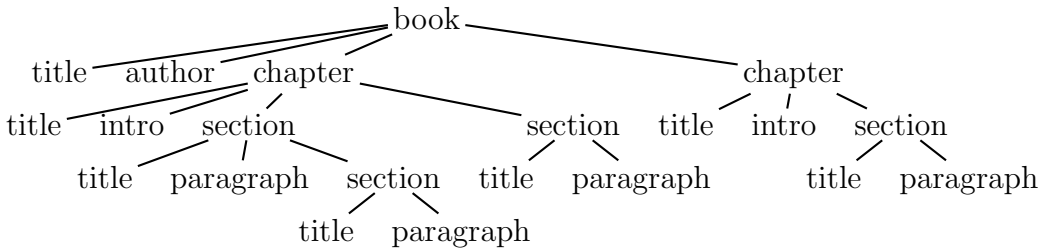


Fig. 3. A document conforming to the schema of Example 10.

q. The symbol c is the parent node of the two copies. So, one could say that the current node is translated in the new parent node labeled c. 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 node in the output tree that can be interpreted as the translation of the current node in the input tree. We therefore say that it is *deleted* and that q is a *deleting state*. For instance, $T^q(a(b)) = c\,d$ where d corresponds to b and not to a.

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

Example 10 The following DTD defines a schema for books:

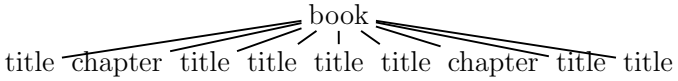
$$\begin{aligned}
\texttt{book} &\to \texttt{title}\ \texttt{author}^+\ \texttt{chapter}^+ \\
\texttt{chapter} &\to \texttt{title}\ \texttt{intro}\ \texttt{section}^+ \\
\texttt{section} &\to \texttt{title}\ \texttt{paragraph}^+\ \texttt{section}^*
\end{aligned}$$

Figure 3 depicts a document conforming to the given schema. The following transducer generates a table of contents:

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

That is, for every chapter of the book a list of its section titles is generated.

The document in Figure 3 is transformed into the tree



The example illustrates the usefulness of deleting states: all intermediate sec-

tions 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.

$$(q, \texttt{book}) \to \texttt{book}(q\ p)$$

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

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

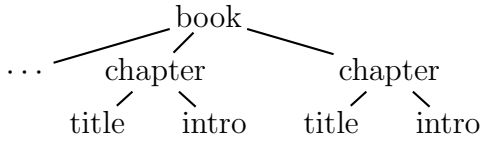
$$(q, \texttt{section}) \to q$$

$$(p, \texttt{chapter}) \to \texttt{chapter}(p')$$

$$(p', \texttt{title}) \to \texttt{title}$$

$$(p', \texttt{intro}) \to \texttt{intro}$$

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.



□

Example 11 The second transducer of Example 10 typechecks with respect to the input schema and the following DTD:

$$\texttt{book} \to \texttt{title}, (\texttt{chapter}, \texttt{title}^*)^*, \texttt{chapter}^*$$

$$\texttt{chapter} \to \texttt{title}, \texttt{intro} \mid \varepsilon$$

□

We define some relevant classes of transducers. A transducer is *non-deleting* if no rhs contains states at its top-level. 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* C if there are at most C occurrences of states in every sequence of siblings in the right-hand sides of rules of T. For

instance, the transducer in Example 6 has copying width 2. By $\mathcal{T}_{bc}^{C}$ we denote the class of transducers that have copying width C. The abbreviation "bc" stands for *bounded copying*. We will often leave the number C implicit and simply write $\mathcal{T}_{bc}$. 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 in a certain application, we write, for instance, $\mathcal{T}_{nd,uc}$ instead of $\mathcal{T}_{nd}$.

2.6 Previous Results

Table 1 summarizes the results obtained in [21]. All problems are complete for the mentioned complexity classes. In the setting of [21], 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. Note that the EXPTIME-hardness of TC[$\mathcal{T}_{nd,bc}$,DTA] was left open in [21] but settled in [22].

$\mathcal{T}$	NTA	DTA	DTD(NFA)	DTD(DFA)
d,c	EXPTIME	EXPTIME	EXPTIME	EXPTIME
d,bc	EXPTIME	EXPTIME	EXPTIME	EXPTIME
nd,c	EXPTIME	EXPTIME	PSPACE	PSPACE
nd,bc	EXPTIME	EXPTIME	PSPACE	PTIME

Table 1
A summary of the results of [21] (upper and lower bounds).

3 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 1, more often than not, the ability to skip nodes in the input tree is critical. Indeed, many simple transformations like the ones in Example 10 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.

Since the typechecking problem is immediately intractable for DTD(NFA)s and transducers with unbounded copying, we focus in this section 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 typecheck-

ing 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.

3.1 A Tractable Case

We start by introducing some terminology to describe the effect of deleting states. Let $T = (Q_T, \Sigma, q_0^T, R_T)$ be a transducer. A *deletion path* is a sequence of states $q_1, \ldots, q_n$ in Q_T such that q_i occurs in $\text{top}(\text{rhs}(q_{i-1}, a_{i-1}))$ for all $i = 2, \ldots, n$ and some $a_2, \ldots, a_n \in \Sigma$. Note that every $q_1, \ldots, q_{n-1}$ is a deleting state as defined in Section 2.5.

The *deletion width of a state* $q \in Q_T$ is the maximum number of states in $\text{top}(\text{rhs}(q, a))$ for all $a \in \Sigma$. We denote the deletion width of q by $\text{dw}(q)$. The *width of a deletion path* $q_1, \ldots, q_n$ is the product $\prod_{i=1}^{n-1} \text{dw}(q_i)$. Note that we do not take the deletion width of q_n into account as it may be zero. We say that T has *deletion path width* K if every deletion path has width smaller than or equal to K.

Example 12 Let T be the transducer consisting of the following rules:

$$
\begin{array}{ll}
(q_0^T, a) \to a(q_1\, q_5) & \\
(q_1, a) \to q_2\, a\, q_2\, a & (q_5, a) \to q_6\, a\, a\, q_6 \\
(q_2, a) \to a\, q_3\, q_3\, a\, q_3 & (q_6, a) \to q_7\, q_7 \\
(q_3, a) \to q_4 & (q_7, a) \to a\, q_8\, a \\
(q_4, a) \to a & (q_8, a) \to a\, a\, q_7
\end{array}
$$

The deletion widths of the states are given as follows:

state	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
deletion width	2	3	1	0	2	2	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. The former has deletion width six and the latter has deletion width four. 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 four. The reason is that the deletion width of q_7 and q_8 is one. Would there be a rule $(q_7, b) \to q_8 q_8$ then paths of arbitrary large deletion width could be constructed.

Notice that the deletion path width of T is six. We discuss a general algorithm to compute the deletion path width of a tree transducer in the proof of Proposition 16. □

A deleting state is *recursively deleting* if it occurs twice in some deletion path; otherwise, it is said to be *non-recursively deleting*. The *deletion depth* of a state q is the maximum length of a deletion path in which it occurs. When no such maximum exists, we say that the state has *unbounded deletion depth*. In particular, all recursively deleting states have unbounded deletion depth.

By $\mathcal{T}_{\text{trac}}^{C,K}$, we denote the class of transducers with copying width at most C and deletion path width at most K. When C and K are not important, we simply write $\mathcal{T}_{\text{trac}}$ instead of $\mathcal{T}_{\text{trac}}^{C,K}$.

Note that a class $\mathcal{T}_{\text{trac}}^{C,K}$ allows recursive deletion, 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 $\mathcal{T}_{\text{trac}}^{C,K}$-transducer is recursively deleting then every associated rhs is of the form hpg where p is a state and h and g are hedges containing no states on their top level and with at most C occurrences of states in every sequence of siblings. When a state is non-recursively deleting, then simultaneous copying and deletion is allowed but only in a bounded fashion. That is, every deletion path containing that state is of deletion width at most K.

Example 13 The first transducer in Example 10 belongs to $\mathcal{T}_{\text{trac}}^{1,1}$ while the second is in $\mathcal{T}_{\text{trac}}^{2,1}$. The transducer of Example 12 is in $\mathcal{T}_{\text{trac}}^{3,6}$.

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 [21] 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.

We use the following terminology in the proof of Lemma 14. For a tree t and a node $u \in \text{Dom}(t)$, we denote by t/u the subtree of t rooted at u. For a hedge h and a DTD (d, s_d), we say that h *partly satisfies* d if for every $u \in \text{Dom}(h)$, $\text{lab}^h(u1) \cdots \text{lab}^h(un) \in L(d(\text{lab}^h(u)))$ where u has n children. Note that there is no requirement on the root nodes of the trees in h. Hence, the term partly.

Lemma 14 *$TC[\mathcal{T}_{trac}^{C,K}, DTD(DFA)]$ can be decided in*

$$\mathcal{O}\Big((|d_{in}||T|^{C\times K}|d_{out}|^{C\times K})^{\alpha}\Big) \text{ time,}$$

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. Let $T = (Q_T, \Sigma, q_T^0, R_T) \in \mathcal{T}_{\text{trac}}^{C,K}$ be a tree transducer. Let d_{in} and d_{out} be the input and output DTDs, respectively. We construct an NTA(NFA) B such that $L(B) = \{t \in L(d_{\text{in}}) \mid T(t) \notin L(d_{\text{out}})\}$. Thus, B accepts all counterexample trees. Therefore, $L(B) = \emptyset$ if and only if T typechecks with respect to d_{in} and d_{out}. We argue that B can be constructed in time

$$\mathcal{O}((|d_{\text{in}}||T|^{C\times K}|d_{\text{out}}|^{C\times K})^{\beta})$$

for a constant β. As the emptiness problem of NTA(NFA)s is in PTIME (Proposition 4), the complexity of the typechecking problem is

$$\mathcal{O}\Big((|d_{\text{in}}||T|^{C\times K}|d_{\text{out}}|^{C\times K})^{\alpha}\Big),$$

for a constant α.

Behavior of B. A tree automaton can easily verify that the input tree satisfies d_{in}. To check that the translated tree violates the output schema d_{out}, B nondeterministically locates a node v in the input tree generating a subtree

$$\sigma(a_1(s_1)\cdots a_m(s_m))$$

such that $a_1 \cdots a_m \notin d_{\text{out}}(\sigma)$. We explain how the latter can be verified. Thereto, let $a(t_1 \cdots t_n)$ be the tree rooted at v. Assume that T processes v in state q and that $\text{rhs}(q, a)$ contains the subtree $\sigma(z_0 q_1 z_1 \cdots z_{k-1} q_k z_k)$, where $z_0, \ldots, z_k \in \Sigma^*$ and $q_1, \ldots, q_k \in Q_T$. Then, B needs to simulate the complement of the DFA D for $d_{\text{out}}(\sigma)$ on the string

$$z_0 \ \text{top}\Big(T^{q_1}(t_1)\cdots T^{q_1}(t_n)\Big)\ z_1 \cdots z_{k-1}\ \text{top}\Big(T^{q_k}(t_1)\cdots T^{q_k}(t_n)\Big)\ z_k.$$

As the strings $\text{top}(T^{q_i}(t_i))$ depend on the subtrees t_i rooted at v, B cannot simply run D. Instead, for each t_i, the automaton B guesses k pairs of states $(p_{i,1}^1, p_{i,2}^1), \ldots, (p_{i,1}^k, p_{i,2}^k)$ of D, and verifies later that indeed $\text{top}(T^{q_j}(t_i))$ takes D from state $p_{i,1}^j$ to state $p_{i,2}^j$. At present, B can only verify that

(1) z_0 takes D from its initial state to $p_{1,1}^1$;
(2) z_k takes D from $p_{n,2}^k$ to an accepting state;
(3) for each $j = 1, \ldots, k-1$, z_j takes D from $p_{n,2}^j$ to $p_{1,1}^{j+1}$; and
(4) for each $i = 1, \ldots, n-1$ and $j = 1, \ldots, k$, we have that $p_{i,2}^j = p_{i+1,1}^j$.

