Typechecking Top-Down XML Tranformations: Fixed Input or Output Schemas ⋆

Wim Martens [a,*], Frank Neven [b], and Marc Gyssens [b]

[a] *Technical University of Dortmund*
Germany

[b] *Hasselt University and*
Transnational University of Limburg,
Agoralaan, Gebouw D
B-3590 Diepenbeek, Belgium

Abstract

Typechecking consists of statically verifying whether the output of an XML transformation always conforms to an output type for documents satisfying a given input type. In this general setting, both the input and output schema as well as the transformation are part of the input for the problem. However, scenarios where the input or output schema can be considered to be fixed, are quite common in practice. In the present work, we investigate the computational complexity of the typechecking problem in the latter setting.

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

---

⋆ An extended abstract of a part of this paper appeared as Section 3 in reference [22] in the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 2004.

* Corresponding author.
Email addresses: wim.martens@udo.edu (Wim Martens), frank.neven@uhasselt.be (Frank Neven), marc.gyssens@uhasselt.be (Marc Gyssens).

1 Introduction

In a typical XML data exchange scenario on the web, a user community creates a common schema and agrees on producing only XML data conforming to that schema. This raises the issue of typechecking: verifying at compile time that every XML document which is the result of a specified query or document transformation applied to a valid input document satisfies the output schema [33,34].

The typechecking problem is determined by three parameters: the classes of allowed input and output schemas, and the class of XML-transformations. As typechecking quickly becomes intractable [2,23,26], we focus on simple but practical XML transformations where only little restructuring is needed, such as, for instance, in filtering of documents. In this connection, we think, for example, of transformations that can be expressed by structural recursion [8] or by a top-down fragment of XSLT [5]. As is customary, we abstract such transformations by unranked tree transducers [19,23]. As schemas, we adopt the usual Document Type Definitions (DTDs) and their robust extensions: regular tree languages [26,17] or, equivalently, specialized DTDs [29,3]. The latter serve as a formal model for XML Schema [31].

Our work should be contrasted with the work on general-purpose XML programming languages like XDuce [14] and CDuce [4] where the programmer adds redundant type annotation to facilitate typechecking. In our setting no types have to be given by the programmer to capture the behavior of the various rules constituting a translation.

The typechecking scenario outlined above is very general: both the schemas and the transducer are determined to be part of the input. However, for some exchange scenarios, it makes sense to consider the input and/or output schema to be fixed when transformations are always from within and/or to a specific community. Therefore, we revisit the various instances of the typechecking problem considered in [23] and determine the complexity in the presence of fixed input and/or output schemas. The main goal of this paper is to investigate to which extent the complexity of the typechecking problem is lowered in scenarios where the input and/or output schema is fixed. An overview of our results is presented in Table 2.

The remainder of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we provide the necessary definitions. In Section 4, we discuss typechecking in the restricted settings of fixed output and/or input schemas. The results are summarized in Table 2. We obtain several new cases for which typechecking is in polynomial time: *(i)* when the input schema is fixed and the schemas are DTDs with SL-expressions; *(ii)* when the output

schema is fixed and the schemas are DTDs with NFAs; and *(iii)* when both the input and output schemas are fixed and the schemas are DTDs using DFAs, NFAs, or SL-expressions. We conclude in Section 5.

2 Related Work

The research on typechecking XML transformations was initiated by Milo, Suciu, and Vianu [26]. 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 [26]. Interestingly, typechecking of k-pebble transducers has recently been related to typechecking of compositions of macro tree transducers [12]. Alon et al. [1,2] investigated typechecking in the presence of data values and show that the problem quickly turns undecidable. As our interest lies in formalisms with a more manageable complexity for the typechecking problem, we choose to work with XML transformations that are much less expressive than k-pebble transducers and that do not change or use data values in the process of transformation.

A problem related to typechecking is type inference [25,29]. 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, namely, checking containment of the inferred schema into the given one. However, characterizing output languages of transformations is quite hard [29]. For this reason, we adopt different techniques for obtaining complexity upper bounds for the typechecking problem.

The transducers considered in the present paper are restricted versions of the DTL-programs, 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). Recently, Maneth et al. considered the typechecking problem for an extension of DTL-programs and obtained that typechecking was still decidable [20]. Their typechecking algorithm, like the one of [26], is based on inverse type inference. That is, they compute the pre-image of all ill-formed output documents and test whether the intersection of the pre-image and the input schema is empty. Tozawa considered typechecking with respect to tree automata for a fragment of top-down XSLT [35]. He uses a more general framework, but he was not able to derive a bound better than double-exponential on the complexity of his algorithm.

Martens and Neven investigated polynomial time fragments of the typechecking problem by putting syntactical restrictions on the tree transducers, and

making them as general as possible [24]. Here, tractability of the typechecking problem is obtained by bounding the *deletion path width* of the tree transducers. The deletion path width is a notion that measures the number of times that a tree transducer copies part of its input. In particular, it also gives rise to tractable fragments of the typechecking problem where the transducer is allowed to delete in a limited manner.

3 Preliminaries

In this section we provide the necessary background on trees, automata, and tree transducers. 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 $\text{Dom}(w) = \{1, \ldots, n\}$. The length of w, denoted by $|w|$, is n. The label a_i of position i in w is denoted by $\text{lab}^w(i)$. 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 on N for a string $w \in \Sigma^*$ is a mapping from $\text{Dom}(w)$ to Q such that $\rho(1) \in \delta(q, \text{lab}^w(1))$ for $q \in I$, and for $i = 1, \ldots, |w| - 1$, $\rho(i+1) \in \delta(\rho(i), \text{lab}^w(i+1))$. 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 $a \in \Sigma$.

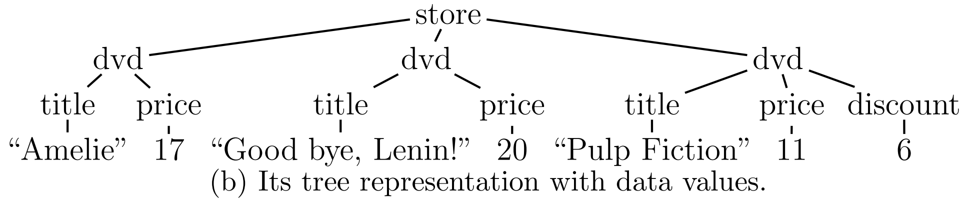
3.1 Trees and Hedges

It is common to view XML documents as finite trees with labels from a finite alphabet Σ. Figures 1(a) and 1(b) give an example of an XML document together with its tree representation. Of course, elements in XML documents can also contain references to nodes. But, as XML schema languages usually do not constrain these nor the data values at leaves, it is safe to view schemas as simply defining tree languages over a finite alphabet. In the rest of this section, we introduce the necessary background concerning XML schema languages.

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 ε in $\mathcal{T}_\Sigma$ and, 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 $a(t_1 \cdots t_n)$

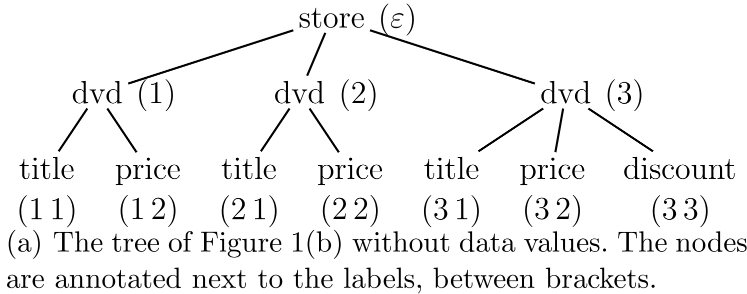
```
<store>
  <dvd>
    <title> "Amelie" </title>
    <price> 17 </price>
  </dvd>
  <dvd>
    <title> "Good bye, Lenin!" </title>
    <price> 20 </price>
  </dvd>
  <dvd>
    <title> "Pulp Fiction" </title>
    <price> 11 </price>
    <discount> 6 </discount>
  </dvd>
</store>
```

(a) An example XML document.

