Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 336 (2005) 153–180

Theoretical Computer Science

www.elsevier.com/locate/tcs

On the complexity of typechecking top-down XML transformations

Wim Martens*, Frank Neven

Limburgs Universitair Centrum, Universitaire Campus, B-3590 Diepenbeek, Belgium

Abstract

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

© 2004 Elsevier B.V. All rights reserved.

Keywords: XML; XSLT; Tree transformations; Typechecking; Unranked tree transducers; Complexity

1. Introduction

XML has emerged as the lingua franca of the Web [1]. The main difference with semi-structured data is the possibility to define schemas. In the context of the Web, such schemas can be used to validate data exchange. In a typical scenario, a user community agrees on a common schema and on producing only XML data conforming to that schema. This raises the issue of typechecking: verifying at compile time that every XML document which is the result of a specified query applied to a valid input, satisfies the output schema [29,30].

* Corresponding author.
E-mail addresses: wim.martens@luc.ac.be (W. Martens), frank.neven@luc.ac.be (F. Neven).

0304-3975/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2004.10.035

In the present paper, we focus on typechecking of XML to XML transformations. As types we adopt the usual document type definitions (DTDs) and their robust extension: regular tree languages [5,15,19] or, equivalently, specialized DTDs [24,25]. The latter serve as a formal model for XML schema [8].

Obviously, typechecking depends on the transformation language at hand. As shown by Alon et al. [2,3], when transformation languages have the ability to compare data values, the typechecking problem quickly turns undecidable. However, Milo, Suciu, and Vianu argued that XML documents can be abstracted by labeled ordered trees and that the capability of most XML transformation languages can be encompassed by k-pebble transducers when data values are ignored [19]. Further, the authors showed that the typechecking problem in this context is decidable. More precisely, given two types τ_1 and τ_2, represented by tree automata, and a k-pebble transducer T, it is decidable whether $T(t) \in \tau_2$ for all $t \in \tau_1$. Here, $T(t)$ is the tree obtained by running T on input t. The complexity, however, is non-elementary and cannot be improved [19].

In an attempt to lower the complexity, we consider much simpler tree transformations: those defined by deterministic top-down uniform tree transducers on unranked trees. Such transformations correspond to structural recursion on trees [6] and to simple top-down XSLT transformations [4,7]. Such transformations are merely used for restructuring and filtering, not for advanced querying (cf. Example 7). The transducers are called uniform as they cannot distinguish between the order of siblings. In brief, a transformation consists of a single top-down traversal of the input tree where every node is replaced by a new tree (possibly the empty tree).

The present paper gives an account of the complexity of the typechecking problem in the latter setting. The complexity is measured in the sizes of the input and output schema plus the size of the transducer. We parameterize the typechecking problem by the kind of allowed schemas and tree transducers. For instance, for DTDs we allow right-hand sides to be represented by DFAs, NFAs or formulas from a logic $\mathcal{SL}$ specifying unordered languages. Tree automata (abstracting XML schema) can be deterministic or non-deterministic.

In Section 3, we discuss typechecking without any restriction on transducers. We show that even for very weak DTDs (e.g., DTDs that use DFAs to represent regular languages) the typechecking problem is EXPTIME-complete. The main dominating factor is the ability of the transducer to delete interior nodes (cf. Example 7 where intermediate section nodes are deleted). Therefore, we focus on non-deleting transformations in the remainder of the paper. In Section 4, we distinguish between tree automata and DTDs as schema languages. In the case of tree automata, the complexity remains EXPTIME-hard. When considering DTDs the complexity drops to PSPACE when NFAs or DFAs are used to specify right-hand sides; when $\mathcal{SL}$-formulas are used the complexity drops to CONP. The PSPACE lower bound crucially depends on the ability of a transducer to make arbitrary copies of the input tree. However, in practice this ability is rarely needed. Usually, the number of copies a transducer makes is rather small (cf. Example 7 where the first rule makes two copies of every chapter). Therefore, it makes sense to consider the class of transducers making at most k copies where k is a number fixed in advance. We show in Section 5 that even on this class, in the case of tree automata and DTDs with NFAs, the complexity remains EXPTIME and PSPACE-hard, respectively. Only when right-hand sides of rules are represented by DFAs, the typechecking problem becomes tractable.

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

	NTA	DTA
Deleting + copying	EXPTIME	EXPTIME
Non-deleting	EXPTIME	EXPTIME
Non-deleting + bounded copying	EXPTIME	In EXPTIME/PSPACE-hard

Table 2
The presented results for DTDs: the top row of the table shows the representation of the input and output schemas and the left column shows the type of tree transducer

	DTD(NFA)	DTD(DFA)	DTD($\mathcal{SL}$)
Deleting + copying	EXPTIME	EXPTIME	EXPTIME
Non-deleting	PSPACE	PSPACE	CONP
Non-deleting + bounded copying	PSPACE	PTIME	CONP

In conclusion, our inquiries reveal that the complexity of the typechecking problem is determined by three features: (1) the ability of the transducer to delete interior nodes; (2) the ability to make an unbounded number of copies of subtrees; and, (3) non-determinism in the schema languages. Only when we disallow all three features, we get a PTIME complexity for the typechecking problem.

An overview of our results is given in Tables 1 and 2. Unless specified otherwise, all complexities are both upper and lower bounds. The top rows of the tables show the representation of the input and output schemas and the left columns show the type of tree transducer. NTA and DTA stand for non-deterministic and deterministic tree automata, respectively. DTD(X) stands for DTDs that use X to represent their regular languages. The exact definitions are given in Section 2.

Related work: A problem related to typechecking is type inference [18,24]. 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 [24].

The transducers considered in the present paper are restricted versions of the ones studied by Maneth and Neven [16]. They already obtained a non-elementary upper bound on the complexity of typechecking (due to the use of monadic second-order logic in the definition of the transducers).

Although the structure of XML documents can be faithfully represented by unranked trees (these are trees without a bound on the number of children of nodes), Milo, Suciu, and Vianu chose to study k-pebble transducers over binary trees as there is an immediate encoding of unranked trees into binary ones, as shown in Section 6. The top-down variants of k-pebble transducers are well-studied on binary trees [13]. However, these results do not aid in the quest to characterize precisely the complexity of typechecking transformations on unranked trees. Indeed, as we show later in Section 6, the class of unranked tree transductions can *not* be captured by ordinary transducers working on the binary encodings. Macro tree transducers can simulate our transducers on the binary encodings [16,11], but as very little is known about their complexity this observation is not of much help. For these reasons, we chose to work directly with unranked tree transducers.

Tozawa considered typechecking w.r.t. tree automata for a fragment of top-down XSLT [31]. 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.

2. Definitions

The material in this paper is sometimes quite technical. To improve readability, we deferred definitions and lemmas that are only needed in proofs to an appendix. In the present section, we provide background on trees, automata, and uniform tree transducers which are necessary to understand the results in this paper.

First, we introduce some preliminary definitions. By $\mathbb{N}$ we denote the set of natural numbers. We fix a finite alphabet Σ. A *string* $w = w_1 \cdots w_n$ 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 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 *non-deterministic 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 on N for 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 $|\delta(q, a)| \leqslant 1$ 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 $\sigma \in \Sigma$ and $w \in \mathcal{T}_\Sigma^*$, $\sigma(w)$ is in $\mathcal{T}_\Sigma$. So, a tree is either ε (empty) or is of the form $\sigma(t_1 \cdots t_n)$ where each t_i is a tree. The latter denotes the tree where the subtrees $t_1, \ldots, t_n$ are attached to the root labeled σ. We write σ rather than $\sigma()$. Note that there is no a priori bound on the number of children of a node in a Σ-tree; such trees are therefore *unranked*. In the following, whenever we say tree, we always mean Σ-tree. A *tree language* is a set of trees.

Later, we will allow hedges in the right-hand side of transducer rules: a *hedge* is a finite sequence of trees. So, the set of hedges, denoted by $\mathcal{H}_\Sigma$, is defined as $\mathcal{T}_\Sigma^*$.

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

- if $h = \varepsilon$, then $\mathrm{Dom}(h) = \emptyset$; (the empty hedge has no nodes),
- if $h = t_1 \cdots t_n$ where each $t_i \in \mathcal{T}_\Sigma$, then $\mathrm{Dom}(h) = \bigcup_{i=1}^n \{iu \mid u \in \mathrm{Dom}(t_i)\}$; ($iu$ refers to node u in the ith tree) and,

- if $h = \sigma(w)$, then $\mathrm{Dom}(h) = \{\varepsilon\} \cup \mathrm{Dom}(w)$ (if h is a tree then its domain consists of the domain of the hedges w and of the root ε).

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

2.2. DTDs

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

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

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

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

To define unordered languages we make use of the specification language $\mathcal{SL}$ inspired by Neven and Schwentick [21] and also used in [2,3]. The syntax of the language is as follows.

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

A string w over Σ satisfies an atomic formula $a^{=i}$ if it has exactly i occurrences of a; w satisfies $a^{\geqslant i}$ if it has at least i occurrences of a. Further, true is satisfied by every string. [1] Satisfaction of Boolean combinations of atomic formulas is defined in the obvious way. By $w \vDash \phi$, we denote that w satisfies $\mathcal{SL}$-formula ϕ.

As an example, consider the $\mathcal{SL}$-formula co-producer$^{\geqslant 1} \rightarrow$ producer$^{\geqslant 1}$. This expresses the constraint that a co-producer can only occur when a producer occurs. The *size* of an $\mathcal{SL}$-formula is the number of symbols that occur in it (every i in $a^{=i}$ or $a^{\geqslant i}$ is written in binary notation).

So, by DTD($\mathcal{SL}$) we then denote DTDs where right-hand sides are represented by $\mathcal{SL}$-formulas.

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

2.3. Tree automata

We recall the definition of non-deterministic tree automata from [5]. We refer the unfamiliar reader to [20] for a gentle introduction.

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

A *run* of B on a tree t is a labeling $\lambda : \mathrm{Dom}(t) \to Q$ such that for every $v \in \mathrm{Dom}(t)$ with n children, $\lambda(v1) \cdots \lambda(vn) \in \delta(\lambda(v), \mathrm{lab}^t(v))$. Note that when v has no children, then the criterion reduces to $\varepsilon \in \delta(\lambda(v), \mathrm{lab}^t(v))$. A run is *accepting* iff the root is labeled with an accepting state, that is, $\lambda(\varepsilon) \in F$. A tree is accepted if there is an accepting run. The set of all accepted trees is denoted by $L(B)$ and is called a *regular tree language*. When $\lambda(v) = q$, we sometimes also say that B assigns q to v.

We extend the definition of δ to trees and hedges by defining a function $\delta^*(h) : \mathcal{H}_\Sigma \to (2^Q)^*$ as follows:

- $\delta^*(a) = \{q \mid \varepsilon \in \delta(q, a)\}$;
- $\delta^*(a(t_1 \cdots t_n)) = \{q \mid \exists q_1 \in \delta^*(t_1), \ldots, \exists q_n \in \delta^*(t_n) \text{ and } q_1 \cdots q_n \in \delta(q, a)\}$;
- $\delta^*(t_1 \cdots t_n) = \delta^*(t_1) \cdots \delta^*(t_n)$.

Note that a tree t is accepted by B if $\delta^*(t) \cap F \neq \emptyset$.

A tree automaton is *bottom-up deterministic* if for all $q, q' \in Q$ with $q \neq q'$ and $a \in \Sigma$, $\delta(q, a) \cap \delta(q', a) = \emptyset$. We denote the set of bottom-up deterministic NTAs by DTA. A tree automaton is *top-down deterministic* if for all $q, q' \in Q$ with $q \neq q'$, $a \in \Sigma$, and $n \geqslant 0$, $\delta(q, a)$ contains at most one string of length n.