Note that for this step, B needs to remember at most $2C$ states of D for each subtree. We briefly sketch how B can verify that the string $\text{top}(T^{q_j}(t_i))$ takes D from state $p_{i,1}^j$ to state $p_{i,2}^j$. 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 B can simply run D. When $\text{rhs}(q_j, \sigma_i)$ contains ℓ deleting states, then we just need to guess ℓ new pairs of states $(p_{i,1}, p_{i,2})$ and defer verification to the children of the present node. As long as the transducer deletes, new pairs of states are guessed. As K is an upper bound for this number, $C \times K$ 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 $p^j_{i,2}$. Note that we also allow recursively deleting states but as these cannot copy, they do not increase the number of pairs of states B has to guess.

Construction. Let $T = (Q_T, \Sigma, q^0_T, R_T)$ and let for each $a \in \Sigma$, $A_a = (Q_a, \Sigma, \delta_a, I_a, F_a)$ be the DFA for which $d_{\text{out}}(a) = L(A_a)$. Without loss of generality, we assume that the sets Q_a are pairwise disjoint. Set $M = C \times K$. Intuitively, M is an upper bound on the number of states of some A_a that B needs to remember. This will become clear later. Next, we define the tree automaton $B = (Q_B, \Sigma, F_B, \delta_B)$. The set of states Q_B is the union of the following sets:

- Σ,
- $\{(a, q) \mid a \in \Sigma, q \in Q_T\}$,
- $\{(a, q, \text{check}) \mid a \in \Sigma, q \in Q_T\}$, and
- $\bigcup_{i=1}^{M} \{(a, (q_1, \ell^b_1, r^b_1), \ldots, (q_M, \ell^b_M, r^b_M)) \mid$
  $(q_1, \ell^b_1, r^b_1) \cdots (q_M, \ell^b_M, r^b_M) \in \{(Q_T \times Q_a \times Q_a)^i \cdot (\#, \#, \#)^{M-i} \mid i = 1, \ldots, M\}$,
  $a, b \in \Sigma\}$, where $\# \notin Q_T \cup \bigcup_{a \in \Sigma} Q_a$.

Note that the size of Q^B is $\mathcal{O}(|\Sigma||Q_T|^M |d_{\text{out}}|^{2M})$. We explain the intuition behind these states. When there is an accepting run on a tree t, then a node v labeled with a state of the form

$$a, (a, q), (a, q, \text{check}), \text{ or } (a, (q_1, \ell^b_1, r^b_1), \ldots, (q_M, \ell^b_M, r^b_M))$$

has the following meaning:

a: v is labeled with a and the subtree rooted at v partly satisfies d_{in}.

(a, 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 does not partly satisfy d_{out}.

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

$(a, (q_1, \ell^b_1, r^b_1), \ldots, (q_M, \ell^b_M, r^b_M))$: v is labeled with a and the subtree rooted at v partly satisfies d_{in}. Furthermore, v is processed by T in states $q_1, \ldots, q_j$, where j is maximal such that $q^j \neq \#$, and v is a descendant of the node labeled with (a, q, check). The triple (q_i, ℓ^b_i, r^b_i), $i \leq j$, indicates that $\text{top}(T^{q_i}(t/v))$ takes A_b from state ℓ^b_i to r^b_i.

The set of final states is $F_B := \{(s_{\text{in}}, q_T^0)\}$ where s_{in} is the start symbol of d_{in}.

The transition function is defined as follows: for all $a, b \in \Sigma$ with $a \neq b$ and $q \in Q_T$

(1) we have

$$\begin{aligned}
&\delta_B(a, b) = \emptyset;\\
&\delta_B((a, q), b) = \emptyset;\\
&\delta_B((a, q, \text{check}), b) = \emptyset; \text{ and}\\
&\delta_B((a, (q_1, \ell_1^c, r_1^c), \ldots, (q_M, \ell_M^c, r_M^c)), b) = \emptyset.
\end{aligned}$$

(2) $\delta_B(a, a) = d_{\text{in}}(a)$ and $\delta_B((a, q), a)$ consists of those strings $a_1 \cdots a_n$ such that there is precisely one index $j \in \{1, \ldots, n\}$ for which $a_j = (b, p)$ or $a_j = (b, p, \text{check})$ where p occurs in $\text{rhs}(q, a)$ and for all $i \neq j$, $a_i \in \Sigma$; further, $a_1 \cdots a_{j-1} b a_{j+1} \cdots a_n \in L(d_{\text{in}}(a))$. Note that $\delta_B((a, q), a)$ is defined in such a way that it ensures that all subtrees partly satisfy d_{in} and that at least one subtree will generate a violation of d_{out}. Clearly, $\delta_B(a, a)$ and $\delta_B((a, q), a)$ can be represented by DFAs whose size is at most quadratic in the size of the input DTD plus the size of the transducer.

(3) We define $\delta_B((a, q, \text{check}), a)$. Let u be an arbitrary node in $\text{rhs}(q, a)$ labeled with $b \in \Sigma$ and let $\overline{A_b} = (Q_b, \Sigma, \delta_b, I_b, Q_b - F_b)$. Let $s_u = z_0 q_1 z_1 \cdots z_{k-1} q_k z_k$ be the concatenation of the labels of the children of u, where every $z_i \in \Sigma^*$ and $q_i \in Q_T$. Intuitively, if v is the node in the input tree t labeled with (a, q, check), and v has n children, then we want to check here whether the string

$$\begin{aligned}
s = z_0 \text{top}(T^{q_1}(t/v1)) \cdots \text{top}(T^{q_1}(t/vn)) z_1 \cdots\\
\cdots z_{k-1} \text{top}(T^{q_k}(t/v1)) \cdots \text{top}(T^{q_k}(t/vn)) z_k
\end{aligned}$$

is accepted by $\overline{A_b}$ (or, equivalently, rejected by A_b). Of course, at v the automaton B does not know the strings $\text{top}(T^{q_j}(t/vi))$. Instead, B guesses $k \cdot n$ pairs of states $(\ell_{j,i}, r_{j,i})$ of $\overline{A_b}$, where $i = 1, \ldots, n$ and $j = 1, \ldots, k$, such that $\overline{A_b}$ accepts the string

$$\begin{aligned}
z_0 (\ell_{1,1}, r_{1,1})(\ell_{1,2}, r_{1,2}) \cdots (\ell_{1,n}, r_{1,n}) z_1 \cdots\\
\cdots z_{k-1} (\ell_{k,1}, r_{k,1})(\ell_{k,2}, r_{k,2}) \cdots (\ell_{k,n}, r_{k,n}) z_k
\end{aligned}$$

where the behavior of $\overline{A_b}$ is modified as follows: when $\overline{A_b}$ reaches $(\ell_{j,i}, r_{j,i})$ in state $\ell_{j,i}$, it moves to state $r_{j,i}$, otherwise it rejects. So, B guesses the input-output behavior $(\ell_{j,i}, r_{j,i})$ of $\overline{A_b}$ at every string $\text{top}(T^{q_j}(t/vi))$. These guesses should then be verified further down in the tree.

Formally, let for $I, F \subseteq Q_b$, $N_b(I, F) = (Q_b, \Sigma \cup (Q_b \times Q_b), \delta_{N_b}, I, F)$ be the DFA that behaves the same way as $\overline{A_b}$, but when it reads a symbol (q_1', q_2') in state q_1' it immediately jumps to state q_2', and rejects otherwise.

The parameterization of the initial and final states of N_b will be needed in bullet (4).

We define $\delta_B((a, q, \text{check}), a)$ as the union of all sets $R(u)$ where u is a node in $\text{rhs}(q, a)$ and each $R(u)$ is defined as follows:

$$\Big(a_1, (q_1, \ell^b_{1,1}, r^b_{1,1}), \ldots, (q_M, \ell^b_{M,1}, r^b_{M,1})\Big) \cdots \\ \cdots \Big(a_n, (q_1, \ell^b_{1,n}, r^b_{1,n}), \ldots, (q_M, \ell^b_{M,n}, r^b_{M,n})\Big)$$

such that

- $a_1 \cdots a_n \in d_{\text{in}}(a)$;
- the string

$$z_0(\ell^b_{1,1}, r^b_{1,1}) \cdots (\ell^b_{1,n}, r^b_{1,n}) z_1 \cdots z_{k-1} (\ell^b_{k,1}, r^b_{k,1}) \cdots (\ell^b_{k,n}, r^b_{k,n}) z_k$$

  is accepted by $N_b(I_b, Q_b - F_b)$;
- $q_1, \ldots, q_k$ are the states as occurring in $s_u = z_0 q_1 z_1 \cdots q_k z_k$; and
- for $i = k+1, \ldots, M$, $j = 1, \ldots, n$: $(q_i, \ell^b_{i,j}, r^b_{i,j}) = (\#, \#, \#)$.

We compute an upper bound for the size of the NFA representing $\delta_B((a, q, \text{check}), a)$. The alphabet size of $\delta_B((a, q, \text{check}), a)$ is bounded by $|\Sigma||Q_T|^C |Q_{\text{out}}|^{2C}$, where $Q_{\text{out}} = \bigcup_{b \in \Sigma} Q_b$. Further, for a node u in $\text{rhs}(q, a)$ labeled with b, $R(u)$ can be accepted by a DFA that simulates in parallel one copy of $d_{\text{in}}(a)$ and at most C copies of $\overline{A}_b$. Note that once u is chosen, the states $q_1, \ldots, q_k$ in $R(u)$, with $k \leq C$, are fixed. Hence, $\delta_B((a, q, \text{check}), a)$ can be represented as a union of $|\text{rhs}(q, a)|$ DFAs with $|d_{\text{in}}(a)||d_{\text{out}}|^C$ states, which bounds the total size of the NFA representing $\delta_B((a, q, \text{check}), a)$ by

$$|\Sigma||Q_T|^C |Q_{\text{out}}|^{2C} \times |\text{rhs}(q, a)||d_{\text{in}}(a)||d_{\text{out}}|^C.$$

(4) Finally, we define $\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$. Let m be the smallest index such that for all $m' > m$, $p^{m'} = \#$. Intuitively, when B arrives at a node v in state $(a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M))$, then it should verify that for every $i = 1, \ldots, m$, $\text{top}(T^{p_i}(t/v))$ takes $\overline{A_b}$ from ℓ^b_i to r^b_i. For every $i = 1, \ldots, m$, let $\text{top}(\text{rhs}(p_i, a))$ be of the form $z_{i,0} q_{i,1} z_{i,1} \cdots q_{i,k_i} z_{i,k_i}$ where $z_{i,j} \in \Sigma^*$ and $q_{i,j} \in Q_T$. When $k_i > 0$ B has to replace (p_i, ℓ^b_i, r^b_i) with a new sequence in $(Q_T \times A_b \times A_b)^*$.