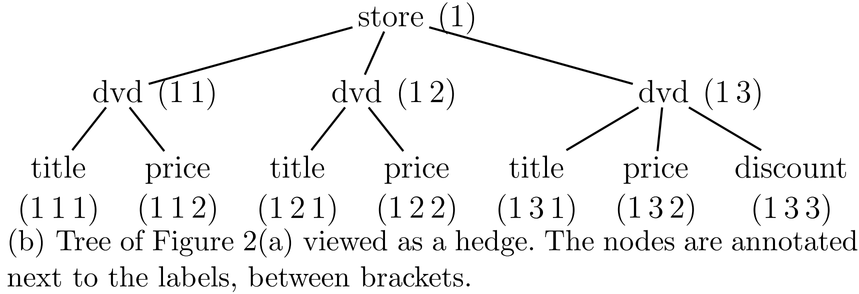


(b) Its tree representation with data values.

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



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



(b) Tree of Figure 2(a) viewed as a hedge. The nodes are annotated next to the labels, between brackets.

Fig. 2. The document of Figure 1 without data values, viewed as a tree and as a hedge.

where each t_i is a tree. When we write $a(t_1 \cdots t_n)$, we implicitly assume that each t_i is non-empty. 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 $\mathrm{Dom}(t)$, is the set defined as follows:

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

Figure 2(a) contains a tree in which we annotated the nodes between brackets. Observe that the n child nodes of a node u are always $u1, \cdots, un$, from left to right. The *label* of a node u in the tree $t = a(t_1 \cdots t_n)$, denoted by $\mathrm{lab}^t(u)$, is defined as follows:

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

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 $\mathrm{depth}(t) = \max\{\mathrm{depth}(t_i) \mid 1 \leq i \leq n\} + 1$. 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 $\mathrm{Dom}(h)$, is the subset of $\mathbb{N}^*$ defined as follows:

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

The *label* of a node $u = iu'$ in the hedge $h = t_1 \cdots t_n$, denoted by $\mathrm{lab}^h(u)$, is defined as $\mathrm{lab}^h(u) = \mathrm{lab}^{t_i}(u')$. 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 example: if the tree in Figure 2(a) were to represent a single-tree hedge, it would have the set of hedge-nodes $\{1, 11, 12, 13, 111, 112, 121, 122, 131, 132, 133\}$, as shown in Figure 2(b). 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, $\mathrm{lab}^{t_1}(\varepsilon) \cdots \mathrm{lab}^{t_n}(\varepsilon)$.

In the sequel, we adopt the following conventions: 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.

3.2 DTDs and Tree Automata

We use extended context-free grammars and tree automata to abstract from DTDs and the various proposals for XML schemas. We parameterize the definition of DTDs by a class of representations $\mathcal{M}$ of regular string languages such as, for instance, the class of DFAs (Deterministic Finite Automata) or NFAs (Non-deterministic Finite Automata). 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)* $\text{lab}^t(\varepsilon) = s_d$ and, *(ii)* for every $u \in \text{Dom}(t)$ with n children, $\text{lab}^t(u1) \cdots \text{lab}^t(un) \in L(d(\text{lab}^t(u)))$. We denote the set of trees satisfying d by $L(d)$.

Given a DTD d, we say that a Σ-symbol a *occurs in* $d(b)$ when there exist Σ-strings w_1 and w_2 such that $w_1 a w_2 \in L(d(b))$. We say that a *occurs in* d if a occurs in $d(b)$ for some $b \in \Sigma$.

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

Example 2 The following DTD (d, store) is satisfied by the tree in Figure 2(a):

$$\begin{aligned} d(\text{store}) &= \text{dvd dvd}^* \\ d(\text{dvd}) &= \text{title price (discount} + \varepsilon) \end{aligned}$$

This DTD defines the set of trees where the root is labeled with “store”; the children of “store” are all labeled with “dvd”; and every “dvd”-labeled node has a “title”, “price”, and an optional “discount” child.

In some cases, our algorithms are easier to explain on well-behaved DTDs as considered next. A DTD d is *reduced* if, for every symbol a that occurs in d, there exists a tree $t \in L(d)$ and a node $u \in \text{Dom}(t)$ such that $\text{lab}^t(u) = a$. Hence, for example, the DTD (d, a) where $d(a) = a$ is not reduced. Reducing a DTD(DFA) is in PTIME, while reducing a DTD(SL) is in CONP (see the Appendix, Corollary 30). Here, SL is a logic as defined next. Basically, reducing a DTD amounts to recursively deleting all symbols defining the empty language, and deleting all symbols that cannot be reached from the start symbol.

To define unordered languages, we make use of the specification language SL studied in [28] and also used in [1,2]. The syntax of this language is as follows:

Definition 3 *For every* $a \in \Sigma$ *and natural number* i*,* $a^{=i}$ *and* $a^{\geq i}$ *are* atomic SL-formulas*; "true" is also an atomic SL-formula. Every atomic SL-formula is an SL-formula and the negation, conjunction, and disjunction of SL-formulas are also SL-formulas.*

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

As an example, consider the SL-formula $\neg(\text{discount}^{\geq 1} \wedge \neg\text{price}^{\geq 1})$. This expresses the constraint that a discount can only occur when a price occurs. The *size* of an SL-formula is the number of symbols that occur in it, that is, Σ-symbols, logical symbols, and numbers (every i in $a^{=i}$ or $a^{\geq i}$ is written in binary notation).

We recall the definition of non-deterministic tree automata from [6]. We refer the unfamiliar reader to [27] for a gentle introduction. Such automata are sometimes also called 'hedge automata'.

Definition 4 *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* $q \in Q$*.*

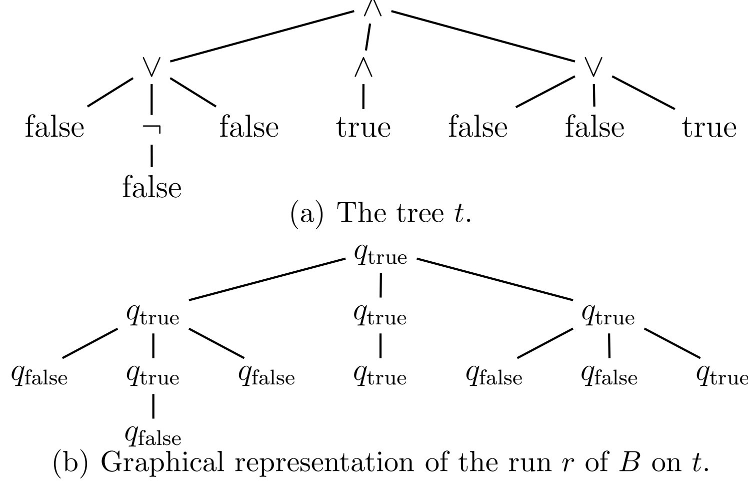
For simplicity, we often denote the regular languages in B's transition function by regular expressions.

A *run* of B on a tree t is a labeling $\lambda : \text{Dom}(t) \to Q$ such that, for every $v \in \text{Dom}(t)$ with n children, $\lambda(v1) \cdots \lambda(vn) \in \delta(\lambda(v), \text{lab}^t(v))$. Note that, when v has no children, the criterion reduces to $\varepsilon \in \delta(\lambda(v), \text{lab}^t(v))$. 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 $a \in \Sigma$, $\delta(q, a) \cap \delta(q', a) = \emptyset$. We denote the set of bottom-up deterministic NTAs by DTA.

Example 5 We give a bottom-up deterministic tree automaton $B = (Q, \Sigma,$

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



(a) The tree t.

(b) Graphical representation of the run r of B on t.

Fig. 3. Illustrations for Example 5.

$\delta, F)$ which accepts the parse trees of well-formed Boolean expressions that are true. Here, the alphabet Σ is $\{\wedge, \vee, \neg, \text{true}, \text{false}\}$. The states set Q contains the states q_{true} and q_{false}, and the accepting state set F is the singleton $\{q_{\text{true}}\}$. The transition function of B is defined as follows:

- $\delta(q_{\text{true}}, \text{true}) = \varepsilon$. We assign the state q_{true} to leafs with label "true".
- $\delta(q_{\text{false}}, \text{false}) = \varepsilon$. We assigns the state q_{false} to leafs with label "false".
- $\delta(q_{\text{true}}, \wedge) = q_{\text{true}} q_{\text{true}}^*$.
- $\delta(q_{\text{false}}, \wedge) = (q_{\text{true}} + q_{\text{false}})^* q_{\text{false}} (q_{\text{true}} + q_{\text{false}})^*$.
- $\delta(q_{\text{true}}, \vee) = (q_{\text{true}} + q_{\text{false}})^* q_{\text{true}} (q_{\text{true}} + q_{\text{false}})^*$.
- $\delta(q_{\text{false}}, \vee) = q_{\text{false}} q_{\text{false}}^*$.
- $\delta(q_{\text{true}}, \neg) = q_{\text{false}}$.
- $\delta(q_{\text{false}}, \neg) = q_{\text{true}}$.