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

2.4. Transducers

We next define the tree transducers used in this paper. To simplify notation, we restrict to one alphabet. That is, we consider transductions mapping Σ-trees to Σ-trees. It is straightforward to define transductions where the input alphabet differs from the output alphabet [16].

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

Definition 4. A *uniform tree transducer* is a tuple (Q, Σ, q^0, R), where Q is a finite set of states, Σ is the input and output alphabet, $q^0 \in Q$ is the initial state, and R is a finite set of rules of the form $(q, a) \to h$, where $a \in \Sigma$, $q \in Q$, and $h \in \mathcal{H}_\Sigma(Q)$. When $q = q^0$, h is restricted to $\mathcal{T}_\Sigma(Q) \setminus Q$.

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

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

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

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

Fig. 1. The XSLT program equivalent to the transducer of Example 5.

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

Example 5. Let $T = (Q, \Sigma, p, R)$ where $Q = \{p, q\}$, $\Sigma = \{a, b, c, d\}$, and R contains the rules

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

Our definition of tree transducers corresponds to structural recursion [6] and a fragment of top-down XSLT. For instance, the XSLT program equivalent to the above transducer is given in Fig. 1 (we assume the program is started in mode p). Note that the right-hand side of $(q, a) \to c\ q$ is a hedge, while the other right-hand sides are trees.

The translation defined by $T = (Q, \Sigma, q^0, R)$ on a tree t in state q, denoted by $T^q(t)$, is inductively defined as follows: if $t = \varepsilon$ then $T^q(t) := \varepsilon$; if $t = a(t_1 \cdots t_n)$ and there is a rule $(q, a) \to h \in R$ then $T^q(t)$ is obtained from h by replacing every node u in h labeled with p by the hedge $T^p(t_1) \cdots T^p(t_n)$. Note that such nodes u can only occur at leaves. So, h is only extended downwards. If there is no rule $(q, a) \to h \in R$ then $T^q(t) := \varepsilon$. Finally, define the transformation of t by T, denoted by $T(t)$, as $T^{q^0}(t)$.

For $a \in \Sigma$, $q \in Q$ and $(q, a) \to h \in R$, we denote h by $\mathrm{rhs}(q, a)$. We also use the abbreviation rhs to stand for right-hand side. If q and a are not important, we say that h is a rhs. The *size* of T is $|Q| + |\Sigma| + \sum_{q \in Q, a \in \Sigma} |\mathrm{rhs}(q, a)|$.

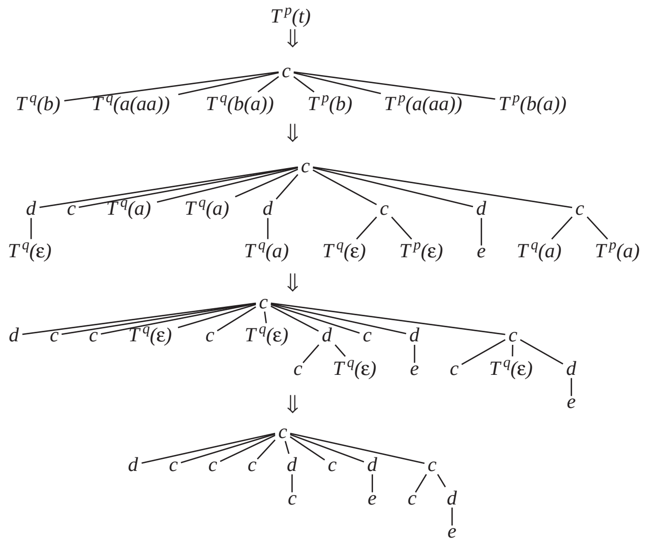
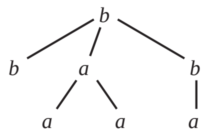


Fig. 2. The translation of $t = b(b\,a(a\,a)b(a))$ by the transducer T of Example 5.

Example 6. In Fig. 2 we give the translation of the tree t defined as



by the transducer of Example 5.

We discuss two important features: *copying* and *deletion*. The rule $(p, b) \to c(q\ p)$ in the above example copies the children of the current node in the input tree two times: one copy is processed in state q and the other in state p. The symbol c is the parent node of the two copies. So the current node in the input tree corresponds to the latter node. The rule $(q, a) \to c\,q$ copies the children of the current node only once. However, no parent node is given for this copy. So, there is no corresponding node for the current node in the input tree. We, therefore, say that it is deleted. For instance, $T^q(a(b)) = c\,d$ where d corresponds to b and not to a.

Example 7. We provide a less abstract example of a transformation. The following DTD (DFA) defines a schema for books:

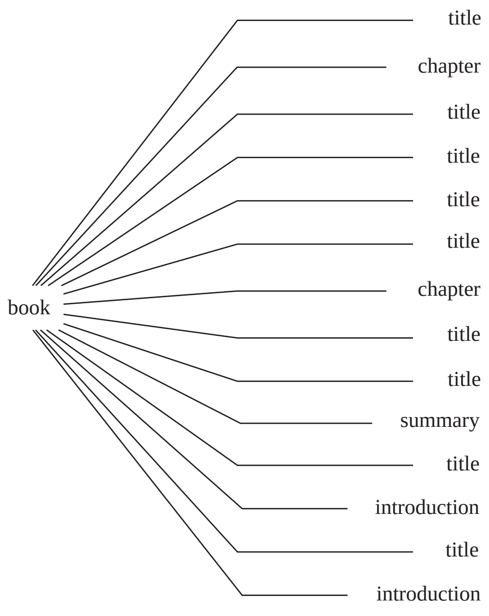
$$\begin{array}{ll} \texttt{book} & \to \texttt{title}, \texttt{author}^+, \texttt{chapter}^+ \\ \texttt{chapter} & \to \texttt{title}, \texttt{introduction}, \texttt{section}^+ \\ \texttt{section} & \to \texttt{title}, \texttt{paragraph}^+, \texttt{section}^* \end{array}$$

We use ‘,’ to denote concatenation. Fig. 3 depicts a document conform to the given schema. The following transducer makes a table of contents by generating for every chapter of the book a list of its section titles. In addition, a summary of the book consisting of the title and introduction of each chapter is added.

$$(q_0, \texttt{book}) \to \texttt{book}(p\ \texttt{summary}\ q)$$
$$(p, \texttt{chapter}) \to \texttt{chapter}\ \ p$$
$$(p, \texttt{title}) \to \texttt{title}$$
$$(p, \texttt{section}) \to p$$
$$(q, \texttt{chapter}) \to q'$$
$$(q', \texttt{title}) \to \texttt{title}$$
$$(q', \texttt{introduction}) \to \texttt{introduction}$$

The rule $(q_0, \texttt{book}) \to \texttt{book}(p\ q)$ makes two copies of each chapter, each of which is processed in states p and q, respectively. State p recursively generates a list of titles. The rule $(p, \texttt{chapter}) \to \texttt{chapter}\ \ p$ allows to list these titles next to the chapter element rather than below. Note that state p deletes all intermediate section nodes. State q generates a list of all chapter titles together with their introductions. By using state q', we make sure that the title of the book is skipped.

The output of the transformation, applied to the document in Fig. 3 is the following tree:



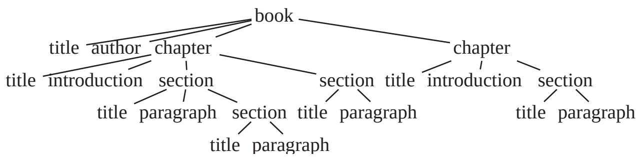


Fig. 3. A document conforming to the schema of Example 7.

2.5. The typechecking problem

We define the problem central to this paper.

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

Definition 9. Given S_{in}, S_{out} and T, the *typechecking problem* consists in verifying whether T typechecks w.r.t. S_{in} and S_{out}.

Example 10. The transducer in Example 7 typechecks w.r.t. the input DTD and the following output DTD:

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

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

Next, we define some classes of tree transducers based on the discussion on deletion and copying following Example 6. A transducer is *non-deleting* if no states occur at the top-level of a rhs. We denote by $\mathcal{T}_g$ the class of all transducers and by $\mathcal{T}_{\text{nd}}$ the class of non-deleting transducers. A transducer T has *copying width* k if there are at most k occurrences of states in a sequence of siblings in a rhs. For instance, the copy width of the transducer in Example 7 is two. By $\mathcal{BW}_k$ we denote the class of non-deleting transducers of bounded copying width k. For a class of representations of regular string languages $\mathcal{M}$, we write TC[$\mathcal{T}$,$\mathcal{M}$] rather than TC[$\mathcal{T}$, DTD($\mathcal{M}$)].

3. The general case

In the present section, we consider the complexity of the typechecking problem in its most general setting. That is, without any restrictions on transducers: both deletion and unbounded copying is allowed. We show that the problem is in EXPTIME for the most powerful schema languages, namely non-deterministic tree automata. However, the problem remains hard for EXPTIME even for the weakest DTDs: DTDs where right-hand sides are specified by DFAs or $\mathcal{SL}$-formulas.

The lower bound is obtained through a reduction from the intersection emptiness problem of n deterministic tree automata which is known to be hard for EXPTIME [27]. The transducer starts by making n copies of the input tree. Thereafter, it simulates a different tree automaton on each copy. All processed nodes are deleted. The only generated output is an error symbol when an automaton rejects. So, the output DTD merely has to check that an error symbol always appears. The latter can be done by a very simple DFA or $\mathcal{SL}$-formula.

The EXPTIME upper bound is obtained by a translation to typechecking of non-deleting transducers. The latter is tackled in the next section.

Theorem 11. (1) *$TC[\mathcal{T}_g, NTA]$ is in* EXPTIME;
(2) *$TC[\mathcal{T}_g, \mathcal{SL}]$ is* EXPTIME-*hard*;
(3) *$TC[\mathcal{T}_g, DFA]$ is* EXPTIME-*hard*.

Proof. (1) Let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a transducer and let A_{in} and $A_{\text{out}} = (Q_A, \Sigma, \delta_A, F_A)$ be two NTAs representing the input and output schema, respectively. We next describe a *non*-deleting transducer S and an NTA B_{out} which can be constructed in LOGSPACE, such that T typechecks w.r.t. A_{in} and A_{out} iff S typechecks w.r.t. A_{in} and B_{out}. From Theorem 12(1) it then follows that TC[$\mathcal{T}_g$,NTA] is in EXPTIME.

Intuitively, S outputs a # whenever T would process a deleting state. For instance, the rule $(q, a) \to c\, q$ is replaced by $(q, a) \to c\,\#(q)$. We assume that $\# \notin \Sigma$. Formally, $S = (Q_S, \Sigma \cup \{\#\}, q_S^0, R_S)$ with $Q_S = Q_T$, $q_S^0 = q_T^0$, and for every rule $(q, a) \to t_1 \cdots t_n$ in R_T, R_S contains the rule $(q, a) \to t'_1 \cdots t'_n$, where for every $i = 1, \ldots, n$, $t'_i = \#(t_i)$ if $t_i \in Q_T$ and $t'_i = t_i$ otherwise. Then, define the #-eliminating function γ as follows: $\gamma(a(h))$ is $\gamma(h)$ when $a = \#$ and $a(\gamma(h))$ otherwise; further, $\gamma(t_1 \cdots t_n) := \gamma(t_1) \cdots \gamma(t_n)$. So, clearly, for all $t \in \mathcal{T}_\Sigma$, $T(t) = \gamma(S(t))$.