So, $\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$ accepts the strings

$$\Big(a_1, (q_1, \ell^b_{1,1}, r^b_{1,1}), \ldots, (q_M, \ell^b_{M,1}, r^b_{M,1})\Big) \cdots \\ \cdots \Big(a_n, (q_1, \ell^b_{1,n}, r^b_{1,n}), \ldots, (q_M, \ell^b_{M,n}, r^b_{M,n})\Big)$$

such that

- $a_1 \cdots a_n \in d_{\text{in}}(a)$; and
- for all $i \leq m$, $q_{j+1} \cdots q_{j+k_i} = q_{i,1} \cdots q_{i,k_i}$, where $j = \Sigma_{x=1}^{i-1} k_x$; and

- for all $i \leq m$, the string

$$z_{i,0}(\ell^b_{j+1,1}, r^b_{j+1,1}) \cdots (\ell^b_{j+1,n}, r^b_{j+1,n}) z_{i,1} \cdots \\ \cdots z_{i,k_i-1}(\ell^b_{j+k_i,1}, r^b_{j+k_i,1}) \cdots (\ell^b_{j+k_i,n}, r^b_{j+k_i,n}) z_{i,k_i}$$

  is accepted by $N_b(\{\ell^b_i\}, \{r^b_i\})$, where $j = \Sigma_{x=1}^{i-1} k_x$; and,
- for $i = (1 + \Sigma_{x=1}^{m} k_x), \ldots, M$, $j = 1, \ldots, n$: $(q_i, \ell_{i,j}, r_{i,j}) = (\#, \#, \#)$.

We need to argue that at all times, $\Sigma_{x=1}^{m} k_x \leq M$. Let for an input tree t, $v \in \mathrm{Dom}(t)$ be the node that is visited in state (a, q, check) by B and let $u \in \mathrm{rhs}(q, a)$ be the node selected in step (3), labeled with b. Assume first that q is a state with bounded deletion depth. To produce the string s that must be tested for membership in $\overline{A_b}$, T visits v's children in at most C states. Let $q, q^1, \ldots, q^\ell$ be an arbitrary deletion path in T, and let for each q^i, D_i be the deletion width of q_i. Then, the nodes at depth i in t/v are visited by T in at most $C \cdot D_1 \cdots D_{i-1}$ states of T. So, every node in t/v is visited by T in at most $C \times K = M$ states to produce s. Hence, M is an upper bound for $\Sigma_{x=1}^{m} k_x$. When q has unbounded deletion depth, only states that do not copy can occur multiple times. These cannot increase the number of states B needs to remember.

We compute an upper bound for the size of

$$\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a).$$

The alphabet size of $\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$ is bounded by $|\Sigma||Q_T|^M |Q_{\mathrm{out}}|^{2M}$. Further, $\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$ simulates one copy of $d_{\mathrm{in}}(a)$ and at most M copies of $\overline{A}_b$ in parallel. Note that the sequence $q_1 \cdots q_M$ is uniquely determined by a and $p_1 \cdots p_m$. Hence,

$$\delta_B((a, (p_1, \ell^b_1, r^b_1), \ldots, (p_M, \ell^b_M, r^b_M)), a)$$

is a DFA of size

$$|\Sigma||Q_T|^M |Q_{\mathrm{out}}|^{2M} \times |d_{\mathrm{in}}(a)||A_b|^M.$$

We compute the size of B. The size of every NFA in B is $\mathcal{O}(|d_{\mathrm{in}}|^2 |Q_T|^{M+1} |d_{\mathrm{out}}|^{3M})$. Further, B has $\mathcal{O}(|\Sigma||Q_T|^M |d_{\mathrm{out}}|^{2M})$ states. Hence the size of B is

$$\mathcal{O}(|d_{\mathrm{in}}|^3 |Q_T|^{3M+1} |d_{\mathrm{out}}|^{5M}).$$

As emptiness of NTA(NFA)s is in PTIME, we get our upper bound. □

From Lemma 14 we immediately obtain that typechecking with respect to DTD(DFA)s is tractable for all classes of tree transducers with a bounded deletion path width:

Theorem 15 *$TC[\mathcal{T}_{trac}, DTD(DFA)]$ is* PTIME*-complete.*

The lower bound follows from Table 1.

Not only do we obtain a PTIME algorithm, Lemma 14 also provides a clear view on the concrete complexity in terms of the different parameters. Although the parameters C and K occur in the exponent, we believe these numbers to be small in practical transformations. It is important to point out that the presence of non-copying recursively deleting states do not affect the parameter K. Hence, 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 [21,22], the present result shows that transformations with small K but arbitrary deletion without copying can still be efficiently typechecked.

Proposition 16 *Let T be a tree transducer. The smallest numbers C and K such that $T \in \mathcal{T}_{trac}^{C,K}$ can be computed in* PTIME.

PROOF. It is obvious that C can be computed in PTIME. We only need to count the maximum number of states that occur as siblings in a rhs in T.

The computation of K is somewhat more complicated. We reduce this problem to the problem of finding a *longest path* (or a path with the *highest cost*) in a directed acyclic graph. The latter problem can be solved in polynomial time (see problem ND29 in [11]). Given a tree transducer $T = (Q_T, \Sigma, q_T^0, R_T)$, we define the *deletion path graph* $G_T = (V_T, E_T)$ — which can still contain cycles — as follows. The set of nodes $V_T = \{(q, a) \mid q \in Q_T, a \in \Sigma\}$. For a node (q, a), the set of outgoing edges is defined as $\{((q, a), (q', a')) \mid a' \in \Sigma, q' \in Q_{q,a}\}$, where $Q_{q,a}$ is the set of states occurring in $\text{top}(\text{rhs}(q, a))$. Note that these edges can be computed in PTIME. To every edge $e = ((q, a), (q', a'))$ we associate a *cost*, denoted $\text{cost}(e)$, which is the number of states occurring at $\text{top}(\text{rhs}(q, a))$. The cost of a *path* p in G_T is the product of the costs of the edges occurring in p. Note that by definition, all costs of edges are at least one and that the deletion path width of T is equal to the largest cost of a path in G_T.

We now transform G_T into an acyclic graph as follows. Assume that there is at least one edge with cost two, otherwise, we immediately know that $K = 1$. First we investigate, for every edge $e = ((q, a), (q', a'))$, if it is part of a cycle. This can be done in NLOGSPACE, and, hence, also in PTIME. If there exists an e which is part of a cycle and $\text{cost}(e) > 1$, then we can immediately halt the algorithm and conclude that K cannot be bounded. Therefore, assume now that every edge occurring in a cycle has cost one. Since cycles with cost one have no effect on the cost of the longest path in G_T, we remove these cycles from G_T by joining the nodes that they connect.

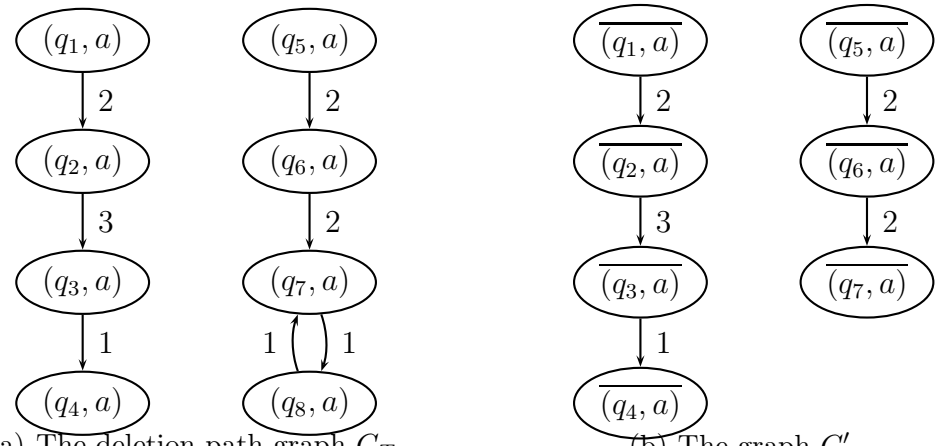
Formally, we define an equivalence relation $\equiv$ between nodes of G_T. For two nodes v and v', we say that $v \equiv v'$ if *(1)* $v = v'$; or *(2)* v and v' occur in

the same cycle (that is, there exists a directed path from v to v' and from v' to v). For a node v, we denote by $\overline{v}$ the set of nodes which are equivalent to v. We now define the graph $G'_T = (V'_T, E'_T)$, where $V'_T = \{\overline{v} \mid v \in V_T\}$ and $E'_T = \{(\overline{(q,a)}, \overline{(q',a')}) \mid ((q,a),(q',a')) \in E_T \text{ and } \overline{(q,a)} \neq \overline{(q',a')}\}$.

Since G'_T is a DAG, we can compute the longest path in G'_T in PTIME. Note that, in the longest path problem, we want to maximize the sum of the costs of the edges, whereas we want to maximize their product. However, this can directly be incorporated in the longest path algorithm, as our costs are always positive integers. It is easy to see that the maximum possible intermediate cost is always bounded by $|T|^{|T|}$. This number can be represented using $|T| \cdot \lceil \log |T| \rceil$ bits, which is polynomial. □

We illustrate how the algorithm in Proposition 16 computes C and K for the tree transducer of Example 12.

Example 17 Let T be the tree transducer defined in Example 12. It is immediate that $C = 3$. The deletion path graph $G_T = (V_T, E_T)$ is graphically represented in Figure 4(a). The graph G'_T, which is obtained from G_T by eliminating the cycles, is shown in Figure 4(b). The path $\overline{(q_1,a)}\ \overline{(q_2,a)}\ \overline{(q_3,a)}\ \overline{(q_4,a)}$ in G'_T has a cost of 6, which is the highest possible cost in G'_T.[2] Therefore, $K = 6$. □



(a) The deletion path graph G_T. (b) The graph G'_T.

Fig. 4. The deletion path width graphs G_T and G'_T of the transducer T from Example 12

3.2 Lower Bounds for Extensions

We show that the scenario of the previous section cannot be enlarged in an obvious way without rendering the typechecking problem intractable. The PTIME

[2] Recall that the cost of a path is defined to be the *product* of the costs of its edges.

result of the previous section is obtained for those classes of transducers that can bound their deletion path width and their copying width by a constant. The restriction on copying width cannot be relaxed: even TC[$\mathcal{T}_{nd,c}$,DTD(DFA)] is PSPACE-hard (cf. Table 1). What about the constraint on a bounded deletion path width? A slight relaxation of this constraint is to require that the deletion path width is finite for each transducer in the class but not necessarily bounded by a predetermined constant. We denote by $\mathcal{T}_{dw=2,cw=2,fdpw}$ the class of such transducers with the additional constraint that the deletion width and copying width of states is restricted to two. The next theorem shows that typechecking in this scenario is intractable.

Theorem 18 *$TC[\mathcal{T}_{dw=2,cw=2,fdpw}, DTD(DFA)]$ is PSPACE-hard.*

PROOF. We reduce the intersection emptiness problem of an arbitrary number of DFAs, which is PSPACE-hard [17], to the typechecking problem. The intersection emptiness problem for DFAs asks whether $\bigcap_{i=1}^{n} L(A_i) = \emptyset$ for a given sequence of DFAs $A_1, \ldots, A_n$.

For $i = 1, \ldots, n$, let $A_i = (Q_i, \Delta, \delta_i, I_i, F_i)$ be a DFA. Define $\Sigma = \Delta \cup \{\#, r, \mathrm{ok}\}$. We construct two DTD(DFA)s d_{in} and d_{out}, and a tree transducer T with deletion and copying width two, and deletion depth $\lceil \log n \rceil$, such that T typechecks with respect to d_{in} and d_{out} if and only if $\bigcap_{i=1}^{n} L(A_i) = \emptyset$.