Consider the tree t depicted in Figure 3(a). The unique accepting run r of B on t can be graphically represented as shown in Figure 3(b). Formally, the run of B on t is the function $\lambda : \text{Dom}(t) \to Q : u \mapsto \text{lab}^r(u)$ mapping a node of t to its label in r. Note that B is a DTA.

As 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 then is $|Q| + |\Sigma| + \sum_{q \in Q, a \in \Sigma} |\delta(q, a)|$. 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.

In our proofs, we will use reductions from the following decision problems for string automata:

Emptiness: Given an automaton A, is $L(A) = \emptyset$?

Universality: Given an automaton A, is $L(A) = \Sigma^*$?
Intersection emptiness: Given the automata $A_1, \ldots, A_n$, is $L(A_1) \cap \cdots \cap L(A_n) = \emptyset$?

The corresponding decision problems for tree automata are defined analogously.

In the Appendix, we show that the following statements hold over the alphabet $\{0, 1\}$ (Corollary 23). These proofs are only minor modifications of the unrestricted alphabet cases. We provide the proofs to make the paper self-contained.

(1) Intersection emptiness of an arbitrary number of DFAs is PSPACE-hard.
(2) Universality of NFAs is PSPACE-hard.

Over the alphabet $\{0, 1, 0', 1'\}$, the following statement holds:

(3) Intersection emptiness of an arbitrary number of TDBTAs is EXPTIME-hard.

TDBTA stands for top-down deterministic binary tree automaton. These are defined in Section 4.1.

3.3 Transducers

We adhere to transducers as a formal model for simple transformations corresponding to structural recursion [8] and a fragment of top-down XSLT.As in [26], the abstraction focuses on structure rather than on content. That is, our tree transducers only restructure trees. Their operation does not depend on the actual data values present in a tree. We next define the tree transducers used in this paper. To simplify notation, we restrict ourselves to one alphabet. That is, we consider transducers mapping Σ-trees to Σ-trees. [2]

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

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

only one tree with a Σ-symbol as its root label. 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. Also, notice that our transducers are not required to be total.

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.

Let q be a state of tree transducer T and $a \in \Sigma$. For a string $w = a_1 \cdots a_n$, we define $\mathrm{top}(T^q(w)) = \mathrm{top}(T^q(a_1)) \cdots \mathrm{top}(T^q(a_n))$.

We give an example of a tree transducer:

Example 7 Let $T = (Q, \Sigma, p, R)$ where $Q = \{p, q\}$, $\Sigma = \{a, b, c, d, e\}$, and R 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 4 (we assume the program is started in mode p).

A comparison with ordinary tree transducers is given in [23].

Example 8 Consider the tree t shown in Figure 5(a). In Figure 5(b) we give the translation of t by the transducer of Example 7. In order to keep the example simple, we did not list $T^q(\varepsilon)$ and $T^p(\varepsilon)$ explicitly in the process of translation.

We discuss two important features of tree transducers: *copying* and *deletion*.