Next, we construct B_{out} such that $\gamma(t) \in L(A_{\text{out}})$ iff $t \in L(B_{\text{out}})$. The underlying idea is quite simple. In a run on $\#(t_1 \cdots t_n)$, B_{out} assigns a state (q_1, q_2, q, a) to the root when the NFA for $\delta_A(q, a)$ halts in state q_2 when processing top$(\gamma(\#(t_1 \cdots t_n)))$ starting in state q_1. Here, q_1, q_2 are states of the automaton for $\delta_A(q, a)$, q is a state of A_{out} and $a \in \Sigma$. The state q and the label a are guessed. In a run on $a(t_1 \cdots t_n)$, with $a \neq \#$, B_{out} assigns a state q to the root when A_{out} assigns q to the root of $\gamma(a(t_1 \cdots t_n))$.

Let for every $a \in \Sigma$ and $q \in Q_A$, $N^{q,a} = (Q^{q,a}, Q_A, \delta^{q,a}, I^{q,a}, F^{q,a})$ be the NFA such that $\delta_A(q, a) = L(N^{q,a})$. We tacitly assume that all $Q^{q,a}$ are disjoint. Define $B_{\text{out}} = (Q_B, \Sigma \cup \{\#\}, \delta_B, F_B)$, where $Q_B = Q_A \cup \{(q_1, q_2, q, a) \mid q \in Q_A, a \in \Sigma, q_1, q_2 \in Q^{q,a}\}$, and $F_B = F_A$.

It remains to define δ_B. Thereto, fix $q \in Q_A$ and $a \in \Sigma$. Let $I, F \subseteq Q^{q,a}$. Let $M^{q,a}(I, F)$ be the automaton behaving in the same way as $N^{q,a}$ with the initial and final states replaced with I and F, respectively; further, when reading a tuple (q_1, q_2, p, b) in state q_1 the automaton jumps to state q_2 when $p = q$ and $b = a$, and rejects otherwise. Clearly, $M^{q,a}(I, F)$ is LOGSPACE constructible from $N^{q,a}$. We then simply define $\delta_B(q, a) := M^{q,a}(I^{q,a}, F^{q,a})$ and $\delta_B((q_1, q_2, p, b), \#) := M^{p,b}(\{q_1\}, \{q_2\})$ for all states $q, (q_1, q_2, p, b) \in Q_B$ and $a \in \Sigma$. It is not difficult to see that $\gamma(t) \in L(A_{\text{out}})$ iff $t \in L(B_{\text{out}})$.

(2) We use a reduction from the intersection emptiness problem of top-down deterministic ranked binary tree automata A_i ($i = 1, \ldots, n$), which is known to be hard for EXPTIME [27]. The problem is stated as follows, given top-down deterministic ranked binary tree automata

$A_1, \ldots, A_n$, is $\bigcap_{i=1}^n L(A_i) = \emptyset$? We define a transducer T and two DTDs d_{in} and d_{out} such that $\bigcap_{i=1}^n L(A_i) = \emptyset$ iff T typechecks w.r.t. d_{in} and d_{out}. In the construction, we exploit the copying power of transducers to make n copies of the input tree: one for each A_i. By using deleting states, we can execute each A_i on its copy of the input tree without producing output. When an A_i does not accept, we output an *error* symbol under the root of the output tree. The output DTD should then only check that an *error* symbol always appears.

Top-down deterministic *ranked* binary tree automata are NTAs which operate on an alphabet that is partitioned in internal labels and leaf labels. If a label a is an internal label, the regular languages $\delta(q, a)$ are empty or only contain one string of length two and if it is a leaf label, the regular languages $\delta(q, a)$ are empty or only contain the empty string. So, such automata are defined over full binary trees, that is, all inner nodes have precisely two children. Further, there is only one start state. Let for $i = 1, \ldots, n$, the top-down deterministic ranked binary tree automata be $A_i = (Q_i, \Sigma, \delta_i, \{q_0^i\})$.

First, we define the alphabet of the transducer. Let $\Sigma = \{a_1, \ldots, a_k\}$ and define $\Sigma_i = \{a_{j,i} \mid a_j \in \Sigma\}$, for $i = 1, 2$. The transducer is defined over the alphabet $\Sigma_T = \Sigma_1 \cup \Sigma_2 \cup \{\$, \text{error}, \text{ok}\}$. The intuition is as follows, the root symbol of the input tree is labeled with \$ and has only one child, which corresponds to the root of a possible input for the n binary tree automata. Every other internal node has two children: a left and a right child labeled with an element of Σ_1 and Σ_2, respectively. Using labels from Σ_1 and Σ_2 allows the transducer to distinguish a left from a right child by simply inspecting its label. Note that the partitioning of leaf nodes and internal nodes in Σ also allows us to distinguish leaf labels from internal labels in Σ_T.

Next, we define the input DTD. Formally, for every internal symbol $a \in \Sigma_T \setminus \{\text{error}, \text{ok}\}$, define $d_{\text{in}}(a) = \bigvee_{i,j=1}^k (C_i^1 \wedge C_j^2)$. Here, $(C_i^1 \wedge C_j^2)$ is the $\mathcal{SL}$-formula expressing that there are two children, one labeled with $a_{i,1}$ and one with $a_{j,2}$ meaning that the first child is a_i and the second is a_j. Formally, for $i, j = 1, \ldots, k$,

$$C_i^1 = \left(\bigwedge_{\ell=1,\ldots,k} (a_{\ell,1}^{=\delta_{\ell i}}) \right) \quad \text{and} \quad C_j^2 = \left(\bigwedge_{\ell=1,\ldots,k} (a_{\ell,2}^{=\delta_{\ell j}}) \right),$$

where $\delta_{\ell i}$ is the Kronecker delta ($\delta_{\ell i} = 1$ if $\ell = i$ and $\delta_{\ell i} = 0$ otherwise). Further, for every leaf symbol $a \in \Sigma_T \setminus \{\text{error}, \text{ok}\}$, define $d_{\text{in}}(a)$ as the empty string. Finally, the start symbol of d_{in} is \$ and define $d_{\text{in}}(\$) = \bigvee_{i=1}^k C_i^1$. The size of $d_{\text{in}}(a)$ is $O(|\Sigma|^3)$.

The transducer $T = (Q_T, \Sigma_T, q_T^0, R_T)$ simulates in parallel the n tree automata on the input tree. When an automaton rejects, the transducer produces an error symbol. However, using deleting states, it only produces output when a leaf node is reached. In this way only a very simple DTD is needed to check whether an error occurred. The transducer is defined as follows: $Q_T = \bigcup_{i=1}^n (Q_i^1 \cup Q_i^2)$, where $Q_i^j = \{q^j \mid q \in Q_i\}$ for $j = 1, 2$. The intuition is that states in Q_i^j should only be used to process the jth child. We tacitly assume that the sets Q_i are disjoint. R_T consists of the following rules:

- $(q_T^0, \$) \to \$(q_0^1 \cdots q_0^n)$. Recall that q_0^i is the initial state of A_i. So, this rule puts a \$ as the root symbol of the output tree and starts the in-parallel simulation of the A_i's.
- For all $m, m' \in \{1, 2\}$ with $m \neq m'$ and $j \in \{1, \ldots, k\}$, add the rule $(q^m, a_{j,m'}) \to \varepsilon$. Left children cannot be processed by right states and vice versa.

- Let $m \in \{1, 2\}$, $j \in \{1, \ldots, k\}$, $i \in \{1, \ldots, n\}$, and $q^m \in Q_i^m$. If a_j is an internal symbol and $\delta_i(q, a_j) = \ell r$, then we add the rule $(q^m, a_{j,m}) \rightarrow \ell^1 r^2$. If a_j is a leaf symbol and $\delta_i(q, a_j) = \varepsilon$, then we add the rule $(q^m, a_{j,m}) \rightarrow$ ok. In both cases, if $\delta_i(q, a_j)$ is empty, we add the rule $(q^m, a_{j,m}) \rightarrow$ error.

Finally, define $d_{\text{out}}(\$) :=$ error$^{\geqslant 1}$. Here, \$ is the start symbol. It remains to verify the correctness. Suppose $t \in \bigcap_{i=1\ldots n} L(A_i)$, then $t' \in L(d_{\text{in}})$ and $T(t')$ contains no error-labeled node where t' is obtained from $\$(t)$ by changing the label of every first (second) child labeled a_j by $a_{j,1}$ ($a_{j,2}$). Conversely, if $t \in L(d_{\text{in}})$ and $T(t)$ does not contain an error symbol, then $t' \in \bigcap L(A_i)$ where t' is obtained from t by dropping the \$-labeled symbol, rearranging children according to their index-number and then dropping the indices.

The proof of (3) follows from the one for (2) as the used $\mathcal{SL}$ can easily be expressed by DFAs of the same sizes. $\square$

4. Non-deleting transformations

The lower bound of the previous section severely depends on the ability of transducers to delete interior nodes and to make an unbounded number of copies of subtrees. In an attempt to lower the complexity, we restrict to non-deleting transformations in the present section. We observe that when schemas are represented by tree automata, the complexity remains EXPTIME-hard. When tree languages are represented by DTDs, the complexity of the typechecking problem drops to PSPACE and is hard for PSPACE even when right-hand sides of rules are represented by DFAs. When employing $\mathcal{SL}$-formulas the complexity is CONP. In summary, we prove the following results:

Theorem 12. (1) *$TC[\mathcal{T}_{\text{nd}}, NTA]$ is* EXPTIME-*complete*;
(2) *$TC[\mathcal{T}_{\text{nd}}, DTA]$ is* EXPTIME-*complete*;
(3) *$TC[\mathcal{T}_{\text{nd}}, NFA]$ is* PSPACE-*complete*;
(4) *$TC[\mathcal{T}_{\text{nd}}, DFA]$ is* PSPACE-*complete*;
(5) *$TC[\mathcal{T}_{\text{nd}}, \mathcal{SL}]$ is* CONP-*complete.*

We prove the different parts of the above theorem in the following subsections.

4.1. Tree automata

The proof establishing the upper bound is similar in spirit to a proof in [22], which shows that containment of Query Automata is in EXPTIME.

Theorem 12(1). *$TC[\mathcal{T}_{\text{nd}}, NTA]$ is* EXPTIME-*complete.*

Proof. Hardness is immediate as containment of NTAs is already hard for EXPTIME [26]. We, therefore, only prove membership in EXPTIME. Let $T = (Q_T, \Sigma, q_T^0, R_T)$ be a non-deleting tree transducer and let $A_{\text{in}} = (Q_{\text{in}}, \Sigma, \delta_{\text{in}}, F_{\text{in}})$ and $A_{\text{out}} = (Q_{\text{out}}, \Sigma, \delta_{\text{out}}, F_{\text{out}})$ be the NTAs representing the input and output schema, respectively.

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

Fig. 4. The algorithm of Theorem 12(1) computing P.