The input DTD (d_{in}, r) is defined as follows: $d_{\mathrm{in}}(r) = \#$ and $d_{\mathrm{in}}(\#) = \# + \Delta^*$. Then, every allowed tree is of the form

where $s \in \Delta^*$. We define the tree transducer $T = (Q_T, \Sigma, q_T^0, R_T)$ where $Q_T = \{q_T^0, q^1, \ldots, q^{\lceil \log n \rceil}\}$ and R_T consists of the following rules:

- $(q_T^0, r) \to r(q^1 \# q^1)$;
- $(q^{i-1}, \#) \to q^i \# q^i$ for $i = 2, \ldots, \lceil \log n \rceil$;
- $(q^i, a) \to \mathrm{ok}$ for $i < \lceil \log n \rceil$ and $a \in \Delta$;
- $(q^{\lceil \log n \rceil}, \#) \to \mathrm{ok}$; and
- $(q^{\lceil \log n \rceil}, a) \to a$ for all $a \in \Sigma$.

Note that T produces a tree of the form $r(w)$ with $w \in (\Delta \cup \{\#, \mathrm{ok}\})^*$. When the depth of the input tree is different from $\lceil \log n \rceil$, w contains the symbol ok. Otherwise, w consists of at least n copies of the Δ-string s.

It remains to define the DFA specifying $d_{\mathrm{out}}(r)$. The automaton starts by simulating A_1. Further, when the DFA encounters the ith occurrence of a $\#$, the simulation of A_{i+1} is started. The DFA accepts when at least one A_i rejects, or when the symbol ok appears in the output.

So, for all $t \in L(d_{\mathrm{in}})$ with depth $\lceil \log n \rceil$, we have that $T(t) \in L(d_{\mathrm{out}})$ if and only if $\bigcap_{i=1}^{n} L(A_i) = \emptyset$. As for all other trees $t \in L(d_{\mathrm{in}})$ we have that $T(t) \in L(d_{\mathrm{out}})$, this instance typechecks if and only if $\bigcap_{i=1}^{n} L(A_i) = \emptyset$. □

For completeness, we also mention here that typechecking is EXPTIME-hard for deleting tree transducers with a deletion and copying width of two. This hardness even holds for a fixed input and output schema [22].

3.3 Tree Automata

In this section, we turn to schemas defined by unranked tree automata. We show that when every right hand side of a rewrite rule contains at most 1 state, recursively deleting of width one remains tractable in the presence of DTA^c(DFA)s. The latter is the class of bottom-up deterministic complete tree automata that use DFAs to represent transition functions. Such transformations are mild generalizations of relabelings and we therefore denote the class of these transducers by $\mathcal{T}_{\text{del-relab}}$. It is hence not surprising that the output type of a transducer in $\mathcal{T}_{\text{del-relab}}$ can be exactly captured by a tree automaton. The latter observation is a generalization of the corresponding result for ranked tree transducers [12] (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^c(DFA)s.

We make use of the following Lemma.

Lemma 19 *Let A be an NTA(NFA) and T be a non-deleting tree transducer for which every rhs contains at most one state. Then we can construct in polynomial time an NTA(NFA) B such that $L(B) = T(L(A))$.*

PROOF. Let $A = (Q_A, \Sigma, \delta_A, F_A)$ be an NTA(NFA) and let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a tree transducer such that every rule in R_T is of the form $(q, a) \to b(h)$, where h contains at most one state. We construct a NTA(NFA) $B =$

$(Q_B, \Sigma, \delta_B, F_B)$ such that $L(B) = T(L(A))$. Intuitively, when B processes a tree t, it guesses the tree t' such that $T(t') = t$ and verifies whether $t' \in L(A)$.

The automaton B is defined as follows:

$$Q_B = \Sigma \times Q_A \times Q_T \times \cup_{(q,a)\in Q_T\times\Sigma}\mathrm{Dom}(\mathrm{rhs}(q,a));$$

$F_B = \Sigma \times F_A \times \{q_T^0\} \times \{\varepsilon\}$. Intuitively, when $t \in L(B)$, it means that there is some tree $t' \in L(A)$ such that $T(t') = t$. If a node $v \in \mathrm{Dom}(t)$ is labeled with (a, q_A, q_T, u) in an accepting run of B, it intuitively means that there is a node v' in t' for which

- the label of v' is a;
- $\lambda(v') = q_A$ in some accepting run λ of A on t';
- v' was visited by T in state q_T; and
- v was constructed by T from node u in $\mathrm{rhs}(q_T, a)$.

Formally, for any $a \in \Sigma$, $q_A \in Q_A$ and $q_T \in Q_T$, let $t_1 = \mathrm{rhs}(q, a)$ and let $i_1 \cdots i_k \in \mathbb{N}^*$ be the unique node in t_1 labeled by a state, if it exists. Then, for every node $u \in \mathrm{Dom}(t_1)$ different from $i_1 \cdots i_{k-1}$ or $i_1 \cdots i_k$, with children $u1, \ldots, un$, we define

$$\delta_B((a, q_A, q_T, u), \mathrm{lab}^t(u)) := \{(a, q_A, q_T, u1) \cdots (a, q_A, q_T, un)\}.$$

It is trivial to construct an NFA of size n that accepts this singleton. Note that this language contains only the empty string if u is a leaf.

Denote by v the node $i_1 \cdots i_{k-1}$ and suppose that v has m children. Then, to define $\delta_B((a, q_A, q_T, v), \mathrm{lab}^{t_1}(v))$, let $D = (Q_D, Q_A, \delta_D, I_D, F_D)$ be the NFA representing $\delta(q_A, a)$ and let q'_T be the state in $\mathrm{rhs}(q_T, a)$. Then,

$$\delta_B((a, q_A, q_T, v), \mathrm{lab}^{t_1}(v))$$

is the NFA accepting the language

$$(a, q_A, q_T, v1) \cdots (a, q_A, q_T, v(i_k{-}1))L(D')(a, q_A, q_T, v(i_k{+}1)) \cdots (a, q_A, q_T, vm)$$

where D' is obtained from D by replacing every transition $\delta_D(p_1, q'_A) = \{p_2\}$ by

- the transitions $\delta_D(p_1, (c, q'_A, q'_T, \varepsilon)) = \{p_2\}$ for every $c \in \Sigma$ when $\mathrm{rhs}(q'_T, c)$ is a tree; and by
- the transitions

$$\delta_D\Big(p_1, (c, q'_A, q'_T, 1)\Big) = \{p_1^{q'_T,c,1}\},$$
$$\delta_D\Big(p_1^{q'_T,c,1}, (c, q'_A, q'_T, 2)\Big) = \{p_1^{q'_T,c,2}\}, \ldots$$
$$\ldots, \delta_D\Big(p_1^{q'_T,c,\ell-1}, (c, q'_A, q'_T, \ell)\Big) = \{p_2\}$$

when $\mathrm{rhs}(q'_T, c)$ is a hedge consisting of $\ell > 1$ trees. The states $p_1^{q'_T,c,1}, \ldots, p_1^{q'_T,c,\ell-1}$ are new states not occurring in the state set of D.

In other words, B guesses a string of children of node v' in t', continues with the simulation of T by remembering q'_T and continues with the simulation of A on t' by running D over the states of A.

So, B has $\mathcal{O}(|\Sigma||A||T|)$ states, and for each such state, the size of B's transition function is $\mathcal{O}(|\Sigma||A||T|)$. □

We are now ready to prove the following theorem.

Theorem 20 *$TC[\mathcal{T}_{del\text{-}relab}, DTA^c(DFA)]$ is* PTIME*-complete.*

PROOF. The lower bound is immediate from Lemma 3.

For the upper bound, we reduce the typechecking problem to the emptiness problem of the intersection of two NTA(NFA)s. To this end, let A_{in} and A_{out} be the input and output tree automaton, respectively.

We construct a non-deleting tree transducer T' from T by replacing every deleting state q in a rhs of T by $\#(q)$. So, T' outputs a $\#$ whenever T would process a deleting state. We assume that $\# \notin \Sigma$. We now construct an NTA(NFA) B_{in} such that $L(B_{\text{in}}) = T'(L(A_{\text{in}}))$. According to Lemma 19, B_{in} can be computed in time polynomial in the size of A_{in} and T'.

As A_{out} is a complete DTA(DFA), the complement $\overline{A_{\text{out}}}$ can easily be computed by switching the final and non-final states. Note that the size of $\overline{A_{\text{out}}}$ is linear in the size of A_{out}.

Define the $\#$-eliminating function γ as follows: $\gamma(a(h))$ is $\gamma(h)$ when $a = \#$ and $a(\gamma(h))$ otherwise; further, $\gamma(t_1 \cdots t_n) := \gamma(t_1) \cdots \gamma(t_n)$. We construct the NTA(NFA) B_{out}, such that B_{out} accepts a tree $t \in \mathcal{T}_{\Sigma \cup \{\#\}}$ if and only if $\gamma(t)$ is accepted by $\overline{A_{\text{out}}}$.

According to the proof of Theorem 11(1) in [21], we can construct B_{out} in LOGSPACE. The instance then typechecks if and only if $L(B_{\text{in}} \cap B_{\text{out}}) = \emptyset$. □

For completeness, we note that typechecking with respect to DTA(DFA)s already turns EXPTIME-hard for tree transducers with a copying width of one, and for which the right-hand sides of rewrite rules are allowed to contain at most two states [22]. In the reduction, both the input and output schemas are fixed.

4 XPath Patterns

An approach complementary to deletion, is the use of XPath patterns to skip nodes of the input tree [7]. As XPath patterns are very likely to occur in practical transformations, it is important to study the complexity of the type-checking problem for tree transducers that allow the use of XPath patterns. We only consider XPath patterns for downward navigation and therefore restrict attention to the following axes and operations: child ($/$), descendant ($//$), wildcard ($*$), disjunction ($|$), and filter ($[\ ]$). We allow element tests and either the child or descendant axis in every fragment of XPath we consider.

Definition 21 *An* XPath$\{/, //, [\ ], |, *\}$ pattern P *is an expression* $\cdot/\phi$ *or* $\cdot//\phi$ *where* ϕ *is defined by the following grammar:*

$$\begin{aligned}
\phi := \quad & \phi_1|\phi_2 && \textit{(disjunction)}\\
& \mid \phi_1/\phi_2 && \textit{(child)}\\
& \mid \phi_1//\phi_2 && \textit{(descendant)}\\
& \mid \phi_1[P] && \textit{(filter)}\\
& \mid a && \textit{(element test)}\\
& \mid * && \textit{(wildcard)}
\end{aligned}$$

An example of an XPath$\{/, //, [\ ], |, *\}$ pattern is $\cdot/(a|b)//c[\cdot//e]/*$.

Note that in our framework, we only use XPath patterns that start with $\cdot$, that is, always start from the context node. We use the following notational convention: for a sequence X of axes and operations, we denote by XPath$\{X\}$ the XPath patterns that only use the axes and operations in $\{X\}$. For instance, XPath$\{/, |\}$ denotes the fragment of XPath$\{/, //, [\ ], |, *\}$ where only element test and the child and disjunction axes are used. An XPath pattern P defines a function $f_P : t \times \mathrm{Dom}(t) \to 2^{\mathrm{Dom}(t)}$. We inductively define f_P as follows.