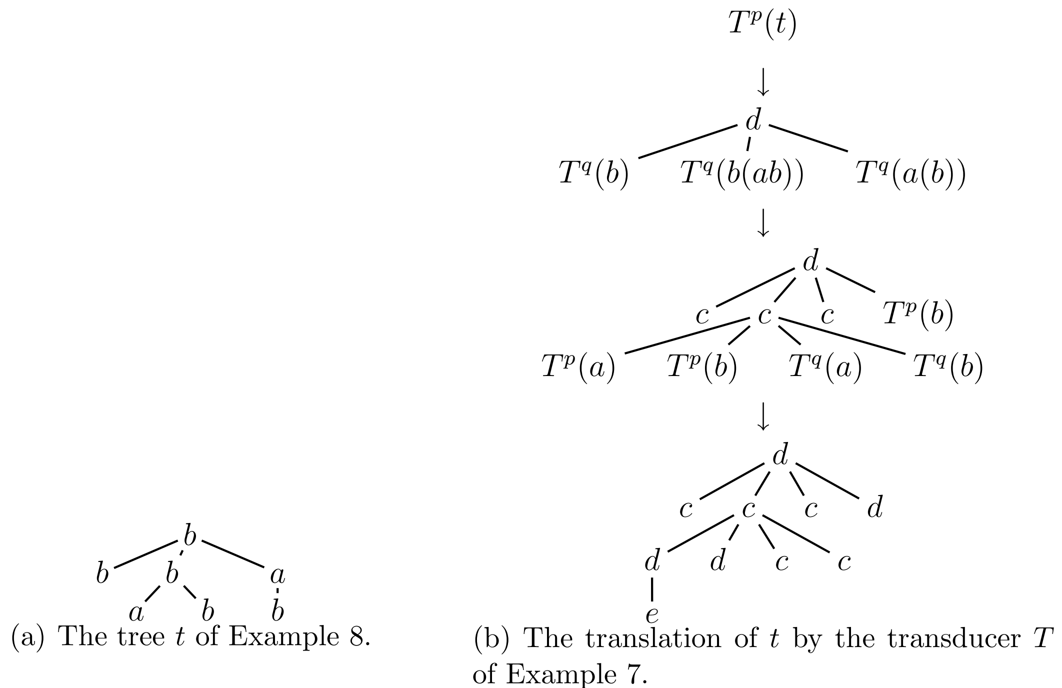
```
<xsl:template match="a" mode ="p">
  <d>
     <e/>
  </d>
</xsl:template>

<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. 4. The XSLT program equivalent to the transducer of Example 7.



(a) The tree t of Example 8.

(b) The translation of t by the transducer T of Example 7.

Fig. 5. A tree and its translation.

In Example 7, 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 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. For instance, $T^q(a(b)) = c\,d$ where d corresponds to b and not to a.

We define some relevant classes of transducers. A transducer is *non-deleting* if no states occur at the top-level of an rhs. We denote by $\mathcal{T}_{nd}$ the class of non-deleting transducers and by $\mathcal{T}_d$ the class of transducers where we allow deletion. Furthermore, a transducer T has *copying width* k if there are at most k occurrences of states in every sequence of siblings in an rhs. More formally, T has copying width k if, for each rhs r of T, and each sequence s of siblings [3] in r, only k nodes in s are labeled with some state of T. For instance, the transducer in Example 7 has copying width 2. Given a natural number k, which we will leave implicit, we denote by $\mathcal{T}_{bc}$ the class of transducers of copying width k. The abbreviation "bc" stands for *bounded copying*. We denote intersections of these classes by combining the indexes. For instance, $\mathcal{T}_{nd,bc}$ is the class of non-deleting transducers with bounded copying. When we want to emphasize that we also allow unbounded copying in a certain application, we write, for instance, $\mathcal{T}_{nd,uc}$ instead of $\mathcal{T}_{nd}$.

At this point, one may be tempted to think that $\mathcal{T}_{d,bc}$ transformations are at least as strong as $\mathcal{T}_{nd,uc}$ transformations. However, this is not the case in general, as the following abstract example shows. For each natural number n, there exists a transducer in $\mathcal{T}_{nd,uc}$ that transforms the input tree $a(b)$ into the output tree $a(b \cdots b)$, where the a-labeled node in the output has n b-labeled children. This, however, is not possible with $\mathcal{T}_{d,bc}$ transducers. Let k be the maximum copying width of the $\mathcal{T}_{d,bc}$ transducers. Then, after reading the root symbol of $a(b)$, a transducer in $\mathcal{T}_{d,bc}$ can either make up to k copies of the b-labeled node, or it can delete the b-labeled node and stop computation. Hence, such $\mathcal{T}_{d,bc}$ transducers cannot produce output trees $a(b \cdots b)$, where the a-labeled node has n b-labeled children, for arbitrarily large n.

3.4 The Typechecking Problem

Definition 9 *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 $t \in S_{in}$.*

[3] That is, all nodes in s have the same parent, or all nodes in s have no parent.

	NTA	DTA	DTD(NFA)	DTD(DFA)	DTD(SL)
d,uc	EXPTIME	EXPTIME	EXPTIME	EXPTIME	EXPTIME
nd,uc	EXPTIME	EXPTIME	PSPACE	PSPACE	coNP
nd,bc	EXPTIME	in EXPTIME PSPACE-hard	PSPACE	PTIME	coNP

Table 1
Results of [23] (upper and lower bounds). The top row shows the representation of the input and output schemas and the left column shows the class of tree transducer: "d", "nd", "uc", and "bc" stand for "deleting", "non-deleting", "unbounded copying", and "bounded copying" respectively.

We now define the problem central to this paper.

Definition 10 *Given S_{in}, S_{out}, and T, the* typechecking problem *consists in verifying whether T typechecks with respect to S_{in} and S_{out}.*

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 representation of a class of tree languages. Then $\mathrm{TC}[\mathcal{T}, \mathcal{S}]$ denotes the typechecking problem where $T \in \mathcal{T}$ and $S_{\mathrm{in}}, S_{\mathrm{out}} \in \mathcal{S}$. Examples of classes of tree languages are those defined by tree automata or DTDs. Classes of transducers are discussed in the previous section. The complexity of the problem is measured in terms of the sum of the sizes of the input and output schemas S_{in} and S_{out} and the transducer T.

Table 1 summarizes the results obtained in [23]. Unless specified otherwise, all problems are complete for the mentioned complexity classes. In the setting of [23], typechecking is only tractable when restricting to non-deleting and bounded copying transducers in the presence of DTDs with DFAs.

Recall that, in this article, we are interested in variants of the typechecking problem where the input and/or output schema is fixed. We therefore introduce some notations that are central to the paper. We denote the typechecking problem where the input schema, the output schema, or both are fixed by $\mathrm{TC}^{i}[\mathcal{T}, \mathcal{S}]$, $\mathrm{TC}^{o}[\mathcal{T}, \mathcal{S}]$, and $\mathrm{TC}^{io}[\mathcal{T}, \mathcal{S}]$, respectively. The complexity of these subproblems is measured in terms of the sum of the sizes of the input and output schemas S_{in} and S_{out}, and the transducer T, minus the size of the fixed schema(s).

4 Main Results

As argued in the Introduction, it makes sense to consider the input and/or output schema not as part of the input for some scenarios. From a complexity

fixed	TT	NTA	DTA	DTD(NFA)	DTD(DFA)	DTD(SL)
in, out, in+out	d,uc	**EXPTIME**	**EXPTIME**	**EXPTIME**	**EXPTIME**	**EXPTIME**
	d,bc	**EXPTIME**	**EXPTIME**	**EXPTIME**	**EXPTIME**	**EXPTIME**
in	nd,uc	**EXPTIME**	**EXPTIME**	**PSPACE**	**PSPACE**	in **PTIME**
	nd,bc	**EXPTIME**	**EXPTIME**	**PSPACE**	**NL**	in **PTIME**
out	nd,uc	**EXPTIME**	**EXPTIME**	**PSPACE**	**PSPACE**	coNP
	nd,bc	**EXPTIME**	**EXPTIME**	**PTIME**	PTIME	coNP
in+out	nd,uc	**EXPTIME**	**EXPTIME**	**NL**	**NL**	**NL**
	nd,bc	**EXPTIME**	**EXPTIME**	**NL**	**NL**	**NL**

Table 2
Complexities of the typechecking problem in the new setting (upper and lower bounds). The top row shows the representation of the input and output schemas, the leftmost column shows which schemas are fixed, and the second column to the left shows the class of tree transducer: “d”, “nd”, “uc”, and “bc” stand for “deleting”, “non-deleting”, “unbounded copying”, and “bounded copying” respectively. In the case of deleting transformations, the different possibilities are grouped as all complexities coincide.

theory point of view, it is important to note here that the input and/or output alphabet then also becomes fixed. In this article, we revisit the results of [23] from that perspective.

The results are summarized in Table 2. As some results already follow from proofs in [23], we printed the results requiring a new proof in bold. The entries where the complexity was lowered (assuming that the complexity classes in question are different) are underlined. Again, all problems are complete for the mentioned complexity classes unless specified otherwise.

We discuss the obtained results: for non-deleting transformations, we get three new tractable cases: *(i)* fixed input schema, *un*bounded copying, and DTD(SL)s; *(ii)* fixed output schema, bounded copying and DTD(NFA)s; and, *(iii)* fixed input and output, *un*bounded copying and all DTDs. It is striking, however, that in the presence of deletion or tree automata (even deterministic ones) typechecking remains EXPTIME-hard for *all* scenarios.

Mostly, we only needed to strengthen the lower bound proofs of [23].

4.1 Deletion: Fixed Input Schema, Fixed Output Schema, and Fixed Input and Output Schema