In brief, our algorithm computes the set

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

Note that since $f(q) = \delta^*_{\text{out}}(T^q(t))$ and t is a tree,[2] the length of $f(q)$ is bounded by the size of the largest rhs in T. Therefore, the number of functions f we consider is bounded by $(2^{|Q_{\text{out}}|})^{|T||Q_T|}$. Intuitively, in the definition of P, t can be seen as a witness of (S, f). Indeed, S is the set of states reachable by A_{in} at the root of t, while for each state q of the transducer, $f(q)$ is the sequence of sets of states reachable by A_{out} at the root of $T^q(t)$. So, the given instance does *not* typecheck iff there exists an $(S, f) \in P$ such that $F_{\text{in}} \cap S \neq \emptyset$ and $F_{\text{out}} \cap f(q^0_T) = \emptyset$. As $T^{q^0_T}(t)$ is always a tree, $f(q^0_T)$ is a subset of Q_{out}. In Fig. 4, an algorithm for computing P is depicted. We will show that this algorithm is in EXPTIME. Hence, typechecking is in EXPTIME. We explain the notation in Fig. 4. By $\text{rhs}(q, a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T]$, we denote the hedge obtained from $\text{rhs}(q, a)$ by replacing every occurrence of a state p by the sequence $f_1(p) \cdots f_n(p)$. By $\hat{\delta}_{\text{out}} : \mathcal{H}_\Sigma(2^{Q_c}) \to (2^{Q_c})^*$ we denote the transition function extended to hedges in $\mathcal{H}_\Sigma(2^{Q_{\text{out}}})$. To be precise, for $a \in \Sigma$, $\hat{\delta}_{\text{out}}(a) := \{q \mid \varepsilon \in \delta_{\text{out}}(q, a)\}$; for $P \subseteq Q_{\text{out}}$, $\hat{\delta}_{\text{out}}(P) := P$; for $h = a(t_1 \cdots t_n)$, $\hat{\delta}_{\text{out}}(h) := \{q \mid \forall i = 1, \ldots, n, \exists q_i \in \hat{\delta}_{\text{out}}(t_i) : q_1 \cdots q_n \in \hat{\delta}_{\text{out}}(q, a)\}$; and for $h = t_1 \cdots t_n$, $\hat{\delta}_{\text{out}}(h) = \hat{\delta}_{\text{out}}(t_1) \cdots \hat{\delta}_{\text{out}}(t_n)$. The correctness of the algorithm follows from the following lemma which is proved by induction on the number of iterations of the while loop.

Lemma 13. *A pair (S, f) has a witness tree of depth i iff $(S, f) \in P_i$.*

Proof. Immediate for $i = 1$.

For the induction step, suppose that, for some i, every pair is in P_{i-1} iff it has a witness of depth $i - 1$. Let $(S, f) \in P_i$, then, by definition, there is an $a \in \Sigma$ and a string $(S_1, f_1) \cdots (S_n, f_n) \in P^*_{i-1}$ so that $S := \{p \mid \exists r_j \in S_j, j = 1, \ldots, n, r_1 \cdots r_n \in \delta_{\text{in}}(p, a)\}$ and for every $q \in Q_T$, $f(q) := \delta^*_{\text{out}}(\text{rhs}(q, a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T])$. Hence, $a(t_1 \cdots t_n)$ is a witness of (S, f), where each t_j is a witness for (S_j, f_j).

[2] Recall that $T^q(t)$ is the translation of t started in state q.

Conversely, suppose that (S, f) has a witness tree $a(t_1 \cdots t_n)$ of depth i. By the induction hypothesis, there exist tuples $(S_1, f_1), \ldots, (S_n, f_n) \in P_{i-1}$ such that t_j is a witness for (S_j, f_j) for each $j = 1, \ldots, n$. Considering the definition of $\hat{\delta}_{\text{out}}$, it is then clear that the algorithm of Fig. 4 puts (S, f) in P_i. $\square$

It remains to show that the algorithm is in EXPTIME. The set P_1 can be computed in time polynomial in the sizes of A_{in}, A_{out}, and T. As $P_i \subseteq P_{i+1}$ for all i, and there are $2^{|Q_{\text{in}}|} \cdot (2^{|Q_{\text{out}}|})^{|T||Q_T|}$ pairs (S, f), the loop can only make an exponential number of iterations. So, it suffices to show that each iteration can be done in EXPTIME. Actually, we argue that it can be checked in PSPACE whether a tuple $(S, f) \in P_i$.

Let (S, f) be a pair. We describe separately how S and f are checked. It should be clear how the two algorithms can be merged into one PSPACE algorithm. We start with S.

(1) For every $q \in Q_{\text{in}}$ and $a \in \Sigma$, let $N^{q,a}$ be the NFA accepting those strings $R_1 \cdots R_k \in (2^{Q_{\text{in}}})^*$ for which there are $r_i \in R_i$ such that $r_1 \cdots r_k \in \delta_{\text{in}}(q, a)$. It is too expensive to actually construct the automaton $N^{q,a}$ as the alphabet is exponentially bigger than the one of $\delta_{\text{in}}(q, a)$. However, the set of states is the same. It is important to note that given a set R_i and a state q, the set of all states reachable from q by reading R_i can be computed in PSPACE.

So, we need to check the existence of an $a \in \Sigma$ and a string $Z := S_1 \cdots S_n$ that is accepted (rejected) by $N^{q,a}$ for all $q \in S$ ($q \in Q_{\text{in}} \setminus S$). The latter can be achieved in PSPACE by guessing an $a \in \Sigma$ and then guessing Z one symbol at a time while executing all $N^{q,a}$'s in parallel for every $q \in Q_{\text{in}}$. Indeed, for every automaton we remember the set of states that can be reached by reading the prefix of Z seen so far. Initially, these sets are the respective initial states. Then, whenever a new S_i is guessed, for each automaton the set of states reachable from a state from the remembered set by reading S_i, is computed. By the discussion above the latter is in PSPACE.

(2) Checking f is more technical. We use the a guessed in the previous step. Denote $\text{rhs}(q, a)[p \leftarrow f_1(p) \cdots f_n(p) \mid p \in Q_T]$ by $\xi_{q,a}$. Now, we need to check for all $q \in Q_T$ whether $f(q) = \hat{\delta}_{\text{out}}(\xi_{q,a})$. For all $p \in Q_{\text{out}}$ and $b \in \Sigma$, let $M^{p,b}$ be the NFA accepting strings $R_1 \cdots R_k \in (2^{Q_{\text{out}}})^*$ for which there are $r_i \in R_i$, $i = 1, \ldots, k$, such that $r_1 \cdots r_k \in \delta_{\text{out}}(p, b)$. Again, we will not construct the latter automata. It is enough to realize that given a state and an $R \subseteq Q_{\text{out}}$, the set of states reachable from this state by reading R can be computed in PSPACE.

First, assume every $\text{rhs}(q, a)$ is of the form $b(q_1 \cdots q_\ell)$. Then, $\xi_{q,a}$ is of the form $b(w_1 \cdots w_\ell)$ with $w_j = f_1(q_j) \cdots f_n(q_j)$. So, to check that $f(q) = \hat{\delta}_{\text{out}}(\xi_{q,a})$, we need to verify that $w = w_1 \cdots w_\ell$ is accepted (rejected) by $M^{p,b}$ for all $p \in f(q)$ ($p \notin f(q)$). However, like in (1), our algorithm successively guesses new f_i's while forgetting the previous ones and should, hence, be able to run the automata on w in this way. As w consists of ℓ parts we guess ℓ sets of states $P_i^{p,b}$, $i = 0, \ldots, \ell$, where $P_0^{p,b}$ is the set of initial states of $M^{p,b}$. The meaning of these sets is the following: every automaton $M^{p,b}$ reaches precisely the states in $P_i^{p,b}$ after reading $w_1 \cdots w_{i-1}$. The algorithm can verify the latter criterion by running $M^{p,b}$ on each w_i separately started in the states $P_{i-1}^{p,b}$ and verifying whether $P_i^{p,b}$ is reached. Running $M^{p,b}$ on w_i can be done in PSPACE as described in (1).

When right-hand sides of rules can be arbitrary trees in $\mathcal{T}(Q_T)$, we guess for every inner node u in a $\mathrm{rhs}(q,a)$ a subset $R_u^{q,a}$ of Q_{out}. When u is the root, then $R_u^{q,a} = f(q)$. Intuitively, these sets represent precisely the sets of states that can be reached at a node u by A_{out}. For leaf nodes u, we define $R_u^{q,a}$ as $\delta^*_{\mathrm{out}}(c)$ and as the sequence $f_1(p)\cdots f_n(p)$ when u is labeled with c and p, respectively. We then need to verify for every inner node u labeled with b with n children, that $R_{u1}^{q,a}\cdots R_{un}^{q,a}$ is accepted (rejected) by $M^{p,b}$ for all $p \in R_u^{q,a}$ ($p \notin R_u^{q,a}$). Again, the latter is checked as described above.

Finally, when right-hand sides of rules can be hedges, one needs to take into account that $f(q)$ can be a sequence of sets of states. $\square$

In the remainder of this section, we examine what happens when tree automata are restricted to be deterministic. From the above result, it is immediate that $\mathrm{TC}[\mathcal{T}_{\mathrm{nd}},\ \mathrm{DTA}]$ is in EXPTIME. Hardness is obtained through a reduction from the intersection emptiness problem of top-down deterministic ranked binary tree automata and is similar to the one in Theorem 11(2): A_{in} defines the same set of trees as d_{in} does with the exception that A_{in} enforces an ordering of the children. The transducer in the proof of Theorem 11(2) starts the in parallel simulation of the n automata, but then, using deleting states, delays the output until it has reached the leaves of the input tree. In the present setting, we can not use deleting states. Instead, we copy the input tree and overwrite the leaves with error symbols when an automaton rejects. The output automaton then checks whether at least one error occurred.

Theorem 12(2). *$TC[\mathcal{T}_{\mathrm{nd}}, DTA]$ is* EXPTIME*-complete.*

Proof. For $i = 1,\ldots,n$, let $A_i = (Q_i, \Sigma, \delta_i, \{q_0^i\})$ be top-down deterministic ranked binary tree automata. The transducer is defined over the alphabet $\Sigma_T = \Sigma_1 \cup \Sigma_2 \cup \{\$, \mathrm{error}, \mathrm{ok}\}$. Here, $\Sigma_i = \{a_i \mid a \in \Sigma\}$, for $i = 1, 2$.

First, we define $A_{\mathrm{in}} = (Q_{\mathrm{in}}, \Sigma_T, \delta_{\mathrm{in}}, \{q_{\mathrm{in}}^1\})$, where $Q_{\mathrm{in}} = \{q_{\mathrm{in}}^1, q_{\mathrm{in}}^2\}$. The intuition is that A_{in} accepts all trees $\$(t)$ where each node in u in t has a left and a right child labeled with elements of Σ_1 and Σ_2, respectively if $\mathrm{lab}^t(u)$ is an internal label, and u has no children if $\mathrm{lab}^t(u)$ is a leaf label. The transition function is defined as follows:

- $\delta_{\mathrm{in}}(q_{\mathrm{in}}^1, \$) = q_{\mathrm{in}}^1$.
- $\delta_{\mathrm{in}}(q_{\mathrm{in}}^i, a_i) = q_{\mathrm{in}}^1 q_{\mathrm{in}}^2$ for $i = 1, 2$ if $a_i \in \Sigma_i$ is an internal label.
- $\delta_{\mathrm{in}}(q_{\mathrm{in}}^i, a_j) = \emptyset$ for all $a_j \in \Sigma_j$, $i \neq j$.
- $\delta_{\mathrm{in}}(q_{\mathrm{in}}^i, a_i) = \varepsilon$ for $i = 1, 2$ if $a_i \in \Sigma_i$ is a leaf label.

Note that A_{in} is bottom-up deterministic.