- $f_{\cdot/\phi}(t,u) := \{v \mid \exists z \in \mathbb{N} : v \in f_\phi(t,uz)\}$;
- $f_{\cdot//\phi}(t,u) := \{v \mid \exists z \in \mathbb{N}^* - \{\varepsilon\} : v \in f_\phi(t,uz)\}$;
- $f_{\phi_1|\phi_2}(t,u) := f_{\phi_1}(t,u) \cup f_{\phi_2}(t,u)$;
- $f_{\phi_1/\phi_2}(t,u) := \{v \mid \exists w \in \mathrm{Dom}(t), z \in \mathbb{N} : w \in f_{\phi_1}(t,u) \text{ and } v \in f_{\phi_2}(t,wz)\}$;
- $f_{\phi_1//\phi_2}(t,u) := \{v \mid \exists w \in \mathrm{Dom}(t), z \in \mathbb{N}^* - \{\varepsilon\} : w \in f_{\phi_1}(t,u) \text{ and } v \in f_{\phi_2}(t,wz)\}$;
- $f_{\phi_1[P]}(t,u) := \{v \mid v \in f_{\phi_1}(t,u) \text{ and } f_P(t,v) \neq \emptyset\}$;
- $f_a(t,u) := \begin{cases} \{u\} & \text{if } \mathrm{lab}^t(u) = a;\\ \emptyset & \text{otherwise;} \end{cases}$
- $f_*(t,u) := \{u\}$;

When a node u is in $f_P(t, \varepsilon)$, we say that P *selects* u in t.

Let $\mathcal{P} \subseteq \text{XPath}\{/, //, [\ ], |, *\}$ be a set of XPath patterns. We explain how the syntax and the semantics of transducers is extended to patterns in $\mathcal{P}$. We denote the latter fragment by $\mathcal{T}^{\mathcal{P}}$. Rules are now of the form $(q, a) \to h$ where $h \in \mathcal{H}_\Sigma((Q \times \mathcal{P}) \cup Q)$. That is, state-pattern pairs $\langle q, P\rangle$ can now also occur at leaves. Previously, all children of the current node were processed; now, only the nodes selected by P starting from the current node. These nodes are processed in document order, that is, the order in which they would occur in a depth-first left to right traversal of the tree. We denote state-pattern pairs with angled parentheses to avoid confusion in the string representation of trees.

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 $\langle p, P\rangle$ by the hedge $T^p(t/u_1) \cdots T^p(t/u_m)$ where $f_P(t, \varepsilon) = \{u_1, \ldots, u_m\}$ and the sequence $u_1, \ldots, u_m$ occurs in document order. Recall that 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 and that every XPath pattern is of the form $\cdot/\phi$ or $\cdot//\phi$. In this way, the context node itself is never selected and the transformation by T always terminates.

Example 22 When making use of XPath patterns, we can write the first document transformation in Example 10 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 15, we show that for very simple XPath patterns added to the formalism typechecking remains in PTIME.

Theorem 23 *$TC[\mathcal{T}_{trac}^{XPath\{/,*\}}, DTD(DFA)]$ is* PTIME*-complete.*

PROOF. The lower bound is immediate from Theorem 15. We prove the upper bound. In particular, we will show that for any tree transducer $T \in \mathcal{T}_{\text{trac}}^{\text{XPath}\{/,*\}}$, we can construct an equivalent tree transducer $T' \in \mathcal{T}_{\text{trac}}$ which has size polynomial in the size of T and the same copying and deletion path width as T. Intuitively, we convert every XPath-pattern P 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, hence, unaffecting the deletion path width.

Formally, let $T = (Q_T, \Sigma, q_T^0, R_T)$ and let $\mathcal{P}_T$ be the set of XPath patterns occurring in T. For every XPath-pattern $P \in \mathcal{P}_T$, we can easily construct a DFA $A_P = (Q_P, \Sigma, \delta_P, \{q_P^I\}, \{q_P^F\})$ accepting all strings $a_1 \cdots a_n$ such that P selects the a_n-labeled node in the tree $r(a_1(\cdots(a_n)))$ when evaluated from the root. Moreover, each A_P has a linear number of states in the number of symbols of P and at most a quadratic number of transitions. Further, A_P is acyclic, only accepts a finite language, and all strings in $L(A_P)$ are of the same length. Without loss of generality, we assume that the sets Q_P 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 $Q_T \cup \bigcup_{P \in \mathcal{P}_T}(Q_T \times Q_P)$. For every rule $(q, a) \to h$ in R_T, and for every $\langle p, P\rangle$ 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 $\langle p, P\rangle$ by (p, q_P^I);
- $((p, q_P), b) \to (p, \delta_P, q_P, b))$ for every $q_P \in Q_P$ and $b \in \Sigma$ such that $\delta_P(q_P, b) \neq q_P^F$; and
- $((p, q_P), b) \to \mathrm{rhs}(p, b)$ for every $q_P \in Q_P$ and $b \in \Sigma$ such that $\delta_P(q_P, b) = q_P^F$.

Note that the final state of A_P itself does not occur in the rewrite rules.

We only need to argue that the XPath patterns 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 patterns, and that we continue in the correct state in Q_T in the nodes that are selected by the XPath patterns. Further, only deleting states of width one are introduced. So, $T' \in \mathcal{T}_{\mathrm{trac}}^{C,K}$ whenever $T \in \mathcal{T}_{\mathrm{trac}}^{C,K}$. □

Although the fragment XPath$\{/, *\}$ is very limited, we show in Theorem 28 that there is not much room for improvement. The lower bounds in bullet (1) follow from a reduction from XPath containment in the presence of DTDs with DFAs [29,40]. This problem is defined as follows: given a DTD(DFA) d and XPath patterns P_1 and P_2, is it true that $f_{P_1}(t, \varepsilon) \subseteq f_{P_2}(t, \varepsilon)$ for all trees t satisfying d.

In the statements of Theorem 24 and Lemma 26, let XPath$\{X\}$ denote any fragment XPath$\{/, |\}$, XPath$\{//, |\}$, XPath$\{/, [\,]\}$ or XPath$\{//, [\,]\}$.

Theorem 24 ([29,39,40]) *XPath$\{X\}$ containment in the presence of DTD(DFA)s is co*NP*-hard.*

We note that Wood used DTDs with DFAs in his coNP-hardness proof of the inclusion problems of XPath$\{/, [\,]\}$ and XPath$\{//, [\,]\}$ [38].

We also make use of the following lemma. The proof uses the notion of *selecting literals* of an XPath pattern. Intuitively, an element test or a wildcard in an XPath pattern is a *selecting literal* if it used for selecting nodes in the document rather dan for navigation in the document. In the following definition, we denote by ℓ an arbitrary $a \in \Sigma$ or a wildcard.

- ℓ is a selecting literal in $\cdot/\phi_2$, in $\cdot//\phi_2$, in ϕ_1/ϕ_2, in $\phi_1//\phi_2$ or in $\phi_2[P]$ if it is a selecting literal in ϕ_2.
- ℓ is a selecting literal in $\phi_1|\phi_2$ if it is a selecting literal in ϕ_1 or in ϕ_2.
- ℓ is a selecting literal in ℓ.

Example 25 *We provide some examples.*

- *The selecting literals of* $\cdot//a/b/((c/d)|(b/e))$ *are labeled* d *and* e.
- *The selecting literal of* $\cdot/a[\cdot/c]//*[\cdot/(b|c)]$ *is labeled* $*$.

Lemma 26 ([25,29]) *Given a DTD(DFA)* d *and XPath*$\{X\}$ *patterns* P_1 *and* P_2*, we can construct a DTD(DFA)* d' *and XPath*$\{X\}$ *patterns* P_1' *and* P_2' *in* LOGSPACE *such that deciding whether*

$$f_{P_1}(t, \varepsilon) \subseteq f_{P_2}(t, \varepsilon) \text{ for all trees } t \text{ satisfying } d,$$

is equivalent to deciding whether for all trees t *satisfying* d'*,*

if P_1' *selects an* x_1*-labeled node in* t*, then* P_2' *selects an* x_2*-labeled node in* t.

PROOF. [Sketch] The DTD d' is identical to d, except that d' also requires that every node has a child leaf labeled with x_1 and one with x_2.

For $i = 1, 2$, pattern P_i' is constructed from P_i by replacing for every selecting literal ℓ

(a) subpatterns $/\ell[\phi_1]\cdots[\phi_n]$ by $/\ell[\phi_1]\cdots[\phi_n]/x_i$; and
(b) subpatterns $//\ell[\phi_1]\cdots[\phi_n]$ by $//\ell[\phi_1]\cdots[\phi_n]//x_i$,

where $[\phi_1]\cdots[\phi_n]$ is a (possibly empty) sequence of filter operations. □

The lower bound in bullet (2) of Theorem 28 follows from a reduction from the intersection emptiness problem for DFAs over a unary alphabet. Given an arbitrary number of DFAs $A_1, \ldots, A_n$ over alphabet $\{a\}$, this problem asks whether $\bigcap_{i=1}^{n} L(A_i) = \emptyset$. In the next lemma, we show that this problem is coNP-hard.

Lemma 27 *Intersection emptiness of an arbitrary number of DFAs over one-letter alphabet* $\{a\}$ *is co*NP*-hard.*

PROOF. We reduce the satisfiability problem for Boolean formulas in 3-CNF to the complement of the intersection emptiness problem. The technique is an adaptation of the proof of Meyer and Stockmeyer establishing that inequivalence of regular expressions over a unary alphabet is NP-hard [34].

Let $\Phi = \phi_1 \wedge \cdots \wedge \phi_k$ be a formula in 3-CNF with variables $\{x_1, \ldots, x_n\}$. Let $p_1, \ldots, p_n$ be the first n primes. Due to the Prime Number Theorem, we only need to check values up to at most n^2 for primality and we can find $p_1, \ldots, p_n$ in logarithmic space since n is given in unary notation. Intuitively, we can represent every truth assignment of Φ with a string a^r by assigning `true` to each x_i if $r \mod p_i = 0$ and `false` otherwise. We now construct a DFA A_i for each ϕ_i such that $\bigcap_{i=1}^{k} L(A_i) \neq \emptyset$ if and only if Φ is satisfiable.

We illustrate the construction of the A_i's by means of an example. Let $\phi_i = (x_1 \vee \neg x_2 \vee x_3)$ be a clause in Φ. Then $L(A_i) = (a^{p_1})^* + \overline{(a^{p_2})^*} + (a^{p_3})^*$. Hence, A_i accepts all strings that satisfy ϕ_i. Note that, since $(a^{p_j})^*$ or its complement can be easily represented by a DFA and since we only take unions of three automata, each A_i has $\mathcal{O}(n^{2 \cdot 3})$ states.

Finally, it is easy to see that a string $w \in \bigcap_{i=1}^{k} L(A_i)$ if and only if w encodes a truth assignment that satisfies Φ. □

Theorem 28 *The following problems are co*NP*-hard.*

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

(2) $TC[\mathcal{T}_{trac}^{XPath\{//\}}, DTD(DFA)]$*.*

PROOF. (1) In all four cases, we can do a reduction from the XPath$\{X\}$ containment problem in the presence of DTD(DFA)s, which is coNP-hard according to Theorem 24.

To this end, let P_1 and P_2 be two XPath$\{X\}$ patterns and let (d, s) be a DTD(DFA). We construct an instance of the typechecking problem that typechecks if and only if $P_1(t, \varepsilon) \subseteq P_2(t, \varepsilon)$ for every $t \in (d, s)$.

The input DTD (d_{in}, r) is identical to (d', s) as constructed in the proof of Lemma 26, except that r is a new alphabet symbol and $d_{\text{in}}(r) = s$. Let P_1' and P_2' be the XPath$\{X\}$ patterns as constructed in the proof of Lemma 26.