The EXPTIME upper bound for typechecking already follows from [23]. Therefore, it remains to show the lower bounds for $\mathrm{TC}^{io}[\mathcal{T}_{d,bc}$,DTD(DFA)] and $\mathrm{TC}^{io}[\mathcal{T}_{d,bc}$,DTD(SL)], which we do in Theorem 11. In fact, if follows from the proof that the lower bounds already hold for transducers with copying width 2.

We require the notion of top-down deterministic binary tree automata in the proof of Theorem 11. A *binary tree automaton* (BTA) is a non-deterministic tree automaton $B = (Q, \Sigma, \delta, F)$ operating on *binary trees*. These are trees where every node has zero, one, or two children. We assume that the alphabet is partitioned in *internal labels* and *leaf labels*. When a label a is an internal label, the regular language $\delta(q, a)$ only contains strings of length one or two. When a is a leaf label, the regular language $\delta(q, a)$ only contains the empty string. A binary tree automaton is *top-down deterministic* if *(i)* F is a singleton and, *(ii)* for every $q, q' \in Q$ with $q \neq q'$ and $a \in \Sigma$, $\delta(q, a)$ contains at most one string. We abbreviate "top-down deterministic binary tree automaton" by TDBTA.

Theorem 11 *(1)* $TC^{io}[\mathcal{T}_{d,bc}, DTD(DFA)]$ *is* EXPTIME*-complete; and (2)* $TC^{io}[\mathcal{T}_{d,bc}, DTD(SL)]$ *is* EXPTIME*-complete.*

PROOF. The EXPTIME upper bound follows from Theorem 11 in [23]. We proceed by proving the lower bounds.

We give a LOGSPACE reduction from the intersection emptiness problem of an arbitrary number of top-down deterministic binary tree automata (TDBTAs) over the alphabet $\Sigma = \{0, 1, 0', 1'\}$. The intersection emptiness problem of TDBTAs over alphabet $\{0, 1, 0', 1'\}$ is known to be EXPTIME-hard (cfr. Corollary 23(3) in the Appendix).

For $i = 1, \ldots, n$, let $A_i = (Q_i, \Sigma, \delta_i, \{\mathrm{start}_i\})$ be a TDBTA, with $\Sigma = \{0, 1, 0', 1'\}$. Without loss of generality, we can assume that the state sets Q_i are pairwise disjoint. We call 0 and 1 *internal labels* and $0'$ and $1'$ *leaf labels*. In our proof, we use the markers 'ℓ' and 'r' to denote that a certain node is a left or a right child. Formally, define $\Sigma_\ell := \{a_\ell \mid a \in \Sigma\}$ and $\Sigma_r := \{a_r \mid a \in \Sigma\}$. We use symbols from Σ_ℓ and Σ_r for the left and right children of nodes, respectively.

We now define a transducer T and two DTDs d_{in} and d_{out} such that $\bigcap_{i=1}^{n} L(A_i) = \emptyset$ if and only if T typechecks with respect to 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

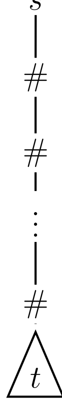


Fig. 6. Structure of the trees defined by the input schema in the proof of Theorem 11.

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. A bit of care needs to be taken, as a bounded copying transducer can not make an arbitrary number of copies of the input tree in the same rule. The transducer therefore goes through an initial copying phase where it repeatedly copies part of the input tree twice, until there are (at least) n copies. The transducer remains in the copying phase as long as it processes special symbols "#". The input trees are therefore of the form as depicted in Figure 6. In addition, the transducer should verify that the number of #-symbols in the input equals $\lceil \log n \rceil$.

The input DTD (d_{in}, s), which we will describe next, uses the alphabet $\Sigma_\ell \cup \Sigma_r \cup \{s, \#\}$, and defines all trees of the form as described in Figure 6, where s and # are alphabet symbols, and every internal node of t (which is depicted in Figure 6) has one or two children. When a node is an only child, it is labeled with an element of Σ_ℓ. Otherwise, it is labeled with an element of Σ_ℓ or an element of Σ_r if it is a left child or a right child, respectively. In this way, the transducer knows whether a node is a left or a right child by examining the label. The root symbol of t is labeled with a symbol from Σ_ℓ. Furthermore, all internal nodes of t are labeled with labels in $\{0_\ell, 0_r, 1_\ell, 1_r\}$ and all leaf nodes are labeled with labels in $\{0'_\ell, 0'_r, 1'_\ell, 1'_r\}$. As explained above, we will use the sequence of #-symbols to make a sufficient number of copies of t.

The input DTD (d_{in}, s) is defined as follows:

- $d_{\text{in}}(s) = \# + 0_\ell + 1_\ell$;
- $d_{\text{in}}(\#) = \# + 0_\ell + 1_\ell$;
- for each $a \in \{0_\ell, 1_\ell, 0_r, 1_r\}$,
  $d_{\text{in}}(a) = (0_\ell + 1_\ell + 0'_\ell + 1'_\ell) + (0_\ell + 1_\ell + 0'_\ell + 1'_\ell)(0_r + 1_r + 0'_r + 1'_r)$; and,
- for each $a \in \{0'_\ell, 1'_\ell, 0'_r, 1'_r\}$, $d_{\text{in}}(a) = \varepsilon$.

Obviously, (d_{in}, s) can be expressed as a DTD(DFA). It can also be expressed

as a DTD(SL), as follows

$$d_{\text{in}}(a) = \Big(\big((\varphi[0_\ell^{=1}] \vee \varphi[1_\ell^{=1}] \vee \varphi[(0'_\ell)^{=1}] \vee \varphi[(1'_\ell)^{=1}])\big)$$
$$\oplus\big((\varphi[0_\ell^{=1}] \vee \varphi[1_\ell^{=1}] \vee \varphi[(0'_\ell)^{=1}] \vee \varphi[(1'_\ell)^{=1}])$$
$$\wedge (\varphi[0_r^{=1}] \vee \varphi[1_r^{=1}] \vee \varphi[(0'_r)^{=1}] \vee \varphi[(1'_r)^{=1}])\big)\Big)$$
$$\wedge\, s^{=0} \wedge \#^{=0}$$

for every $a \in \{0_\ell, 1_\ell, 0_r, 1_r\}$, where

- $\oplus$ denotes the "exclusive or";
- for every $i \in \{\ell, r\}$ and $x \in \{0_i, 1_i, 0'_i, 1'_i\}$, $\varphi[x^{=1}]$ denotes the conjunction

$$(x^{=1} \wedge \bigwedge_{y \in \{0_i, 1_i, 0'_i, 1'_i\} \setminus \{x\}} y^{=0}).$$

Notice that the size of the SL-formula expressing $d_{\text{in}}(a)$ is constant.

We construct a tree transducer $T = (Q_T, \Sigma_T, q^{\varepsilon}_{\text{copy}}, R_T)$. The alphabet of T is $\Sigma_T = \Sigma_\ell \cup \Sigma_r \cup \{s, \#, \text{error}, \text{ok}\}$. The state set Q_T includes the set $\{q^\ell, q^r \mid q \in Q_i, i \in \{1, \ldots, n\}\}$. Furthermore, the transducer will use $\lceil \log n \rceil$ special copying states q^j_{copy} to make at least n copies of the input tree. To define Q_T formally, we first introduce the notation $D(k)$, for $k = 0, \ldots, \lceil \log n \rceil$. Intuitively, $D(k)$ corresponds to the set of nodes of a complete binary tree of depth $k+1$. For example, $D(1) = \{\varepsilon, 0, 1\}$ and $D(2) = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$. The idea is that, if $i \in D(k) \setminus D(k-1)$, for $k > 0$, then i represents the binary encoding of a number in $\{0, \ldots, 2^k - 1\}$. Formally, if $k = 0$, then $D(k) = \{\varepsilon\}$; otherwise, $D(k) = D(k-1) \cup \bigcup_{j=0,1}\{ij \mid i \in D(k-1)\}$. The state set Q_T is then the union of the sets $Q^\ell = \{q^\ell \mid q \in Q_j, 1 \leq j \leq n\}$, $Q^r = \{q^r \mid q \in Q_j, 1 \leq j \leq n\}$, the set $\{q^j_{\text{copy}} \mid j \in D(\lceil \log n \rceil)\}$ and the set $\{\text{start}^\ell_j \mid n+1 \leq j \leq 2^{\lceil \log n \rceil}\}$. Note that the last set can be empty. It only contains dummy states translating any input to the empty string.

We next describe the action of the tree transducer T. Roughly, the operation of T on the input $s(\#(\#(\cdots\#(t))))$ can be divided in two parts: *(i)* copying the tree t a sufficient number of times while reading the #-symbols; and, *(ii)* simulating one of the TDBTAs on each copy of t. The tree transducer outputs the symbol "error" when one of the TDBTAs rejects t, or when the number of #-symbols in its input is not equal to $\lceil \log n \rceil$. Apart from copying the root symbol s to the output tree, T only writes the symbol "error" to the output. Hence, the output tree always has a root labeled s which has zero or more children labeled "error". The output DTD, which we define later, should then verify whether the root has at least one "error"-labeled child.