The transducer $T = (Q_T, \Sigma_T, q_T^0, R_T)$ is defined similarly as in Theorem 11(2): $Q_T = \bigcup_{i=1}^n (Q_i^1 \cup Q_i^2)$, where $Q_i^k = \{q^k \mid q \in Q_i\}$. Again, the intuition is that states in Q_i^j should only be used to process the jth child. R_T consists of the following rules:

- $(q_T^0, \$) \to \$(q_0^1 \cdots q_0^n)$. So, this rule puts a \$ as the root symbol of the output tree and starts the in-parallel simulation of the A_i's.
- For all $j, j' \in \{1, 2\}$ with $j \neq j'$, add the rule $(q^j, a'_j) \to \varepsilon$.
- Let $j \in \{1, 2\}$, $i \in \{1,\ldots,n\}$, and $q^j \in Q_i^j$. If a_j is an internal symbol and $\delta_i(q,a) = \ell r$, then we add the rule $(q^j, a_j) \to a_j(\ell^1 r^2)$. If a_j is a leaf symbol and $\delta_i(q,a) = \varepsilon$, then we add the rule $(q^j, a_j) \to \mathrm{ok}$. In both cases, if $\delta_i(q,a)$ is empty, we add the rule $(q^j, a_j) \to \mathrm{error}$.

Finally, we define the output automaton $A_{\text{out}} = (Q_{\text{out}}, \Sigma_T, \delta_{\text{out}}, \{q_e\})$ which accepts all trees with at least one error-labeled leaf. Formally, $Q_{\text{out}} = \{q_o, q_e\}$ and δ_{out} is defined as follows: $i \in \{1, 2\}$,

- $\delta_{\text{out}}(q_o, \$) = q_o^*$.
- $\delta_{\text{out}}(q_e, \$) = Q_{\text{out}}^* q_e Q_{\text{out}}^*$.
- $\delta_{\text{out}}(q_o, a_i) = q_o^*$ for all $a_i \in \Sigma_i$.
- $\delta_{\text{out}}(q_e, a_i) = Q_{\text{out}}^* q_e Q_{\text{out}}^*$ for all $a_i \in \Sigma_i$.
- $\delta_{\text{out}}(q_e, \text{error}) = \varepsilon$.
- $\delta_{\text{out}}(q_o, \text{ok}) = \varepsilon$.

Again, A_{out} is bottom-up deterministic. □

4.2. DTDs

When we consider DTD(NFA)s to represent input schemas the complexity drops to PSPACE. We reduce the typechecking problem to the emptiness problem of NTAs where transition functions are represented by loop-free two-way alternating string automata, denoted 2AFA^{lf}. The complexity of the latter problem is in PSPACE (Theorem 19 in the appendix). Alternating and string automata are discussed in the appendix (Section A.1). In particular, the constructed NTA accepts precisely those trees which satisfy the input DTD but are transformed by the transducer to trees outside the output DTD. Hence, the instance typechecks if and only if the NTA accepts the empty language. The proof makes use of two-way non-deterministic string automata, denoted 2NFA, which are also defined in the appendix.

Theorem 12(3). *$TC[\mathcal{T}_{\text{nd}}, \text{NFA}]$ is* PSPACE*-complete.*

Proof. The hardness result is immediate as containment of regular expressions is known to be PSPACE-hard [28]. For the other direction, let T be a non-deleting tree transducer. Let d_{in} and d_{out} be the input and output DTDs, respectively. We construct an NTA(2AFA^{lf}) B such that $L(B) = \{t \in L(d_{\text{in}}) \mid T(t) \notin L(d_{\text{out}})\}$. Moreover, the size of B is polynomial in the size of T, d_{in}, and d_{out}. Thus, $L(B) = \emptyset$ iff T typechecks w.r.t. d_{in} and d_{out}. By Theorem 19(2), the former is in PSPACE.

To explain the operation of the automaton, we introduce the following notions. Let q be a state of T and $a \in \Sigma$ then define $q(a) = \text{top}(\text{rhs}(q, a))$. For a string $w = a_1 \cdots a_n$, we define $q(w) := q(a_1) \cdots q(a_n)$. For a hedge h and a DTD 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.

Intuitively, the automaton B works as follows on $t \in \mathcal{T}_\Sigma$: (1) B checks that $t \in L(d_{\text{in}})$; (2) at the same time, B non-deterministically picks a node $v \in \text{Dom}(t)$ and a state q in which v is processed; B then accepts if h does not partly satisfy d_{out}, where h is obtained from $\text{rhs}(q, a)$ by replacing every state p by the string $p(\text{lab}^t(v1) \cdots \text{lab}^h(vn))$. Here, we assume that v is labeled a and has n children. As d_{out} is specified by NFAs and we have to check that d_{out} is *not* partly satisfied, we need to check membership in the complement of a regular expression. We therefore use alternation to specify the transition function of B.

Additionally, as T can copy its input, it is convenient to use two-way automata. The latter will become clear in the actual construction.

Formally, let $T = (Q_T, \Sigma, q_T^0, R_T)$. Define $B = (Q_B, \Sigma, F_B, \delta_B)$ as follows. The set of states Q^B is the union of the following sets: Σ, $\{(a, q) \mid a \in \Sigma, q \in Q_T\}$, and $\{(a, q, \text{check}) \mid a \in \Sigma, q \in Q_T\}$. If 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, check) 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}.

The set of final states is $F_B := \{(a, q_T^0) \mid a \in \Sigma\}$. The transition function is defined as follows: for all $a, b \in \Sigma$, $q \in Q_T$:

(1) $\delta_B(a, b) = \delta_B((a, q), b) = \delta_B((a, q, \text{check}), b) = \emptyset$ for all $a \neq b$;

(2) $\delta_B(a, a) = d_{\text{in}}(a)$ and $\delta_B((a, q), a)$ 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 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 satsify 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 NFAs whose size is polynomial in the size of the input.

(3) Finally, $\delta_B((a, q, \text{check}), a) = \{a_1 \cdots a_n \mid a_1 \cdots a_n \in d_{\text{in}}(a)$ and h does not partly satisfy $L(d_{\text{out}})\}$. Here, h is obtained from rhs(q, a) by replacing every state p by $p(a_1 \cdots a_n)$.

It remains to argue that $\delta_B((a, q, \text{check}), a)$ can be computed by a 2AFA$^{\text{lf}}$ A of polynomial size. We sketch the construction of this automaton. First, for every $b \in \Sigma$ and $m \in \{\text{out}, \text{in}\}$, let A_m^b be the NFA accepting $d_m(b)$.

For every v in rhs(q, a), let s_v be concatenation of the labels of the children of v. Define the 2NFA N_v as follows: suppose s_v is of the form $z_0 p_1 z_1 \cdots p_\ell z_\ell$ where $z_i \in \Sigma^*$ and $p_i \in Q_T$, then $a_1 \cdots a_n \in L(N_v)$ if and only if

$$z_0 p_1(a_1 \cdots a_n) z_1 \cdots p_\ell(a_1 \cdots a_n) z_\ell \in L(A_{\text{out}}^{\text{lab}^h(v)}).$$

As s_v is fixed, N_v can recognize this language by reading $a_1 \cdots a_n$ ℓ times while simulating $A_{\text{out}}^{\text{lab}^h(v)}$. More precisely, the automaton simulates $A_{\text{out}}^{\text{lab}^h(v)}$ on $z_{i-1} p_i(a_1 \cdots a_n)$ on the $(i + 1)$th pass. Note that N_v does not loop.

It remains to describe the construction of the 2AFA$^{\text{lf}}$ A. On input $a_1 \cdots a_n$, A first checks whether $a_1 \cdots a_n \in L(A_{\text{in}}^a)$ by simulating A_{in}^a. Hereafter, A goes back to the beginning of the input string, guesses an internal node v in rhs(q, a) and simulates the complement of N_v. As N_v is a 2NFA that does not loop, A is a 2AFA$^{\text{lf}}$ whose size is linear in the size of the N_v's. This completes the construction of B. $\square$

The next result shows that typechecking remains PSPACE-hard even when NFAs are replaced by DFAs. The main source of complexity is the ability of transducers to make an arbitrary number of copies.

Theorem 12(4). *$TC[\mathcal{T}_{\rm nd}, DFA]$ is* PSPACE*-complete.*

Proof. The intersection emptiness problem of deterministic finite automata is stated as follows: given a sequence of DFAs $M_i = (Q_i, \Sigma, \delta_i, s_i, F_i)$, $i = 1, \ldots, n$, is $\bigcap_{i=1}^n L(M_i) = \emptyset$? This problem is known to be PSPACE-hard [12]. We define a transducer $T = (Q_T, \Sigma \cup \{\#_0, \ldots, \#_n\}, q_T^0, R_T)$ and two DTDs $d_{\rm in}$ and $d_{\rm out}$ such that T typechecks w.r.t. $d_{\rm in}$ and $d_{\rm out}$ iff $\bigcap_{i=1}^n L(M_i) = \emptyset$.

The DTD $d_{\rm in}$ has as start symbol s and defines a tree of depth one where the string formed by the children of the root is an arbitrary string in Σ^*. The transducer makes n copies of this string separated by the delimiters $\#_i$: $Q_T = \{q, q_T^0\}$ and R_T contains the rules $(q_T^0, s) \to s(\#_0 q \#_1 q \cdots \#_{n-1} q \#_n)$ and $(q, a) \to a$, for every $a \in \Sigma$. Finally, $d_{\rm out}$ defines a tree of depth one with start symbol s such that $d_{\rm out}(s) =$

$$\{\#_0 w_1 \#_1 w_2 \#_2 \cdots \#_{n-1} w_n \#_n \mid \exists j \in \{1, \ldots, n\} \text{ such that } M_j \text{ does not accept } w_j\}.$$

Clearly, $d_{\rm out}(s)$ can be represented by a DFA whose size is polynomial in the sizes of the M_i's. Indeed, the DFA just simulates every M_i on the string following $\#_{i-1}$ till it encounters $\#_i$. It verifies that at least one M_i rejects. $\square$

Next, we focus on $\mathcal{SL}$-expressions as right-hand sides of DTDs. The complexity drops to CONP. Lemmas 17 and 18 are stated and proven in the appendix.

Theorem 12(5). *$TC[\mathcal{T}_{\rm nd}, \mathcal{SL}]$ is* CONP*-complete.*

Proof. First, we prove the hardness result by a reduction from validity of propositional formulas which is known to be complete for CONP [23]. Let ϕ be a propositional formula over the variables $v_1, \ldots, v_n$. Set $\Sigma := \{a_1, \ldots, a_n\}$. Define $d_{\rm in}$ as the DTD with start symbol a_1 defining depth one trees where the string formed by the children of the root can be arbitrary. Intuitively, every string w is a truth assignment: v_i is true iff at least one a_i occurs in w. The transducer T is the identity, and $d_{\rm out}(a_1) = \phi'$ where ϕ' is the formula obtained from ϕ where every occurrence of v_i is replaced by $a_i^{\geqslant 1}$ for $i = 1, \ldots, n$. Clearly, this instance typechecks iff ϕ is valid.

Next, we prove the upper bound. Let $T = (Q_T, \Sigma, q_T^0, R_T)$ and let $(d_{\rm in}, s_{\rm in})$ and $(d_{\rm out}, s_{\rm out})$ be the input and output DTD respectively. We describe an NP algorithm guessing a counterexample. In brief, we would like to guess an input tree t satisfying $d_{\rm in}$, a node $v \in \mathrm{Dom}(t)$ labeled with a and a state $q \in Q_T$ in which v is processed such that $T^q(a(w))$ does not satisfy $d_{\rm out}$. Here, w is the string obtained by concatenating the labels of the children of v. An immediate problem is that we cannot simply guess a whole tree t as the size of the latter might be exponential in the size of $d_{\rm in}$. Therefore, we simply guess a path ending in v which can be extended to a tree satisfying $d_{\rm in}$ and a string of children w with the desired property. We explain this next.