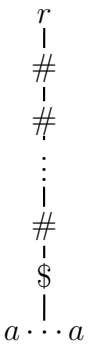
We define the tree transducer $T = (\{q_T^0, q_1\}, \Sigma, q_T^0, R_T)$. The set R_T contains the following rule:

$$(q_0, r) \to \begin{array}{ccc} & r & \\ \langle q_1, P_1' \rangle & & \langle q_1, P_2' \rangle \end{array}$$

For state q_1 we have the rules $(q_1, x_1) \to x_1$ and $(q_1, x_2) \to x_2$, which is the identity transformation on x_1 and x_2. The output DTD d_{out} has start symbol r and $d_{\text{out}}(r) = x_2^* + (x_1 x_1^* x_2 x_2^*)$. The latter checks that x_1 does not appear or appears together with x_2. The correctness now follows from the statement of Lemma 26.

(2) We reduce the intersection emptiness problem for an arbitrary number of DFAs $A_1, \ldots, A_n$, defined over alphabet $\{a\}$, which is coNP-hard (Lemma 27) to the typechecking problem.

The input DTD (d_{in}, r) is defined as follows: $d_{\text{in}}(r) = \#$, $d_{\text{in}}(\#) = \# + \$$, and $d_{\text{in}}(\$) = a^*$. So, trees are of the form



We define the tree transducer $T = (\{q_T^0, q_1, q_2, q_3\}, \{a, r, \#, \$\}, q_T^0, R_T)$ with the following rewrite rules:

$$(q_T^0, r) \to r(\langle q_1, \cdot//\# \rangle) \quad (q_1, \#) \to \langle q_2, \cdot//\$ \rangle$$
$$(q_2, \$) \to \langle q_3, \cdot//a \rangle \$ \qquad (q_3, a) \to a$$

The transducer starts by selecting every #-labeled node. For each of those (say there are k) it selects the single \$-labeled descendant node. So, k copies of the input string in $L(a^*)$ are made, separated by the \$-symbol.

The output DTD simulates the ith DFAs on the ith copy and accepts if one of them rejects or if there are less than n copies. So, the instance typechecks if the intersection is empty. Note that the copying width (C) and the deletion path width (K) are both one. □

The previous results show that to retain tractability of typechecking only very restricted XPath patterns can be added to $\mathcal{T}_{trac}$, or even $\mathcal{T}_{nd,bc}$. Next, we look at transducers where patterns are specified by DFAs (rather than by XPath patterns). We denote this fragment by $\mathcal{T}^{DFA}$. The semantics of such selecting DFAs is as follows: given a DFA A and a context node u, a descendant v of u is selected by A if and only if A accepts the string of labels on the path from u to v. From Theorem 28(2) it follows that typechecking is already hard when we allow patterns to be specified by DFAs in $\mathcal{T}_{\text{trac}}$ transducers (for instance, every XPath{//}-pattern used in the proof of Theorem 28(2) can be translated to an equivalent DFA in linear time). When we completely disallow deletion however, we still have tractability.

Theorem 29 *$TC[\mathcal{T}^{DFA}_{nd,bc}, DTD(DFA)]$ is in* PTIME.

PROOF. We show that for any tree transducer $T \in \mathcal{T}^{DFA}_{nd,bc}$, we can construct an equivalent tree transducer $T' \in \mathcal{T}_{\text{trac}}$ with size linear in the size of T, and the same copying and deletion path width as T.

The proof is quite analogous to the proof of Theorem 23. We simulate every DFA-pattern in T by deleting states in T'. The simulation of such DFAs only introduces deleting states of deletion width one.

Formally, let T be the transducer $(Q_T, \Sigma, q^0_T, R_T)$. Let $A_x = (Q_x, \Sigma, \delta_x, \{q^I_x\}, \{q^F_x\})$, $x \in X$ be the sets of selecting DFAs in T, where X is a set of indices. Without loss of generality, we assume that the sets Q_x are pairwise disjoint and disjoint from Q_T.

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

- $(q, a) \to h'$ where h' is the hedge h where every $\langle p, A_x\rangle$ is replaced by (p, q^I_x);
- $((p, q_x), b) \to (p, \delta_x(q_x, b))$ for every $p_x \in Q_x$ and $b \in \Sigma$ such that $\delta_x(p_x, b) \neq q^F_x$; and
- $((p, q_x), b) \to \text{rhs}(p, b)\ (p, q^F_x)$ for every $p_x \in Q_x$ and $b \in \Sigma$ such that $\delta_x(p_x, b) = q^F_x$. Since T is non-deleting, no states occur in $\text{top}(\text{rhs}(p, b))$ and hence, (p, q_x) has deletion path width one.

The main difference with Theorem 23 is that when we arrive in a final state of A_x, the simulation of A_x still needs to go on. This is shown in the third bullet. There, the output hedge consists of the output generated by the selection of the current node, followed by the output generated by selecting descendant nodes of the current node by A_x. Hence, the document order is respected. Again, $T' \in \mathcal{T}^{C,K}_{\text{trac}}$ whenever $T \in \mathcal{T}^{C,K}_{\text{trac}}$. □

As shown by Green et al., any XPath pattern in XPath$\{/, //, *\}$ for which the number of wildcards occurring between two descendant axes is bounded from above by c, can be translated to an equivalent DFA of size $\mathcal{O}(n^c)$, where n is the size of the pattern [13]. We hence obtain that typechecking is in PTIME for $\mathcal{T}_{nd,bc}^{\text{XPath}\{/,//,*\}}$ where patterns are such that c is bounded by a constant.

It remains open whether typechecking for $\mathcal{T}_{nd,bc}^{\text{XPath}\{/,//,*\}}$ is in PTIME in general.

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 14 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. As the typechecking problem is already PSPACE-hard when we use DTD(DFA)s, we have to restrain the schemas, for example, 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 with respect to DTD(RE$^+$)s is in PTIME. We note that every DTD(RE$^+$) is either non-recursive (that is, 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 two [21].

Notice that deciding inclusion and equivalence for RE$^+$ expressions is in PTIME, as every such expression can be transformed to a corresponding DFA in linear time. Moreover, deciding whether the intersection of an arbitrary number of RE$^+$ expressions is empty can also be decided in PTIME [23]. We further note that Benedikt, Fan, and Geerts, among other things, obtained that satisfiability of various fragments of XPath is tractable in the presence of a DTD(RE$^+$) [3].

We present the typechecking algorithm and show its correctness. For the rest of this section, 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 introduce some notational shorthands. For an RE$^+$-expression e and DTD d, we denote by d^e the hedge

language

$$\{a_1(h_1)\cdots a_n(h_n) \mid a_1\cdots a_n \in L(e) \text{ and } \forall i = 1,\ldots,n : a_i(h_i) \in L(d, a_i)\}.$$

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 $L(d_{\text{in}})$ such that T processes at least one node of t labeled with a in state q. The set of reachable pairs 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 $\text{rhs}(q, a)$ that

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

where $e = d_{\text{in}}(a)$, $z_0q_1z_1\cdots z_{k-1}q_kz_k$ 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 denoted by $T^q(h)$ the hedge $T^q(t_1)\cdots T^q(t_n)$. We denote the above language

$$\{z_0\text{top}(T^{q_1}(h))z_1\cdots z_{k-1}\text{top}(T^{q_k}(h))z_k \mid h \in d^e_{\text{in}}\}$$

by $L_{q,a,u}$. Note that the latter language is not necessarily regular, or even context-free.

We construct an extended context-free grammar $G_{q,a,u}$ such that $L(G_{q,a,u}) \subseteq d_{\text{out}}(\sigma)$ if and only if $L_{q,a,u} \subseteq d_{\text{out}}(\sigma)$. More specifically, define $G_{q,a,u} = (V, \Sigma, P, S)$, 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 $S = \langle q, a\rangle$ is the start symbol. Each non-terminal $\langle p, b\rangle$ corresponds to the string language $\{\text{top}(T^p(t)) \mid t \in L(d_{\text{in}}, b)\}$. It remains to define the production rules P. For the start symbol $\langle q, a\rangle$, we have the rule

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

where $e = e_1^{\theta_1}\cdots e_n^{\theta_n}$, every $e_i \in \Sigma$ and θ_i is either $+$ or the empty string. For a non-terminal $\langle p, b\rangle$ let $d_{\text{in}}(b) = b_1^{\alpha_1}\cdots b_m^{\alpha_m}$ and let $\text{top}(\text{rhs}(p, b)) = s_0p_1s_1\cdots p_\ell s_\ell$. Then we add the rule

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

to P. If there is no $\text{rhs}(p, b)$ 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 non-recursive, $G_{q,a,u}$ is also non-recursive and that $L_{q,a,u} \subseteq L(G_{a,q,u})$.

Our next goal is to prove the following theorem, which states that typechecking reduces to checking inclusion of the language defined by the constructed

grammar in the language defined by an RE^+ expression.

Theorem 30 *For every $q \in Q$, $a \in \Sigma$ and $u \in rhs(q,a)$,*

$$L_{q,a,u} \subseteq L(d_{out}(\sigma)) \text{ if and only if } L(G_{q,a,u}) \subseteq L(d_{out}(\sigma)),$$

where σ is the label of u.

So, typechecking reduces to testing whether $L(G_{q,a,u}) \subseteq L(d_{\text{out}}(\sigma))$. The latter can be reduced to emptiness testing of the cross-product of the pushdown automaton equivalent to $G_{q,a,u}$ and the DFA accepting the complement of $L(d_{\text{out}}(\sigma))$. All applied constructions and the emptiness test can be done PTIME [14,32].

We now prove Theorem 30 in a series of lemmas. The theorem immediately follows from Lemma 36. We fix a transducer T and an input and output schema d_{in} and d_{out}.

First, we introduce some additional notation and concepts. We bring an RE^+-expressions e in *normal form* as follows. In e, we replace every occurrence of a symbol a and a^+ by $a^{=1}$ and $a^{\geq 1}$, respectively. Next, we repeatedly combine successive terms $a^{=i}a^{=j}$ as $a^{=i+j}$, and $a^{\geq i}a^{=j}$, $a^{=i}a^{\geq j}$ or $a^{\geq i}a^{\geq j}$ as $a^{\geq i+j}$. When no combinations can be made anymore, we say that the resulting expression is *normalized*.

For a normalized RE^+-expression $e = a_1^{\theta_1 x_1} \cdots a_n^{\theta_n x_n}$, we denote by $e_{\min}$ the minimal string $a_1^{x_1} \cdots a_n^{x_n}$. A string is *vast with respect to* e, or *e-vast*, when it is of the form $a_1^{y_1} \cdots a_n^{y_n}$ where for every $i = 1, \ldots, n$, $y_i > x_i$ if θ_i is $\geq$ and $y_i = x_i$ otherwise. Note that when $L(e)$ is a singleton, the minimal string is e-vast.

We call two string languages L_1 and L_2 RE^+-equivalent, denoted $L_1 \equiv L_2$, if for every RE^+-expression e, $L_1 \subseteq L(e) \Leftrightarrow L_2 \subseteq L(e)$. Obviously, this is an equivalence relation.