Formally, the transition rules in R_T are defined as follows:

- $(q^{\varepsilon}_{\text{copy}}, s) \rightarrow s(q^{0}_{\text{copy}} q^{1}_{\text{copy}})$. This rule puts s as the root symbol of the output tree.
- $(q^{i}_{\text{copy}}, \#) \rightarrow q^{i0}_{\text{copy}} q^{i1}_{\text{copy}}$ for $i \in D(\lceil \log n \rceil - 1) - \{\varepsilon\}$. These rules copy the tree t in the input at least n times, provided that there are enough #-symbols.
- $(q^{i}_{\text{copy}}, \#) \rightarrow \text{start}^{\ell}_{k}$, where $i \in D(\lceil \log n \rceil) - D(\lceil \log n \rceil - 1)$, and i is the binary representation of k. This rule starts the in-parallel simulation of the A_i's. For $i = n+1, \ldots, 2^{\lceil \log n \rceil}$, start^{ℓ}_{i} is just a dummy state transforming everything to the empty tree.
- $(q^{i}_{\text{copy}}, a) \rightarrow \text{error}$ for $a \in \Sigma$ and $i \in D(\lceil \log n \rceil)$. This rule makes sure that the output of T is accepted by the output tree automaton if there are not enough #-symbols in the input.
- $(\text{start}^{\ell}_{k}, \#) \rightarrow \text{error}$ for all $k = 1, \ldots, 2^{\lceil \log n \rceil}$. This rule makes sure that the output of T is accepted by the output tree automaton if there are too much #-symbols in the input.
- $(q^{\ell}, a_r) \rightarrow \varepsilon$ and $(q^{r}, a_{\ell}) \rightarrow \varepsilon$ for all $q \in Q_j$, $j = 1, \ldots, n$. This rule ensures that tree automata states intended for left (respectively right) children are not applied to right (respectively left) children.
- $(q^{\ell}, a_{\ell}) \rightarrow q^{\ell}_{1} q^{r}_{2}$ and $(q^{r}, a_r) \rightarrow q^{\ell}_{1} q^{r}_{2}$, for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = q_1 q_2$, and a is an internal symbol. This rule does the actual simulation of the tree automata A_i, $i = 1, \ldots, n$.
- $(q^{\ell}, a_{\ell}) \rightarrow q^{\ell}_{1}$ and $(q^{r}, a_r) \rightarrow q^{\ell}_{1}$, for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = q_1$ and a is an internal symbol. This rule does the actual simulation of the tree automata A_i, $i = 1, \ldots, n$.
- $(q^{\ell}, a_{\ell}) \rightarrow \varepsilon$ and $(q^{r}, a_r) \rightarrow \varepsilon$ for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = \varepsilon$ and a is a leaf symbol. This rule simulates accepting computations of the A_i's.
- $(q^{\ell}, a_{\ell}) \rightarrow \text{error}$ and $(q^{r}, a_r) \rightarrow \text{error}$ for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a)$ is undefined. This rule simulates rejecting computations of the A_i's.

It is straightforward to verify that, on input $s(\#(\#(\cdots\#(t))))$, T outputs the tree s if and only if there are $\lceil \log n \rceil$ #-symbols in the input and $t \in L(A_1) \cap \cdots \cap L(A_n)$.

Finally, $d_{\text{out}}(s) = \text{error error}^*$, which can easily be defined as a DTD(DFA) and as a DTD(SL).

It is easy to see that the reduction can be carried out in deterministic logarithmic space, that T has copying width 2, and that d_{in} and d_{out} do not depend on $A_1, \ldots, A_n$. □

4.2 Non-deleting: Fixed Input Schema

We turn to the typechecking problem in which we consider the input schema as fixed. We start by showing that typechecking is in PTIME in the case where we use DTDs with SL-expressions and the tree transducer is non-deleting (Theorem 13). To this end, we recall a lemma and introduce some necessary notions that are needed for the proof of Theorem 13.

For an SL-formula ϕ, we say that two strings w_1 and w_2 are *ϕ-equivalent* (denoted $w_1 \equiv_\phi w_2$) if $w_1 \models \phi$ if and only if $w_2 \models \phi$.

For $a \in \Sigma$ and $w \in \Sigma^*$, we denote by $\#_a(w)$ the number of a's occurring in w. We recall Lemma 17 from [23]:

Lemma 12 *Let ϕ be an SL-formula and let k be the largest integer occurring in ϕ. For every $w, w' \in \Sigma^*$, for every $a \in \Sigma$, if*

- *$\#_a(w') > k$ when $\#_a(w) > k$, and*
- *$\#_a(w') = \#_a(w)$, otherwise,*

then $w \equiv_\phi w'$.

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".

We are now ready to show the first PTIME result:

Theorem 13 *$TC^i[\mathcal{T}_{nd,uc}$, DTD(SL)$]$ is in* PTIME.

PROOF. Denote the tree transformation by $T = (Q_T, \Sigma, q_T^0, R_T)$ and the input and output DTDs by $(d_{\text{in}}, s_{\text{in}})$ and $(d_{\text{out}}, s_{\text{out}})$, respectively. As d_{in} is fixed, we can assume that d_{in} is reduced.

A PTIME algorithm is given that searches for a counter example. If no counter example can be found, then it follows that T typechecks with respect to d_{in}. The outline of the typechecking algorithm is as follows:

(1) Compute the set of "reachable pairs" (q, a) for which there exists a tree $t \in L(d_{\text{in}})$ and a node $u \in \text{Dom}(t)$ such that $\text{lab}^t(u) = a$ and T visits u in state q. That is, we compute all pairs (q, a) such that either
   - $q = q_T^0$ and $a = s_{\text{in}}$; or
   - (q', a') is a reachable pair, there is a q-labeled node in $\text{rhs}(q', a')$, and there exists a string $w_1 a w_2 \in d_{\text{in}}(a')$ for $w_1, w_2 \in \Sigma^*$.

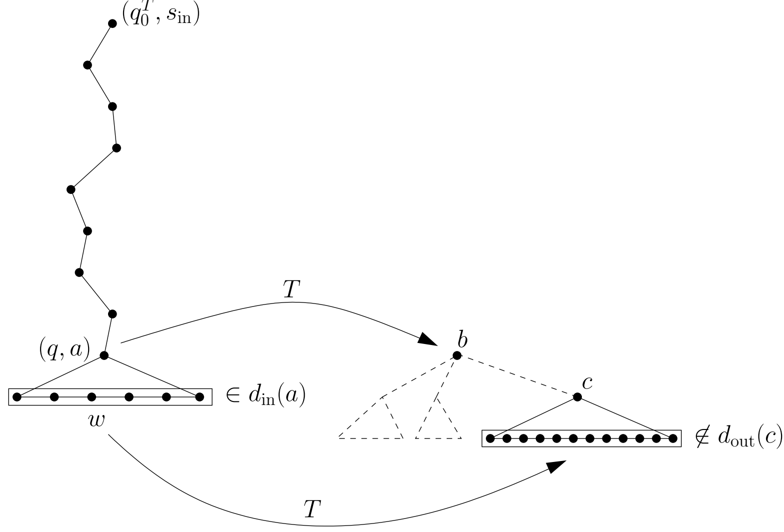


Fig. 7. Illustration of the typechecking algorithm in the proof of Theorem 13.

(2) For each such pair (q, a) test whether there exists a string $w \in d_{in}(a)$ such that $T^q(a(w))$ does not partly satisfy d_{out}. We call w a *counterexample*.

The algorithm is successful, if and only if there exists a counterexample.

We illustrate the general operation of the typechecking algorithm in Figure 7. In this figure, T visits the a-labeled node on the left in state q. Consequently, T outputs the hedge $\text{rhs}(q, a)$, which is illustraded by dotted lines on the right. The typechecking algorithm searches for a node u in $\text{rhs}(q, a)$ (which is labeled by c in the figure), such that the string of children of u is not in $L(d_{out}(c))$.

Notice that the typechecking algorithm does not assume that d_{out} is reduced (recall the definition of a reduced DTD from Section 3.2). We need to show that the algorithm is correct, that is, there exists a counterexample if and only if T does not typecheck with respect to d_{in} and d_{out}. Clearly, when the algorithm does not find a counterexample, T typechecks with respect to d_{in} and d_{out}. Conversely, suppose that the algorithm finds a pair (q, a) and a string w such that $T^q(a(w))$ does not partly satisfy d_{out}. So, since we assumed that d_{in} is reduced, there exists a tree $t \in L(d_{in})$ and a node $u \in \text{Dom}(t)$ such that $\text{lab}^t(u) = a$ and u is visited by T in state q. Also, there exists a node v in $T^q(a(w))$, such that the label of u is c and the string of children of u is not in $d_{out}(c)$. We argue that $T(t) \notin L(d_{out})$. There are two cases:

(i) if $L(d_{out})$ contains a tree with a c-labeled node, then $T(t) \notin d_{out}$ since $T^q(a(w))$ does not partly satisfy d_{out}; and
(ii) if $L(d_{out})$ does *not* contain a tree with a c-labeled node, then $T(t) \notin d_{out}$ since $T(t)$ contains a c-labeled node.