First, we introduce some notation. For a DTD (d, s_d) and $a \in \Sigma$, we denote by d^a the DTD d with start symbol a, that is, (d, a). Let k be the largest number occurring in any $\mathcal{SL}$-formula in $d_{\rm in}$ or $d_{\rm out}$. Set $r := (k + 1) \cdot |\Sigma|$.

The algorithm consists of three main parts:

(1) First, we sequentially guess a subset D of the derivable symbols $\{b \in \Sigma \mid L(d_{\text{in}}^b) \neq \emptyset\}$.

(2) Next, we guess a path of a tree in d_{in}. In particular, we guess a sequence of pairs $(a_i, q_i) \in D \times Q_T$, $i = 0, \ldots, m$, with $m \leqslant |\Sigma| \cdot |Q_T|$, such that
  (a) $a_0 = s_{\text{in}}$ and $q_0 = q_T^0$;
  (b) there is a tree $t \in L(d_{\text{in}})$ and a node $v \in \text{Dom}(t)$ such that $a_0 \cdots a_m$ is the concatenation of the labels of the nodes on the path from the root to v; and,
  (c) for all $i = 0, \ldots, m$: T visits a_i in state q_i.

(3) Finally, we guess a string $w \in D^*$ of length at most r such that $T^{q_m}(a_m(w))$ does not partly satisfy d_{out}. As r can be exponentially large, we do not guess w itself, but a representation of w. Here, partly satisfaction is as defined in the proof of Theorem 12(3).

We describe in detail how the three parts can be implemented and show that the verification of the guesses can be done in PTIME. As all the guesses can be done at the beginning, we obtain an NP algorithm.

(1) We compute D as follows.
  (a) *Guessing phase*: guess a sequence of different symbols $b_1, \ldots, b_{m'}$ in Σ. So, $m' \leqslant |\Sigma|$. Guess vectors $v_1, \ldots, v_{m'}$ where each $v_i = (\ell_1^i, \ldots, \ell_{i-1}^i) \in \{0, \ldots, k+1\}^{i-1}$. Intuitively, the vector v_i corresponds to the string $b_1^{\ell_1^i} \cdots b_{i-1}^{\ell_{i-1}^i}$. So, we interchangeably talk about the vector and the string v_i. Note that some ℓ_j^i may be zero.
  (b) *Checking phase*: For each $i = 1, \ldots, m'$, test that the string v_i satisfies $d_{\text{in}}(b_i)$. Note that this can be done in PTIME.

  Let $S_i = \{b_j \mid j \leqslant i\}$. From Lemma 17, it follows that if there is a string w in S_i^* such that w satisfies $d_{\text{in}}(b_i)$ then there is one such that each symbol occurs at most $k+1$ times. Hence, it suffices to guess vectors in $\{0, \ldots, k+1\}^{i-1}$. Finally, a simple induction shows that $D \subseteq \{b \in \Sigma \mid L(d_{\text{in}}^b) \neq \emptyset\}$.

(2) The requirement (a) can easily be checked. (c) can be checked by verifying that $q_{i+1} \in \text{rhs}(q_i, a_i)$ for all i. Let $D = \{b_1, \ldots, b_{|D|}\}$. To test (b), it suffices to guess a vector $v_i = (\ell_1, \ldots, \ell_{|D|}) \in \{0, \ldots, k+1\}^{|D|}$ for every $i \in \{0, \ldots, m-1\}$ such that $\ell_j \neq 0$ when $a_{i+1} = b_j$ and test whether $b_1^{\ell_1} \cdots b_{|D|}^{\ell_{|D|}}$ satisfies $d_{\text{in}}(a_i)$. As every symbol is in D, the path can be expanded to a tree satisfying d_{in}. By Lemma 17, it follows that guessing vectors of that size suffices. The upper bound $|\Sigma| \cdot |Q_T|$ on m can be obtained by a simple pumping argument.

(3) Before we describe the last part of the algorithm, we make the link explicit between the transducer T, the function f and the c's described in Lemma 18. We start with some notation. Let q be a state of T and $a \in \Sigma$ then define $q(a) := \text{top}(\text{rhs}(q, a))$. For a string $w = a_1 \cdots a_n$, we define $q(w) := q(a_1) \cdots q(a_n)$. For $a \in \Sigma$ and $w \in \Sigma^*$, we also define $\#_a(w)$ to be the number of a's occurring in w. Let $q \in Q_T$, $a \in \Sigma$ and let u be a node in $\text{rhs}(q, a)$. Let $z = z_0 p_1 z_1 \cdots p_\ell z_\ell$ be the concatenation of the labels of the children of u, such that $p_i \in Q_T$ and $z_i \in \Sigma^*$. For every $s \in \Sigma^*$, define $f_u^{q,a}(s)$ as the string obtained from z by replacing every p_i by the string $p_i(s)$. Now, we define the c's corresponding to $f_u^{q,a}(s)$. For every $b \in \Sigma$, set $c^b := \#_b(z)$ and for every $e \in \Sigma$, set $c_e^b := \sum_{j=1}^{\ell} \#_b(p_j(e))$. Clearly, for every $b \in \Sigma$ and every $s \in \Sigma^*$, $\#_b(f_u^{q,a}(s)) = c^b + \sum_{e \in \Sigma}(c_e^b \cdot \#_e(s))$.

So, the algorithm guesses a node u in rhs(q_m, a_m). We do not guess a string w but rather a vector in $\{1, \dots, k+1\}^{|\Sigma|}$ representing such a string (as in the previous bullets). We check whether $f_u^{q_m,a_m}(w)$ does not satisfy $d_{\text{out}}(a)$ where the label of u is a. Take f as $f_u^{q_m,a_m}$, ϕ_1 as $d_{\text{in}}(a_m)$, and ϕ_2 as $d_{\text{out}}(a)$. Then from Lemma 18, it follows that it suffices to guess a string represented by a vector in $\{1, \dots, k+1\}^{|\Sigma|}$. This completes the description of the algorithm. □

5. Transducers of bounded width

As can be inferred from Theorem 12, disallowing deletion lowers the complexity of the typechecking problem in the presence of DTDs. Unfortunately, the problem still remains intractable. In the context of DTD(DFA)s, the high complexity is a consequence of the copying power of transducers (cf. the proof of Theorem 12(4)). Therefore, we bound in advance the width of transducers by only considering transducers in the class $\mathcal{BW}_k$ for a fixed k (cf. Section 2.5). In the case of DTD(DFA)s we then finally obtain a tractable scenario.

Theorem 14. (1) *$TC[\mathcal{BW}_k, NTA]$ is* EXPTIME*-complete*;
(2) *$TC[\mathcal{BW}_k, NFA]$ is* PSPACE*-complete*;
(3) *$TC[\mathcal{BW}_k, DFA]$ is* PTIME*-complete*;
(4) *$TC[\mathcal{BW}_k, \mathcal{SL}]$ is* CONP*-complete.*

The lower bounds of (1), (2), and (4) follow immediately from the construction in the proofs of Theorem 12(1), (3), and (5).

Theorem 14(3). *$TC[\mathcal{BW}_k, DFA]$ is* PTIME*-complete.*

Proof. A PTIME lower bound is obtained by a reduction from PATH SYSTEMS [9]. PATH SYSTEMS is the following decision problem. Given a set P, a set $A \subseteq P$ of axiomas, a set $R \subseteq P^3$ of inference rules and some $p \in P$, is p provable from A using R? Let p be the start symbol of d_{in}. Further, for every $(a, b, c) \in R$, $d_{\text{in}}(c) = \{ab\}$; for every $a \in A$, $d_{\text{in}}(a) = \{\varepsilon\}$. Let $L(d_{\text{out}})$ be empty and let T be the transducer that copies the input tree. Then T typechecks w.r.t. d_{in} and d_{out} iff p has no proof.

In the proof of Theorem 12(3), TC[$\mathcal{T}_{\text{nd}}$,NFA] is reduced to the emptiness of NTA(2AFA$^{\text{lf}}$)s. In that proof, alternation was needed to express negation of NFAs; two-wayness was needed because T could make arbitrary copies of the input tree. However, when transducers can make only a bounded number of copies and DFAs are used, TC[$\mathcal{BW}_k$,DFA] can be LOGSPACE-reduced to emptiness of NTA(NFA)s. From Theorem 19(1), it then follows that TC[$\mathcal{BW}_k$,DFA] is in PTIME. □

6. Ranked versus unranked

We briefly motivate why we use unranked transducers rather than their more deeply studied ranked counterparts.

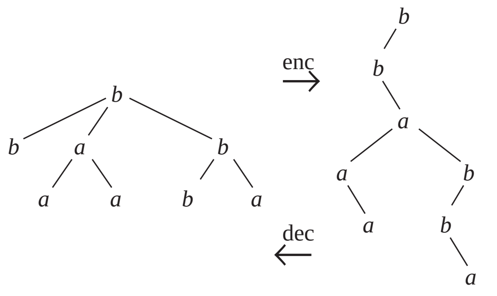


Fig. 5. An unranked tree and its binary encoding.

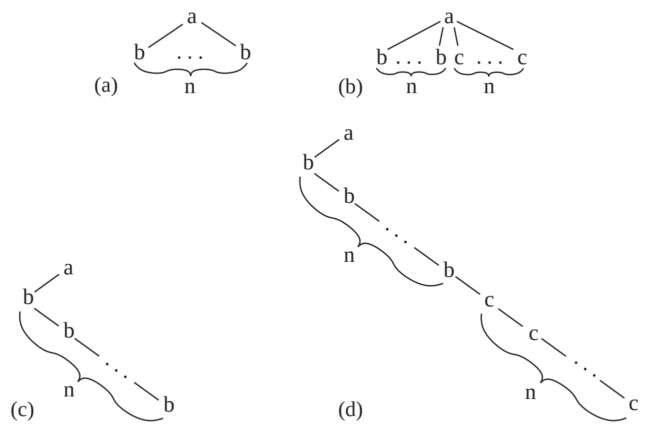


Fig. 6. (a) and (b) are unranked trees. (c) and (d) are their binary encodings respectively.

It is known that unranked trees can be uniformly encoded as binary trees. However, we argue that unranked tree transducers cannot be simulated by deterministic top-down ranked tree transducers on binary trees using the standard encoding. As mentioned in the introduction, macro tree transducers can simulate our transducers on the binary encodings [11,16], but as very little is known about their complexity this observation is not of much help.

For an illustration of the standard encoding, see, e.g., Fig. 5. The encoding is denoted by *enc* and the decoding by *dec*. Intuitively, the first child of a node remains the first child of that node in the encoding, but it is explicitly encoded as a left child. The remaining children are right descendants of the first child. Note that we allow a node to have a right child without having a left child, but this issue can easily be resolved by inserting dummy symbols in the encoding.

A formal definition of deterministic top-down ranked tree transducers can be found in [13]. In Fig. 6, we show two tree languages (n is arbitrary) and their binary encodings. Let L_1, L_2, L_3 and L_4 be the tree languages represented by the trees in Figs. 6(a)–6(d), respectively.