Lemma 31 *For any RE^+-expression e and e-vast string e_{vast},*

$$L(e) \equiv \{e_{min}, e_{vast}\}.$$

PROOF. Let e be of the form $a_1^{\theta_1 x_1} \cdots a_n^{\theta_n x_n}$. Let f be an arbitrary RE^+-expression such that $\{e_{\min}, e_{\text{vast}}\} \subseteq L(f)$. As $e_{\min} \in L(f)$, f is of the form $a_1^{\theta'_1 y_1} \cdots a_n^{\theta'_n y_n}$, where $y_i \leq x_i$ for every $i = 1, \ldots, n$. Moreover, when θ'_i is $=$, then $y_i = x_i$. Since $e_{\text{vast}} = a_1^{z_1} \cdots a_n^{z_n} \in L(f)$, for every $i = 1, \ldots, n$, θ'_i is $\geq$ whenever $z_i > x_i$, and consequently, when θ_i is $\geq$. Therefore, $L(e) \subseteq L(f)$. Clearly, $\{e_{\min}, e_{\text{vast}}\} \subseteq L(f)$ when $L(e) \subseteq L(f)$. This proves the lemma. $\square$

Corollary 32 *Let e, f be RE$^+$-expressions. If $L(e) \not\subseteq L(f)$ then either $e_{min} \notin L(f)$ or $e_{vast} \notin L(f)$ for any e-vast string $e_{vast} \in L(e)$.*

Lemma 33 *Let e be an RE$^+$-expression and e_{vast} an e-vast string. For any $L \subseteq \Sigma^*$, if $\{e_{min}, e_{vast}\} \subseteq L \subseteq L(e)$ then $L \equiv L(e)$.*

PROOF. Let f be an arbitrary RE$^+$-expression such that $L \subseteq L(f)$. Towards a contradiction, assume that $L(e) \not\subseteq L(f)$. But then, according to Corollary 32, either $e_{\min} \notin L(f)$ or $e_{\text{vast}} \notin L(f)$. This leads to the desired contradiction. The other direction is trivial since $L \subseteq L(e)$. □

A string language L is *bounded* when there is an RE$^+$-expression $e = a_1^+ \cdots a_\ell^+$ where $a_i \neq a_{i+1}$ for each $i = 1, \ldots, \ell - 1$ such that $L \subseteq L(e)$. We refer to e as a *witness*. Two bounded languages are *bound equivalent* when they share the same witness expression. A language is *unbounded* when it is not bounded.

For every $p \in Q_T$ and $b \in \Sigma$, define $R_{p,b}$ to be the set of strings $\{\text{top}(T^p(t)) \mid t \in L(d_{\text{in}}, b)\}$. Consider the grammar $G_{q,a,u} = (V, \Sigma, P, S)$ as defined earlier in this section. Denote by $L(\langle p, b\rangle)$ the language accepted by $(V, \Sigma, P, \langle p, b\rangle)$ for every non-terminal $\langle p, b\rangle \in V$. That is, $L(\langle p, b\rangle)$ is the grammar $G_{q,a,u}$, but with start symbol $\langle p, b\rangle$. Note that, by definition of $G_{q,a,u}$, for each $p \in Q_T$, $b \in \Sigma$ we have that $R_{p,b} \subseteq L(\langle p, b\rangle)$, and in particular, $L_{q,a,u} \subseteq L(G_{q,a,u})$. Hence, the next lemma immediately follows.

Lemma 34 *(1) For every $p \in Q_T$, $b \in \Sigma$, if $L(\langle p, b\rangle)$ is bounded, then $R_{p,b}$ and $L(\langle p, b\rangle)$ are bound equivalent.*
(2) If $L(G_{q,a,u})$ is bounded, then $L(G_{q,a,u})$ and $L_{q,a,u}$ are bound equivalent.

We now show that the languages defined by the constructed grammars are bounded if and only if $R_{p,b}$ and $L_{q,a,u}$ are bounded, respectively.

Lemma 35 *(1) For every $p \in Q_T$, $b \in \Sigma$, $L(\langle p, b\rangle)$ is bounded if and only if $R_{p,b}$ is bounded.*
(2) $L(G_{q,a,u})$ is bounded if and only if $L_{q,a,u}$ is bounded.

PROOF. We only prove (1) as the proof of (2) is similar. As $G_{q,a,u} = (V, \Sigma, P, \langle q, a\rangle)$ is non-recursive, we can prove this lemma by induction on the maximum depth d of derivation trees in $(V, \Sigma, P, \langle p, b\rangle)$.

When $d = 1$, then $\langle p, b\rangle \to w$ is a rule in P for some $w \in \Sigma^*$.

By definition of $G_{q,a,u}$ and $R_{p,b}$, we then have that $L(\langle p, b\rangle) = \{w\} = R_{p,b}$. So, the statement of the lemma follows.

We turn to the induction step. Assume $d > 1$. Let

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

be a rule in P. Then, $R_{p,b}$ is the set

$$\Big\{s_0\mathrm{top}\Big(T^{p_1}(t_1)\cdots T^{p_1}(t_n)\Big)s_1\cdots s_{\ell-1}\mathrm{top}\Big(T^{p_\ell}(t_1)\cdots T^{p_\ell}(t_n)\Big)s_\ell \mid t_1\cdots t_n \in d_1^{b_1^{\alpha_1}\cdots b_m^{\alpha_m}}\Big\}.$$

The latter is equal to

$$\begin{aligned}\Big\{s_0\mathrm{top}\Big(T^{p_1}(b_1(h_1^1))\cdots T^{p_1}(b_1(h_1^{k_1}))\cdots T^{p_1}(b_m(h_m^1))\cdots T^{p_1}(b_m(h_m^{k_m}))\Big)s_1\cdots\\ \cdots s_{\ell-1}\mathrm{top}\Big(T^{p_\ell}(b_1(h_1^1))\cdots T^{p_\ell}(b_1(h_1^{k_1}))\cdots T^{p_\ell}(b_m(h_m^1))\cdots T^{p_\ell}(b_m(h_m^{k_m}))\Big)s_\ell\\ \mid b_1(h_1^1)\cdots b_1(h_1^{k_1})\cdots b_m(h_m^1)\cdots b_m(h_m^{k_m}) \in d^{b_1^{\alpha_1}\cdots b_m^{\alpha_m}}\Big\}.\end{aligned}$$

As $R_{p,b} \subseteq L(\langle p, b\rangle)$, $R_{p,b}$ is bounded when $L(\langle p, b\rangle)$ is. We next show that if $L(\langle p, b\rangle)$ is unbounded then $R_{p,b}$ is unbounded. We distinguish two cases.

(i) There is an $L(\langle p_i, b_j\rangle)$ which is unbounded. By induction,

$$R_{p_i,b_j} = \{\mathrm{top}(T^{p_i}(t)) \mid t \in d_{\mathrm{in}}^{b_j}\}$$

is unbounded. As for every string $w \in R_{p_i,b_j}$, there are strings w_1, w_2 such that $w_1 w w_2 \in R_{p,b}$, we have that the latter language is also unbounded.

(ii) Every $L(\langle p_i, b_j\rangle)$ is bounded, but there are a ℓ, m such that $L(\langle p_\ell, b_m\rangle)$ contains a string with at least two different alphabet symbols and α_m is $+$. Clearly, $L(\langle p, b\rangle)$ is unbounded. By induction, R_{p_ℓ,b_m} is bounded. By Lemma 34(1), $L(\langle p_\ell, b_m\rangle)$ and R_{p_ℓ,b_m} are bound equivalent. Therefore, since $L(\langle p_\ell, b_m\rangle)$ contains a string with at least two different alphabet symbols, every string

$$\mathrm{top}(T^{p_\ell}(b_m(h_m^1))), \ldots, \mathrm{top}(T^{p_\ell}(b_j(h_m^{k_m})))$$

contains at least two different alphabet symbols. As k_m can be arbitrarily large, $R_{p,b}$ is unbounded.

□

For every $a \in \Sigma$, we define trees $t_a^{\min}$ and t_a^{vast} in $L(d_{\mathrm{in}})$ as follows:

- when $d_{\mathrm{in}}(a) = \varepsilon$ then $t_a^{\min} = t_a^{\mathrm{vast}} = a$; and
- when $d_{\mathrm{in}}(a) = a_1^{\alpha_1}\cdots a_n^{\alpha_n}$ then

(i) $t_a^{\min} = a(t_{a_1}^{\min}\cdots t_{a_n}^{\min})$ and

(ii) $t_a^{\mathrm{vast}} = a(h_{a_1}\cdots h_{a_n})$, where for every $i = 1, \ldots, n$ we have
  - $h_{a_i} = t_{a_i}^{\mathrm{vast}} t_{a_i}^{\mathrm{vast}}$ when α_i is $+$; and

· $h_{a_i} = t_{a_i}^{\text{vast}}$, otherwise.

Theorem 30 now follows from Lemma 36(2).

Lemma 36 *(1) For every $p \in Q_T$, $b \in \Sigma$, $L(\langle p, b\rangle) \equiv R_{p,b}$; and*
(2) $L(G_{q,a,u}) \equiv L_{q,a,u}$.

PROOF. As $G_{q,a,u}$ is non-recursive, we can prove this lemma by induction on the maximum depth d of the derivation trees of $(V, \Sigma, P, \langle p, b\rangle)$.

We prove by induction on d that for any $p \in Q_T$, $b \in \Sigma$,

(IH) if $R_{p,b}$ is bounded, then there is an RE$^+$-expression $r_{p,b}$ such that
(1) $L(\langle p, b\rangle) \subseteq L(r_{p,b})$;
(2) $(r_{p,b})_{\min} = \text{top}(T^p(t_b^{\min})))$ and $\text{top}(T^p(t_b^{\text{vast}}))$ is $r_{p,b}$-vast. (Note that the latter strings are in $R_{p,b}$.)

We argue that the lemma is proven when (IH) holds. Indeed, by Lemma 33 we have that $L(\langle p, b\rangle) \equiv L(r_{p,b}) \equiv R_{p,b}$. When $R_{p,b}$ is unbounded, then so is $L(\langle p, b\rangle)$ (Lemma 35(1)). By definition, $L(\langle p, b\rangle) \equiv \langle p, d_{\text{in}}, b\rangle$.

Suppose that $d = 1$, then $\langle p, b\rangle \to w$ for some $w \in \Sigma^*$.

By definition, $L(\langle p, b\rangle) = \{w\} = R_{p,b}$. Define $r_{p,b} = w = r_{p,b}^{\min} = r_{p,b}^{\text{vast}}$. IH now holds.

We turn to the induction step. Assume $d > 1$. Let

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

be a rule in P. Then $R_{p,b} =$

$$\{s_0\text{top}(T^{p_1}(t_1) \cdots T^{p_1}(t_n))s_1 \cdots s_{\ell-1}\text{top}(T^{p_\ell}(t_1) \cdots T^{p_\ell}(t_n))s_\ell \mid t_1 \cdots t_n \in d^{b_1^{\alpha_1} \cdots b_m^{\alpha_m}}\}.$$

Assume $R_{p,b}$ is bounded. The latter implies that $L(\langle p, b\rangle)$ is bounded (Lemma 35). As the maximum depth of the derivation trees rooted at each $\langle p_i, b_j\rangle$ is $d - 1$, there are corresponding RE$^+$-expressions r_{p_i,b_j} for which the induction hypothesis holds.

Define the RE$^+$-expression $r'_{p,b}$ as

$$s_0(r_{p_1,b_1})^{\alpha_1} \cdots (r_{p_1,b_m})^{\alpha_m} s_1 \cdots s_{\ell-1}(r_{p_\ell,b_1})^{\alpha_1} \cdots (r_{p_\ell,b_m})^{\alpha_m} s_\ell.$$

We now construct $r_{p,b}$ from $r'_{p,b}$ as follows. For any $i = 1, \ldots, \ell$, $j = 1, \ldots, n$, if α_j is $+$ and $r_{p_i,b_j} = c^{=m}$ or $r_{p_i,b_j} \equiv c^{\geq m}$ then replace $(r_{p_i,b_j})^+$ by $c^{\geq m}$. Finally,

normalize the resulting expression. Note that no r_{p_i,b_j} can contain two different alphabet symbols as $L(\langle p_j, b_i\rangle)$ is bounded.