We proceed by showing that the algorithm can be carried out in polynomial

time. As the input schema is fixed, step (1) of the algorithm is in polynomial time. Indeed, we can compute the set of reachable pairs (q, a) in a top-down manner by a straightforward reachability algorithm.

To show that step (2) of the typechecking algorithm is in polynomial time, fix a tuple (q, a) that was a reachable pair in step (1) and a node u in $\text{rhs}(q, a)$ with label b. Let $z'_0 q_1 z'_1 \cdots q_n z'_n$ be the concatenation of u's children, where all $z'_0, \ldots, z'_n \in \mathcal{H}_\Sigma$ and $q_1, \ldots, q_n \in Q_T$. Let, for each i, $z_i = \text{top}(z'_i)$. We now search for a string $w \in \Sigma^*$ for which $w \models d_{\text{in}}(a)$, but for which $z_0\ \text{top}(T^{q_1}(w))\ z_1 \cdots \text{top}(T^{q_n}(w))\ z_n \not\models d_{\text{out}}(b)$. Recall from Section 3.3 that $\text{top}(T^q(w))$ is the homomorphic extension of $\text{top}(T^q(a))$ for $a \in \Sigma$, which is $\text{top}(\text{rhs}(q, a))$ in the case of non-deleting tree transducers.

Denote $d_{\text{in}}(a)$ by ϕ. Let $\{a_1, \ldots, a_s\}$ be the different symbols occurring in ϕ and let k be the largest integer occurring in ϕ. According to Lemma 12, every Σ-string is ϕ-equivalent to a string of the form $w = a_1^{m_1} \cdots a_s^{m_s}$ with $0 \leq m_i \leq k+1$ for each $i = 1, \ldots, s$. Note that there are $(k+1)^s$ such strings, which is a constant number, as it only depends on the input schema. For the following, the algorithm considers each such string w.

Fix such a string w such that $w \models \phi$. For each symbol c in $d_{\text{out}}(b)$, the number $\#_c(z_0 \text{top}(T^{q_1}(w)) z_1 \cdots \text{top}(T^{q_n}(w)) z_n)$ is equal to the linear sum

$$k_1^c \times \#_{a_1}(w) + \cdots + k_\ell^c \times \#_{a_\ell}(w) + k_{\ell+1}^c \times \#_{a_{\ell+1}}(w) + k_s^c \times \#_{a_s}(w) + k^c,$$

where $k^c = \#_c(z_0 \cdots z_n)$ and for each $i = 1, \ldots, s$, we have $k_i^c = \#_c(\text{top}(T^{q_1}(a_i)) \cdots \text{top}(T^{q_n}(a_i)))$. We now must test if there exists a string $w' \equiv_\phi w$ such that $z_0 \text{top}(T^{q_1}(w')) z_1 \cdots \text{top}(T^{q_n}(w')) z_n \not\models d_{\text{out}}(b)$. Let $a_1, \ldots, a_\ell$ be the symbols that occur at least $k+1$ times in w and $a_{\ell+1}, \ldots, a_s$ be the symbols that occur at most k times in w, respectively. Then, deciding whether w' exists is equivalent to finding an integer solution to the variables $x_{a_1}, \ldots, x_{a_s}$ for the boolean combination of linear (in)equalities $\Phi = \Phi_1 \wedge \neg\Phi_2$, where

- Φ_1 states that $w' \equiv_\phi w$, that is,

$$\Phi_1 = \bigwedge_{i=1}^{\ell} (x_{a_i} > k) \wedge \bigwedge_{j=\ell+1}^{s} \big(x_{a_j} = \#_{a_j}(w)\big);$$

  and
- Φ_2 states that $z_0 q_1(w') z_1 \cdots q_n(w') z_n \models d_{\text{out}}(b)$, that is, Φ_2 is defined by replacing every occurrence of $c^{=i}$ or $c^{\geq i}$ in $d_{\text{out}}(b)$ by the equation

$$\sum_{j=1}^{s} (k_j^c \times x_{a_j}) + k^c = i$$

or by

$$\sum_{j=1}^{s}(k_j^c \times x_{a_j}) + k^c \geq i,$$

respectively.

In the above (in)equalities, x_{a_i}, $1 \leq i \leq s$, represents the number of occurrences of a_i in w'.

Finding a solution for Φ now consists of finding integer values for $x_{a_1}, \ldots, x_{a_s}$ so that Φ evaluates to `true`. Corollary 28 in the Appendix shows that we can decide in PTIME whether such a solution for Φ exists. □

Theorem 14 *$TC^i[\mathcal{T}_{nd,bc}$, DTD(DFA)] is* NLOGSPACE*-complete.*

PROOF. In Theorem 19(2), we prove that the problem is NLOGSPACE-hard, even if both the input and output schemas are fixed. Hence, it remains to show that the problem is in NLOGSPACE.

Let us denote the tree transformation by $T = (Q_T, \Sigma, q_T^0, R_T)$ and the input and output DTDs by (d_{in}, r) and d_{out}, respectively. We can assume that d_{in} is reduced. [4]

The first part of the algorithm is similar to the one in Theorem 13. The typechecking algorithm can be summarized as follows:

(1) Guess a sequence of pairs $(q_0, a_0), (q_1, a_1), \ldots, (q_n, a_n)$ in $Q_T \times \Sigma$, such that
 - $(q_0, a_0) = (q_T^0, r)$; and
 - for every pair (q_i, a_i), q_{i+1} occurs in $\text{rhs}(q_i, a_i)$ and a_{i+1} occurs in some string in $L(d_{\text{in}}(a_i))$.

 We only need to remember (q_n, a_n) as a result of this step.

(2) Test whether there exists a string $w \in d_{\text{in}}(a_n)$ such that $T^q(a_n(w))$ does not partly satisfy d_{out}.

The algorithm is successful if and only if w exists and, hence, the problem does not typecheck.

The first step is a straightforward reachability algorithm, which is in NLOGSPACE. It remains to show that the second step is in NLOGSPACE.

Let (q, a) be the pair (q_n, a_n) computed in step one. We guess a node in $\text{rhs}(q_n, a_n)$, say that it is labeled with $b \in \Sigma$. Let $d_{\text{out}}(b) = (Q_{\text{out}}, \Sigma, \delta_{\text{out}},$

[4] Reducing d_{in} would be PTIME-complete otherwise, see Corollary 30 in the Appendix.

$\{p_I\}, F)$ be a DFA and let k be the copying bound of T. Let $z'_0 q_1 z'_1 \cdots q_\ell z'_\ell$ be the concatenation of u's children, where $\ell \leq k$. Let, for each i, $z_i = \text{top}(z'_i)$.

So we want to check whether there exists a string w such that $z_0 \text{top}(T^{q_1}(w)) z_1 \cdots \text{top}(T^{q_\ell}(w)) z_\ell$ is not accepted by $d_{\text{out}}(b)$. We guess w one symbol at a time and simulate in parallel ℓ copies of $d_{\text{out}}(b)$ and one copy of $d_{\text{in}}(a)$.

By $\hat{\delta}$ we denote the canonical extension of δ to strings in Σ^*. We start by guessing states $p_1, \ldots, p_\ell$ of $d_{\text{out}}(b)$, where $p_1 = \hat{\delta}_{\text{out}}(p_I, z_0)$, and keep a copy of these on tape, to which we refer as $p'_1, \ldots, p'_\ell$. Next, we keep on guessing symbols c of w, whereafter we replace each p_i by $\hat{\delta}_{\text{out}}(p_i, \text{top}(T^{q_i}(c)))$. The input automaton obviously starts in its initial state and is simulated in the straightforward way.

The machine non-deterministically stops guessing, and checks whether, for each $i = 1, \ldots, \ell - 1$, $\hat{\delta}_{\text{out}}(p_i, z_i) = p'_{i+1}$ and $\hat{\delta}_{\text{out}}(p_\ell, z_\ell) \in F$. For the input automaton, it simply checks whether the current state is the final state. If the latter tests are positive, then the algorithm accepts, otherwise, it rejects.

We only keep $2\ell + 1$ states on tape, which is a constant number, so the algorithm runs in NLOGSPACE. □

Theorem 15 *(1) $TC^i[\mathcal{T}_{nd,uc}$, DTD(DFA)] is* PSPACE*-complete; and (2) $TC^i[\mathcal{T}_{nd,bc}$, DTD(NFA)] is* PSPACE*-complete.*

PROOF. In [23], it was shown that both problems are in PSPACE. We proceed by showing that they are also PSPACE-hard.

(1) We reduce the intersection emptiness problem of an arbitrary number of deterministic finite automata with alphabet $\{0, 1\}$ to the typechecking problem. This problem is known to be PSPACE-hard, as shown in Corollary 23(1) in the Appendix. Assume given the DFAs $M_1, \ldots, M_n$. Our reduction only requires logarithmic space. We define a transducer $T = (Q_T, \{s, 0, 1, \#_0, \ldots, \#_n\}, q_T^0, R_T)$ and two DTDs d_{in} and d_{out} such that T typechecks with respect to d_{in} and d_{out} if and only if $\bigcap_{i=1}^n L(M_i) = \emptyset$.