The language L_1 can be transformed to L_2 by the tree transducer $T = (Q, \Sigma, q^0, R)$ where $Q = \{q^0, q^b, q^c\}$, $\Sigma = \{a, b, c\}$, and R contains the rules

$$(q^0, a) \rightarrow a(q^b q^c) \qquad (q^b, b) \rightarrow b \qquad (q^c, b) \rightarrow c.$$

Basically, b^n is transformed to $b^n c^n$. However, as we argue next, L_3 cannot be transformed to L_4 by a deterministic top-down ranked tree transducer. For a tree t, let path(t) be the set of all strings formed by concatenating the labels of a path in t from the root to a leaf. For a tree language L, define the string language path$(L) = \{\text{path}(t) \mid t \in L\}$. Given a regular tree language L and a deterministic top-down ranked tree transducer R, the language path$(R(L))$, where $R(L) = \{R(t) \mid t \in L\}$, is regular [13, Corollary 20.13]. Since path$(L_4) = \{ab^n c^n \mid n \geqslant 1\}$ and L_3 is a regular tree language, L_4 cannot be the result of applying a deterministic top-down ranked tree transducer to L_3.

7. Conclusion

Motivated by simple transformations obtained by using structural recursion or XSLT, we studied typechecking for top-down XML transformers in the presence of both DTDs and tree automata. In this setting the complexity of the typechecking problem ranges from PTIME to EXPTIME. In particular, when tree automata are used in specifying schema languages, there is no hope for tractable algorithms. Indeed, in all considered scenarios, the typechecking problem remains EXPTIME-hard. The situation differs when we look at DTDs. We identified three sources of complexity: (1) deletion; (2) unbounded copying; and, (3) non-determinism in schema languages. Hence, we only obtained a PTIME typechecking algorithm when no deletion is allowed, the amount of copying is fixed in advance, and when DTD(DFA) are used to represent schemas.

Though the presented results shed some light on precisely which features determine the complexity of typechecking, it fails to identify relevant fragments for which typechecking is tractable. Indeed, although it makes sense to limit copying in advance, disallowing deleting completely is not very sensible as deleting occurs in many simple transformations (cf. Example 7).

Establishing tractable and practically relevant fragments is the topic of a subsequent paper [17]. Building further on the results of this paper, we obtain relevant tractable scenarios by enforcing combined restrictions on the deleting and copying power of transducers and by considering restricted DTDs. We also incorporate XPath expressions. As a byproduct of our new results we obtain that the complexity of TC[$\mathcal{BW}_k$,DTA] is EXPTIME-hard.

Appendix A.

A.1. Alternating string automata

We discuss two-way alternating string automata [14]. To prevent automata falling off the input string, we use delimiters $\triangleright$ and $\triangleleft$ not occurring in Σ. By $\Sigma_{\triangleright\triangleleft}$ we denote $\Sigma \cup \{\triangleright, \triangleleft\}$.

We tacitly assume that $\triangleright$ and $\triangleleft$ only occur on the left and right end of the string, respectively.

Definition 15. A *two-way alternating automaton* is a tuple $A = (Q, \Sigma_{\triangleright\triangleleft}, \delta, I, F, r, U)$ where

- Q is a finite set of states;
- I, F, U are subsets of Q and are the sets of initial, final and universal states, respectively;
- $r \in Q \setminus F$ is the rejecting state;
- $\delta : Q \times \Sigma_{\triangleright\triangleleft} \to 2^{Q\times\{\leftarrow,-,\rightarrow\}}$ is the transition function.

A *configuration* of A on a string $w = \triangleright w_2 \cdots w_{n-1}\triangleleft$ is a pair (j, q), where $j \in \mathrm{Dom}(w)$ and $q \in Q$. Intuitively, j is the current tape position and q is the current state. A configuration (j, q) is *initial* (*accepting*) if $q \in I$ ($q \in F$) and $j = 1$ ($j = |w|$). A configuration (j, q) is *universal* (*existential*) if $q \in U$ ($q \in Q - U$). Given $\gamma = (j, q)$ and $\gamma' = (j', q')$, we define the *step*-relation $\vdash$ on configurations as follows: $\gamma\vdash\gamma'$ iff $(q', d) \in \delta(q, a)$, $\mathrm{lab}^w(j) = a$, and $j' = j - 1$, $j' = j$, or $j' = j + 1$ iff $d = \leftarrow$, $d = -$, or $d = \rightarrow$, respectively. We assume that an automaton never attempts to move to the left (right) of a delimiter $\triangleright$ ($\triangleleft$). Further, we assume that A only reaches a final state at the delimiter $\triangleleft$ and that a computation branch of A only rejects by reaching r at the delimiter $\triangleleft$. Note that because of this last convention, the transition function of a two-way alternating finite automaton is complete, that is, for all $a \in \Sigma \cup \{\triangleright\}$, $q \in Q$, $\delta(q, a) \neq \emptyset$ and for all $q \in Q \setminus (\{r\} \cup F)$, $\delta(q, \triangleleft) \neq \emptyset$. For a configuration γ, a γ-*run* of A on a string w is a (possibly infinite) tree where nodes are labeled with configurations as follows:

(1) the root is labeled with γ;
(2) every inner node labeled with an existential configuration γ has exactly one child γ' and $\gamma\vdash\gamma'$; and,
(3) let for any universal configuration γ, $\{\gamma_1, \ldots, \gamma_m\} := \{\gamma' \mid \gamma \vdash \gamma'\}$, then every inner node labeled with γ has exactly m children labeled $\gamma_1, \ldots, \gamma_m$.

An *accepting* γ-run is a γ-run which does not contain an infinite path and where every leaf node is labeled with an accepting configuration. A *run* is a γ-run where γ is an initial configuration. The language accepted by A is defined as $L(A) := \{w \in \Sigma^* \mid$ there is an accepting run of A on $\triangleright w\triangleleft\}$. The *size* of a A is $|\Sigma| + |Q| + \sum_{q\in Q, a\in\Sigma} |\delta(q, a)|$.

We denote by 2AFA the class of all two-way alternating finite automata. We say that A *loops* on w if there is a run on w which contains an infinite path. An automaton is then *loop-free* when it never loops. We denote the class of loop-free two-way alternating finite automata by $2\mathrm{AFA}^{\mathrm{lf}}$. Note that $2\mathrm{AFA}^{\mathrm{lf}}$ accept only regular string languages [14]. A two-way non-deterministic automaton, denoted 2NFA, is a 2AFA where $U = \emptyset$.

The construction in the next lemma is a slight adaptation of a construction from Vardi [32]. In Theorem 19, we use an on-the-fly construction of the automaton N constructed in this proof. Although the lemma appears in the literature without a restriction to loop-free automata [10], it is not clear how to adapt it to an on-the-fly algorithm.

Lemma 16. *Let A be an $2AFA^{lf}$, then there exists an NFA N whose size is exponential in the size of A such that $L(N) = L(A)$.*

Proof. Let $A = (Q_A, \Sigma_{\rhd\lhd}, \delta_A, I_A, F_A, r_A, U_A)$ be an 2AFA$^{\mathrm{lf}}$. We construct an NFA $N = (Q_N, \Sigma_{\rhd\lhd}, \delta_N, I_N, F_N)$ with $Q_N = (2^{Q_A} \times 2^{Q_A})$, $I_N = \{(\emptyset, U) \mid U \cap I_A \neq \emptyset\}$, $F_N = \{(U, \emptyset) \mid U \cap F_A \neq \emptyset$ and $r_A \notin U\}$. For ease of exposition, N also operates over delimited strings. Intuitively, when N is in state (U, V) when processing the jth symbol of input $w = w_1 \cdots w_n$, then for every state $p \in V$, A must accept $w_1 \cdots w_n$ when started in p on position j. Note that $w_1 = \rhd$ and $w_n = \lhd$. The set U is the set V of position $j - 1$. Initial and final states are of the form $(\emptyset, U)$ and $(U, \emptyset)$ as the two-way automaton cannot move past the left and right delimiter, respectively.

The transition function is defined as follows. For every $(U, V), (T, U) \in Q_N$ and $a \in \Sigma_{\rhd\lhd}$, $(U, V) \in \delta_A((T, U), a)$ iff for every p in $U - F_A$ the following holds:

- if p is an existential state then there exists a pair $(p', d') \in \delta_A(p, a)$ such that $p' \in T$ if $d' = \leftarrow$, $p' \in U$ if $d' = -$, and $p' \in V$ if $d' = \rightarrow$; and,
- if p is a universal state then for all pairs $(p', d') \in \delta(p, a)$, $p' \in T$ if $d' = \leftarrow$, $p' \in U$ if $d' = -$, and $p' \in V$ if $d' = \rightarrow$.

Clearly, the size of N is exponential in the size of A. It remains to show that $L(A) = L(N)$. Clearly, $\rhd\varepsilon\lhd \in L(A)$ iff $\rhd\varepsilon\lhd \in L(N)$. Therefore, let $w = \rhd w_1 \cdots w_n \lhd$ for $n > 0$. Suppose that there is an accepting run r of A on input w. Define $Q_0 = \emptyset$, $Q_i = \{p \mid (i, p)$ is a label in $r\}$ for $i = 1, \ldots, n + 1$, $Q_{n+2} = \{p \mid (n + 2, p)$ is a leaf label in $r\}$, and $Q_{n+3} = \emptyset$. It is easy to check that $(Q_0, Q_1) \in I_N$ and ρ is an accepting run for N on w where $\rho(i) = (Q_i, Q_{i+1})$ for $i = 1, \ldots, n + 2$.

For the other direction, suppose ρ is an accepting run of N on w. Then, let $(Q_i, Q_{i+1}) = \rho(i)$ for every $i \in \mathrm{Dom}(w)$. For $i \in \mathrm{Dom}(w)$, define $m_d(i)$ as $i - 1$, i, and $i + 1$, when d is $\leftarrow, -, \rightarrow$, respectively. We define the depth of a configuration (i, q) where $q \in Q_i$, denoted $\mathrm{depth}(i, q)$, inductively as follows: if $q \in F_A$ then $\mathrm{depth}(i, q) = 0$; otherwise, $\mathrm{depth}(i, q)$ is

$$\max\{\mathrm{depth}(j, q') + 1 \mid (q', d) \in \delta_A(q, \mathrm{lab}^w(i)), q' \in Q_j \text{ and } m_d(i) = j\}.$$

As A does not loop this notion is well-defined. By induction on the depth of configurations $\gamma = (i, q)$, it is easy to construct an accepting γ-run of height $\mathrm{depth}(i, q)$. The claim then follows for an initial configuration $(1, q)$ with $q \in Q_1 \cap I_A$.

When a 2AFA is not loop-free, then the $\mathrm{depth}(i, q)$ is not well-defined for all strings, and the construction of a run for the 2AFA from a run of the NFA might lead to an infinite tree. □

A.2. Unordered string languages

By $\#_x(y)$ we denote the number of x's occurring in y for $x \in \Sigma$ and $y \in \Sigma^*$. The following lemma is a useful tool in proving results about $\mathcal{SL}$.