From the construction and the induction hypothesis it follows that $L(\langle p, b\rangle) \subseteq L(r'_{p,b}) \subseteq L(r_{p,b})$, so (1) holds.

It remains to show (2). Clearly,

$$r^{\min}_{p,b} = s_0(r_{p_1,b_1})_{\min} \cdots (r_{p_1,b_m})_{\min} s_1 \cdots s_{\ell-1}(r_{p_\ell,b_1})_{\min} \cdots (r_{p_\ell,b_m})_{\min} s_\ell.$$

Now define

$$(r_{p,b})_{\text{vast}} = s_0(r_{p_1,b_1})^{x_1}_{\text{vast}} \cdots (r_{p_1,b_m})^{x_m}_{\text{vast}} s_1 \cdots s_{\ell-1}(r_{p_\ell,b_1})^{x_1}_{\text{vast}} \cdots (r_{p_\ell,b_m})^{x_m}_{\text{vast}} s_\ell,$$

where for every i we have that $x_i = 1$ if α_i is ε and $x_i = 2$ otherwise. Note that the string $(r_{p,b})_{\text{vast}}$ is $r_{p,b}$-vast.

It remains to show that $(r_{p,b})_{\min} = \text{top}(T^p(t^{\min}_b))$ and $(r_{p,b})_{\text{vast}} = \text{top}(T^p(t^{\text{vast}}_b))$. By induction, $(r_{p,b})_{\min}$

$$\begin{aligned}
&= s_0 \text{top}(T^{p_1}(t^{\min}_{b_1}) \cdots T^{p_1}(t^{\min}_{b_m})) s_1 \cdots s_{\ell-1} \text{top}(T^{p_\ell}(t^{\min}_{b_1}) \cdots T^{p_\ell}(t^{\min}_{b_m})) s_\ell \\
&= s_0 \text{top}(T^{p_1}(t^{\min}_{b_1} \cdots t^{\min}_{b_m})) s_1 \cdots s_{\ell-1} \text{top}(T^{p_\ell}(t^{\min}_{b_1} \cdots t^{\min}_{b_m})) s_\ell \\
&= \text{top}(T^p(t^{\min}_b))
\end{aligned}$$

and we analogously have that $(r_{p,b})_{\text{vast}}$

$$\begin{aligned}
&= s_0 \text{top}(T^{p_1}(t^{\text{vast}}_{b_1}))^{x_1} \cdots \text{top}(T^{p_1}(h^{\text{vast}}_{b_m}))^{x_m} s_1 \cdots \\
&\qquad \cdots s_{\ell-1} \text{top}(T^{p_\ell}(h^{\text{vast}}_{b_1}))^{x_1} \cdots \text{top}(T^{p_\ell}(h^{\text{vast}}_{b_m}))^{x_m} s_\ell \\
&= s_0 \text{top}(T^{p_1}(h^{\text{vast}}_{b_1}) \cdots T^{p_1}(h^{\text{vast}}_{b_m})) s_1 \cdots s_{\ell-1} \text{top}(T^{p_\ell}(h^{\text{vast}}_{b_1}) \cdots T^{p_\ell}(h^{\text{vast}}_{b_m})) s_\ell \\
&= s_0 \text{top}(T^{p_1}(h^{\text{vast}}_{b_1} \cdots h^{\text{vast}}_{b_m})) s_1 \cdots s_{\ell-1} \text{top}(T^{p_\ell}(h^{\text{vast}}_{b_1} \cdots h^{\text{vast}}_{b_m})) s_\ell \\
&= \text{top}(T^p(t^{\text{vast}}_b))
\end{aligned}$$

where $h^{\text{vast}}_{b_i} = t^{\text{vast}}_{b_i} t^{\text{vast}}_{b_i}$ when α_i is $+$ and $h^{\text{vast}}_{b_i} = t^{\text{vast}}_{b_i}$ otherwise. □

We have thus obtained the following Theorem:

Theorem 37 *$TC[\mathcal{T}_{d,c}, DTD(RE^+)]$ 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 [23] that inclusion of obvious extensions of RE$^+$-expressions is coNP-hard, typechecking for the corresponding fragment is coNP-hard. In

particular, [23] 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 [23], 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^+ [24]. Of course, this argument only holds for setting imposing the same restrictions on input and output schemas.

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.

6 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 (cfr. Proposition 4(3)), we can also generate a counterexample tree for the typechecking algorithm in polynomial time. Further, from the proof of Lemma 36 it follows that if an instance of $TC[\mathcal{T}_{\text{trac}}, DTD(DFA)]$ does not typecheck, we either have that $t_a^{\min}$ or t_a^{vast} is a counterexample, where a is the start symbol of the DTD. Note that both trees can be easily represented by a polynomial sized extended context free grammar. We have thus obtained the following.

Corollary 38 *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.

Note that in the case of $TC[\mathcal{T}_{d,c}, DTD(\text{RE}^+)]$, testing whether $t_a^{\min}$ or t_a^{vast} are counterexamples gives rise to a slightly different typechecking algorithm than the one we exhibited in Section 5. However, we believe that algorithms similar to the one in Section 5 might also be useful for typechecking with respect to DTDs using other formalisms than RE$^+$-expressions, or for incomplete typechecking algorithms. This is not the case for the algorithm that tests whether $t_a^{\min}$ or t_a^{vast} are counterexamples.

We say that an instance of the typechecking problem typechecks *almost always* if 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 [16]. Since the finiteness problem of NTA(NFA) is decidable in PTIME, according to Proposition 4(1), we have obtained the following.

Corollary 39 *Almost always typechecking of $\mathcal{T}_{trac}$ transducers with respect to DTD(DFA)s is in* PTIME.

7 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.

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. 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 deletion 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.

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.

A Appendix: Basic Results

In the following lemma, we treat the emptiness problem for DTA^cs: given a DTA^c A, is $L(A) = \emptyset$?

Proof of Lemma 3 *The emptiness problem is* PTIME*-complete for DTA^c(DFA).*

PROOF. The upper bound follows from a reduction to the emptiness problem for NTA(NFA)s, which is in PTIME (cf. Proposition 4).

For the lower bound, we reduce PATH SYSTEMS [9], which is known to be PTIME-complete, to our problem. PATH SYSTEMS is the decision problem defined as follows: given a finite set P, a set $A \subseteq P$ of axioms, a set $R \subseteq P^3$ of inference rules and some $p \in P$, is p provable from A using R? We construct a DTA^c(DFA) $A = (\Sigma \cup \{q_{\text{error}}\}, \Sigma, \delta, \Sigma)$ such that $L(A)$ is empty if and only if p is provable. In particular, for every $(a, b, c) \in R$, we add the string ab to $\delta(c, c)$; for every $a \in A$, $\delta(a, a) = \{\varepsilon\}$. Further, for every $a \in \Sigma$ we define $\delta(q_{\text{error}}, a)$ as $(\Sigma \cup \{q_{\text{error}}\})^* - L(\delta(a, a))$. Clearly, (d, p) satisfies the requirements. □

Proof of Proposition 4

(1) Finiteness of NTA(NFA) is in PTIME*.*
(2) Emptiness of NTA(NFA) is in PTIME*.*
(3) For a NTA(NFA) N, we can generate a description of a tree $t \in L(N)$ in PTIME*.*

PROOF. Part (1) immediately follows from results in [8]. Indeed, an efficient way to test for finiteness of is to check the existence of a loop. A language is infinite if and only if there is a loop on some useful state, that is, some state that can be used in an accepting run on some tree.

Part (2) immediately follows from results in [21]. We briefly give the algorithm as it is used in part (3). To this end, let $A = (Q, \Sigma, \delta, F)$ be an NTA(NFA). The emptiness algorithm is then depicted in Figure A.1. When the algorithm is finished, we have that $L(A)$ is empty if and only if R does not contain a final state.

Further, part (3) is an easy adaptation of the emptiness algorithm in part (2). Indeed, for every computed state $q \in R_i$ where $i > 1$, we can remember the witnesses symbol $a \in \Sigma$ and the string $w \in R_{i-1}^* \cap \delta(q, a)$. Using these witnesses, a DAG-representation of the counterexample tree t can easily be computed in a top-down manner, starting from an accepting state in $R_{|Q|}$. □

$R_1 := \{q \in Q \mid \exists a \in \Sigma, \varepsilon \in \delta(q,a)\}$;
for $i := 2$ to $|Q|$ **do**
  $R_i := \{q \in Q \mid \exists a \in \Sigma, \delta(q,a) \cap R^*_{i-1} \neq \emptyset\}$;
end for
$R := R_{|Q|}$;

Fig. A.1. The emptiness algorithm in [21] computing the set R of reachable states.

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] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *Proceedings of the 24th Symposium on Principles of Database Systems (PODS 2005)*, pages 25–36. ACM Press, 2005.

[4] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[5] 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.

[6] 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.

[7] J. Clark and S. DeRose. XML Path Language (XPath). `http://www.w3.org/TR/xpath`, 1999.

[8] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997.

[9] S.A. Cook. An observation on time-storage trade-off. *Journal of Computer and System Sciences*, 9(3):308–316, 1974.

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

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[12] 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.

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

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

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

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

[17] D. Kozen. Lower bounds for natural proof systems. In *Proceedings 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 254–266. IEEE, 1977.

[18] 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.

[19] S. Maneth and F. Neven. Structured document transformations based on XSL. In *Research Issues in Structured and Semistructured Database Programming (DBPL 1999)*, pages 79–96. Springer, 2000.

[20] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proceedings of the 23d Symposium on Principles of Database Systems (PODS 2004)*, pages 23–34. ACM Press, 2004.

[21] W. Martens and F. Neven. On the complexity of typechecking top-down XML transformations. *Theoretical Computer Science*, 336(1):153–180, 2005.

[22] W. Martens, F. Neven, and M. Gyssens. On typechecking top-down XML transformations: Fixed input or output schemas. Submitted, available at http://alpha.uhasselt.be/wim.martens, 2005.

[23] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. In *Proceedings of the 29th International Symposium on Mathematical Foundations of Computer Science (MFCS 2004)*, pages 889–900. Springer, 2004.

[24] W. Martens, F. Neven, and T. Schwentick. Complexity of decision problems for simple regular expressions. Full Version of [23]. In preparation, available at http://alpha.uhasselt.be/wim.martens, 2005.

[25] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[26] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS 1999)*, pages 215–226. ACM Press, 1999.

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

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

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

[30] 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.

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

[32] M. Sipser. *Introduction to the Theory of Computation*. Brooks/Cole Publishing, 1997.

[33] C.M. Sperberg-McQueen and H. Thompson. XML Schema. http://www.w3.org/XML/Schema, 2005.

[34] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *Conference Record of the 5th Annual ACM Symposium on Theory of Computing (STOC 1973)*, pages 1–9. ACM Press, 1973.

[35] D. Suciu. Typechecking for semistructured data. In *Proceedings of the 8th Workshop on Data Bases and Programming Languages (DBPL 2001)*, pages 1–20. Springer, 2001.

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

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

[38] P. T. Wood. Containment for XPath fragments under DTD constraints. Full version of [40].

[39] P. T. Wood. Minimising simple XPath expressions. In *Proceedings of the 4th International Workshop on the Web and Databases (WebDB 2001)*, pages 13–18, 2001.

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