Lemma 17. *Let ϕ be an $\mathcal{SL}$-formula and let k be the largest integer occurring in ϕ. Let $s, s' \in \Sigma^*$ be as follows*:

- *if $\#_a(s) > k$ then $\#_a(s') > k$;*
- *otherwise, $\#_a(s) = \#_a(s')$.*

Then $s \models \phi$ iff $s' \models \phi$.

```
$R_1 := \{q \in Q \mid \exists a \in \Sigma, \varepsilon \in \delta(q,a)\};$
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. 7. Computing the set R of reachable states.

Proof. We can assume that negations in ϕ only occur in front of atomic formulas. We call an atomic $\mathcal{SL}$-formula or a negation of an atomic $\mathcal{SL}$-formula a *literal*.

To prove the lemma, simply observe that for each $a \in \Sigma$, such that $\#_a(s) > k$, s satisfies all literals of the form $a^{\geqslant i}$ and $\neg a^{=j}$ and s violates all literals of the form $\neg a^{\geqslant i}$ and $a^{=j}$ where $i, j \in \{0, \ldots, k\}$. The same holds for s'. $\square$

We make use of the next lemma in the proof of Theorem 12(5).

Lemma 18. *Let ϕ_1 and ϕ_2 be $\mathcal{SL}$-formulas and let k be the largest integer occurring in ϕ_1 or ϕ_2. Let $f : \Sigma^* \to \Sigma^*$ be a function so that for every $b \in \Sigma$ there exists a fixed sequence of natural numbers c^b, $(c^b_a)_{a\in\Sigma}$ for which $\#_b(f(s)) = c^b + \sum_{a\in\Sigma}(c^b_a \times \#_a(s))$ for every $s \in \Sigma^*$. If there is a string $s \vDash \phi_1$ then there is a string $s' \in \Sigma^*$ such that*

- *$s' \vDash \phi_1$*
- *$f(s') \vDash \phi_2$ iff $f(s) \vDash \phi_2$, and*
- *each symbol occurs maximally $k+1$ times in s'.*

Proof. Intuitively, the function f characterizes the effect of our tree transformations on a string of siblings in the input tree. Let s be a string such that $s \vDash \phi_1$ and there exists a symbol that occurs more than $k+1$ times in s. We construct s' from s by deleting x arbitrary occurrences of every symbol a that occurs $k+1+x$ times in s. So, $k+1$ occurrences remain. Since by Lemma 17, $s' \vDash \phi_1$, we only need to show that $f(s') \vDash \phi_2$ iff $f(s) \vDash \phi_2$. Therefore, take an arbitrary symbol $b \in \Sigma$. Then $\#_b(f(s)) = c^b + \sum_{a\in\Sigma}(c^b_a \cdot \#_a(s))$. If for all $a \in \Sigma$ that occur more than k times in s, $c^b_a = 0$, then $\#_b(f(s)) = \#_b(f(s'))$. If this is not the case, take $a \in \Sigma$ that occurs more than k times in s and $c^b_a \neq 0$. Then $\#_b(f(s)) \geqslant \#_a(s) > k$ and $\#_b(f(s')) \geqslant \#_a(s') > k$, so, according to Lemma 17, $f(s) \vDash \phi_2$ iff $f(s') \vDash \phi_2$. $\square$

A.3. Complexity of tree automata

We prove the following theorem which is a useful tool for obtaining upper bounds on the complexity of the typechecking problem.

Theorem 19. (1) *Emptiness of NTA(NFA) is in* PTIME;
(2) *Emptiness of NTA($2AFA^{lf}$) is in* PSPACE.

Proof. (1) Let $B = (Q, \Sigma, \delta, F)$ be an NTA(NFA). The algorithm in Fig. 7 computes the set of reachable states $R := \{q \mid \exists t \in \mathcal{T}_\Sigma : q \in \delta^*(t)\}$ in a bottom-up manner. Clearly, $L(B) = \emptyset$ iff $R \cap F = \emptyset$. Note that $R_i \subseteq R_{i+1}$ and $R_1 = \{\delta^*(a) \mid a \in \Sigma\}$. We argue that the algorithm is in PTIME. Clearly, R_1 can be computed in PTIME. Further, the for-loop makes a linear number of iterations. Every iteration is a linear number of non-emptiness tests of the intersection of an NFA with R_{i-1}^* where $R_{i-1} \subseteq Q$. Clearly, the latter is in PTIME.

(2) From the proof of Theorem 19(1), it follows that emptiness of an NTA can be reduced to a polynomial number of tests of the following form:

(i) $\varepsilon \in \delta(q, a)$; and,

(ii) $\delta(q, a) \cap R_{i-1}^* \neq \emptyset$.

We show that when $\delta(q, a)$ is represented by a 2AFA$^{\mathrm{lf}}$, both tests can be done in PSPACE.

Let $B = (Q, \Sigma, \delta, F)$ be an NTA(2AFA$^{\mathrm{lf}}$) and let for every $q \in Q$ and $a \in \Sigma$, $A^{q,a} = (Q^{q,a}, Q_{\rhd\lhd}, \delta^{q,a}, I^{q,a}, F^{q,a}, r^{q,a}, U^{q,a})$ be the 2AFA$^{\mathrm{lf}}$ representing $\delta(q, a)$. Denote by $N^{q,a}$ the NFA equivalent to $A^{q,a}$ given by the construction of Lemma 16. Of course, we cannot construct $N^{q,a}$ in polynomial space as it is exponentially bigger than $A^{q,a}$. Therefore, we will construct $N^{q,a}$ on the fly. We denote the transition function of $N^{q,a}$ by $\delta_N^{q,a}$.

We first argue that given a $b \in R_{i-1}^*$ and two states (T, U), (U, V) of $N^{q,a}$, we can check in PSPACE that $(U, V) \in \delta_N^{q,a}((T, U), b)$. Indeed, we just have to check for all elements $p \in U$ the constraints mentioned in Lemma 16. That is, if p is existential, we check that there is a $(p', d') \in \delta_N^{q,a}(p, b)$ such that $p' \in T$, $p' \in U$, or $p' \in V$ depending on d'. If p is a universal state, we have to verify that for all $(p', d') \in \delta_N^{q,a}(p, b)$, $p' \in T$, $p' \in U$, or $p' \in V$ depending on d'. These two tests merely involve set membership and require only constant space.

We first describe the algorithm to check (i). We need to check whether $\rhd\lhd$ is accepted by $N^{q,a}$. To this end, we guess states (T_1, U_1), (T_2, U_2), (T_3, U_3) such that the first state is an initial state; the last state is an accepting state; and, $(T_2, U_2) \in \delta_N^{q,a}((T_1, U_1), \rhd)$ and $(T_3, U_3) \in \delta_N^{q,a}((T_2, U_2), \lhd)$. By the previous discussion, the latter can be done in PSPACE.

Next, we describe the algorithm to check (ii). Given $R_{i-1} \subseteq Q$, $q \in Q$ and $a \in \Sigma$, we need to check whether $q \in R_i$. The latter reduces to verifying whether there is some string $b_1 \cdots b_n$ in R_{i-1}^* that is accepted by $A^{q,a}$ or, equivalently, $N^{q,a}$.

(1) *Initialization step*: We start by guessing an initial state (T, U) and a state (U, V) such that $(U, V) \in \delta_N^{q,a}((T, U), \rhd)$. We write the state (U, V) on the tape.

(2) *Iteration step*: Let (U, V) be the state written on the tape. We guess a state (U', V') such that $(U', V') \in \delta_N^{q,a}((U, V), \lhd)$. If (U', V') is final, then we know that $R_{i-1}^* \cap A^{q,a} \neq \emptyset$ and accept. Otherwise, we erase (U', V') and guess a symbol $b \in R_{i-1}$ and a state (U'', V'') such that $(U'', V'') \in \delta_N^{q,a}((U, V), b)$. We erase (U, V), write (U'', V'') on the tape and resume at the beginning of the iteration step.

Clearly, $R_{i-1}^* \cap A^{q,a} \neq \emptyset$ iff there is a run of the algorithm that accepts. Further, by the discussion above, the algorithm only uses polynomial space. □

References

[1] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufmann, Los Altos, CA, 1999.

[2] N. Alon, T. Milo, F. Neven, D. Suciu, V. Vianu, Typechecking XML views of relational databases, ACM Trans. Comput. Logic 4 (3) (2003) 315–354.

[3] N. Alon, T. Milo, F. Neven, D. Suciu, V. Vianu, XML with data values: typechecking revisited, J. Comput. System Sci. 66 (4) (2003) 688–727.

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

[5] A. Brüggemann-Klein, M. Murata, 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, D. Suciu, UnQl: a query language and algebra for semistructured data based on structural recursion, VLDB J. 9 (1) (2000) 76–110.

[7] J. Clark, XSL transformations version 1.0, ⟨http://www.w3.org/TR/WD-xslt⟩, (August 1999).

[8] World Wide Web Consortium, XML Schema, ⟨http://www.w3.org/XML/Schema⟩.

[9] S.A. Cook, An observation on time-storage trade-off, J. Comput. System Sci. 9 (3) (1974) 308–316.

[10] S.S. Cosmadakis, H. Gaifman, P.C. Kanellakis, M. Vardi, Decidable optimization problems for database logic programs, in: Proc. 20th Annu. ACM Sympos. on Theory of Computing, ACM Press, New York, 1988, pp. 477–490.

[11] J. Engelfriet, H. Vogler, Macro tree transducers, J. Comput. System Sci. (1985).

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

[13] F. Gécseg, M. Steinby, Tree languages, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Vol. 3, Springer, Berlin ,1997, pp. 1–68 (Chapter 1).

[14] R.E. Ladner, R.J. Lipton, L.J. Stockmeyer, Alternating pushdown and stack automata, SIAM J. Comput. 13 (1) (1984) 135–155.

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

[16] S. Maneth, F. Neven, Structured document transformations based on XSL, in: R. Connor, A. Mendelzon (Eds.), Research Issues in Structured and Semistructured Database Programming (DBPL'99), Lecture Notes in Computer Science, Vol. 1949, Springer, Berlin, 2000, pp. 79–96.

[17] W. Martens, F. Neven, Frontiers of tractability for typechecking simple XML transformations, in: Proc. 23rd Sympos. Principles of Database Systems (PODS 2004), 2004, pp. 23–34.

[18] T. Milo, D. Suciu, Type inference for queries on semistructured data, in: Proc. 18th ACM Sympos. Principles of Database Systems, ACM Press, New York, 1999, pp. 215–226.

[19] T. Milo, D. Suciu, V. Vianu, Typechecking for XML transformers, J. Comput. System Sci. 66 (1) (2003) 66 –97.

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

[21] F. Neven, T. Schwentick, XML schemas without order, Unpublished manuscript, 1999.

[22] F. Neven, T. Schwentick, Query automata on finite trees, Theoret. Comput. Sci. 275 (2002) 633–674.

[23] C. Papadimitriou, Computational Complexity, Addison-Wesley, Reading, MA, 1994.

[24] Y. Papakonstantinou, V. Vianu, DTD inference for views of XML data, in: Proc. 19th Sympos. on Principles of Database Systems (PODS 2000), ACM Press, New York, 2000, pp. 35–46.

[25] Y. Papakonstantinou, V. Vianu, Incremental validation of XML documents, in: Proc. 9th Internat. Conf. on Database Theory (ICDT 2003), Springer, Berlin, 2003, pp. 47–63.

[26] H. Seidl, Deciding equivalence of finite tree automata, SIAM J. Comput. 19 (3) (1990) 424–437.

[27] H. Seidl, Haskell overloading is DEXPTIME-complete, Inform. Process. Lett. 52 (2) (1994) 57–60.

[28] L.J. Stockmeyer, A.R. Meyer, Word problems requiring exponential time: preliminary report, in: Conf. Record of 5th Annu. ACM Sympos. on Theory of Computing, 1973, pp. 1–9.

[29] D. Suciu, Typechecking for semistructured data, in: Proc. 8th Workshop on Data Bases and Programming Languages (DBPL 2001), 2001.

[30] D. Suciu, The XML typechecking problem, SIGMOD Rec. 31 (1) (2002) 89–96.

[31] A. Tozawa, Towards static type checking for XSLT, in: Proceedings of the ACM Symposium on Document Engineering, 2001, pp. 18–27.

[32] M.Y. Vardi, A note on the reduction of two-way automata to one-way automata, Inform. Process. Lett. 30 (1989) 261–264.