The DTD (d_{in}, s) defines trees of depth two, where the string formed by the children of the root labeled s is an arbitrary string in $\{0, 1\}^*$, so $d_{\text{in}}(s) = (0 + 1)^*$. 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 $(q_T^0, s) \to s(\#_0 q \#_1 q \ldots \#_{n-1} q \#_n)$ and $(q, a) \to a$, for every $a \in \{0, 1\}$. Finally, (d_{out}, s) defines a tree of depth two as follows:

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

Clearly, $d_{\text{out}}(s)$ 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 $\#_{i-1}$, until it encounters $\#_i$. It then verifies that at least one M_i rejects.

It is easy to see that this reduction can be carried out by a deterministic logspace algorithm.

(2) This is an easy reduction from the universality problem of an NFA N with alphabet $\{0, 1\}$. The latter problem is PSPACE-hard, as shown in Corollary 23(2) in the Appendix. Again, the input DTD (d_{in}, s) defines a tree of depth two where $d_{\text{in}}(s) = (0 + 1)^*$. The tree transducer is the identity transformation. The output DTD d_{out} has as start symbol s and $d_{\text{out}}(s) = L(N)$. Hence, this instance typechecks if and only if $\{0, 1\}^* \subseteq L(N)$.

This reduction can be carried out by a deterministic logspace algorithm. □

4.3 Non-deleting: Fixed Output Schema

Again, upper bounds carry over from [23]. Also, when the output DTD is a DTD(NFA), we can convert it into an equivalent DTD(DFA) in constant time. As the PTIME typechecking algorithm for TC[$\mathcal{T}_{nd,bc}$,DTD(DFA)] in [23] also works when the input DTD is a DTD(NFA), we have that the problem TC^o[$\mathcal{T}_{nd,bc}$,DTD(NFA)] is in PTIME. As the PTIME-hardness proof for TC[$\mathcal{T}_{nd,bc}$,DTD(DFA)] in [23] uses a fixed output schema, we immediately obtain the following.

Theorem 16 *$TC^o[\mathcal{T}_{nd,bc}, DTD(NFA)]$ is* PTIME*-complete.*

The lower bound in the presence of tree automata will be discussed in Section 4.4. The case requiring some real work is TC^o[$\mathcal{T}_{nd,uc}$, DTD(DFA)].

Theorem 17 *$TC^o[\mathcal{T}_{nd,uc}$, DTD(DFA)] is* PSPACE*-complete.*

PROOF. In [23], it was shown that the problem is in PSPACE. We proceed by showing PSPACE-hardness.

We use a LOGSPACE reduction from the corridor tiling problem [9]. Let $(T, V, H, \bar{\vartheta}, \bar{\beta})$ be a tiling system, where $T = \{\vartheta_1, \ldots, \vartheta_k\}$ is the set of tiles, $V \subseteq T \times T$ and $H \subseteq T \times T$ are the sets of vertical and horizontal constraints respectively, and $\bar{\vartheta}$ and $\bar{\beta}$ are the top and bottom row, respectively. Let n be the width of $\bar{\vartheta}$ and $\bar{\beta}$. The tiling system has a solution if there is an $m \in \mathbb{N}$ such that the space $m \times n$ (m rows and n columns) can be correctly tiled

with the additional requirement that the bottom and top row are $\bar{\beta}$ and $\bar{\vartheta}$, respectively. A tiling can then be seen as a mapping $\lambda : m \times n \to T$. A tiling λ is correct when all horizontal and vertical constraints are respected. That is, $(\lambda(k,j), \lambda(k,j+1)) \in H$ and $(\lambda(k,j), \lambda(k+1,j)) \in V$.

We define the input DTD d_{in} over the alphabet $\Sigma := \{(i, \vartheta_j) \mid j \in \{1,\ldots,k\}, i \in \{1,\ldots,n\}\} \cup \{r\}$; r is the start symbol. Define

$$d_{\text{in}}(r) = \#\bar{\beta}\#\Big(\Sigma_1 \cdot \Sigma_2 \cdots \Sigma_n \#\Big)^* \bar{\vartheta}\#,$$

where we denote by Σ_i the set $\{(i, \vartheta_j) \mid j \in \{1,\ldots,k\}\}$. Here, # functions as a row separator. For all other alphabet symbols $a \in \Sigma$, $d_{\text{in}}(a) = \varepsilon$. So, d_{in} encodes all possible tilings that start and end with the bottom row $\bar{\beta}$ and the top row $\bar{\vartheta}$, respectively.

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

The state set Q_B is partitioned into two sets, Q_{hor} and Q_{ver}:

- Q_{hor} is for the horizontal constraints: for every $i \in \{1,\ldots,n-1\}$ and $\vartheta \in T$, $q_{i,\vartheta} \in Q_{\text{hor}}$ transforms the rows in the tiling such that it is possible to check that when position i carries a ϑ, position $i+1$ carries a ϑ' such that $(\vartheta, \vartheta') \in H$; and,
- Q_{ver} is for the vertical constraints: for every $i \in \{1,\ldots,n\}$ and $\vartheta \in T$, $p_{i,\vartheta} \in Q_{\text{ver}}$ transforms the rows in the tiling such that it is possible to check that when position i carries a ϑ, the next row carries a ϑ' on position i such that $(\vartheta, \vartheta') \in V$.

The tree transducer B always starts its transformation with the rule

$$(q_B^0, r) \to r(w),$$

where w is the concatenation of all of the above states, separated by the delimiter \$. The other rules are of the following form:

- Horizontal constraints: for all $(j, \vartheta) \in \Sigma$ add the rule $(q_{i,\vartheta}, (j, \vartheta')) \to \alpha$

where $q_{i,\vartheta} \in Q_{\text{hor}}$ and

$$\alpha = \begin{cases} \texttt{trigger} & \text{if } j = i \text{ and } \vartheta = \vartheta' \\ \texttt{other} & \text{if } j = i \text{ and } \vartheta \neq \vartheta' \\ \texttt{ok} & \text{if } j = i+1 \text{ and } (\vartheta, \vartheta') \in H \\ \texttt{error} & \text{if } j = i+1 \text{ and } (\vartheta, \vartheta') \notin H \\ \texttt{other} & \text{if } j \neq i \text{ and } j \neq i+1 \end{cases}$$

Finally, $(q_{i,\vartheta}, \#) \to \texttt{hor}$.

The intuition is as follows: if the i-th position in a row is labeled with ϑ, then this position is transformed into `trigger`. Position $i+1$ is transformed to `ok` when it carries a tile that matches ϑ horizontally. Otherwise, it is transformed to `error`. All other symbols are transformed into an `other`.

On a row, delimited by two `hor`-symbols, the output DFA rejects if and only if there is a `trigger` immediately followed by an `error`. When there is no `trigger`, then position i was not labeled with ϑ. So, the label `trigger` acts as a trigger for the output automaton.

- Vertical constraints: for all $(j, \vartheta) \in \Sigma$, add the rule $(p_{i,\vartheta}, (j, \vartheta')) \to \alpha$ where $p_{i,\vartheta} \in Q_{\text{ver}}$ and

$$\alpha = \begin{cases} \texttt{trigger1} & \text{if } (j, \vartheta') = (i, \vartheta) \text{ and } (\vartheta, \vartheta) \in V \\ \texttt{trigger2} & \text{if } (j, \vartheta') = (i, \vartheta) \text{ and } (\vartheta, \vartheta) \notin V \\ \texttt{ok} & \text{if } j = i,\ \vartheta \neq \vartheta', \text{ and } (\vartheta, \vartheta') \in V \\ \texttt{error} & \text{if } j = i,\ \vartheta \neq \vartheta', \text{ and } (\vartheta, \vartheta') \notin V \\ \texttt{other} & \text{if } j \neq i \end{cases}$$

Finally, $(p_{i,\vartheta}, \#) \to \texttt{ver}$.

The intuition is as follows: if the i-th position in a row is labeled with ϑ, then this position is transformed into `trigger1` when $(\vartheta, \vartheta) \in V$ and to `trigger2` when $(\vartheta, \vartheta) \notin V$. Here, both `trigger1` and `trigger2` act as a trigger for the output automaton: they mean that position i was labeled with ϑ. But no `trigger1` and `trigger2` can occur in the same transformed row as either $(\vartheta, \vartheta) \in V$ or $(\vartheta, \vartheta) \notin V$. When position i is labeled with $\vartheta' \neq \vartheta$, then we transform this position into `ok` when $(\vartheta, \vartheta') \in V$, and in `error` when $(\vartheta, \vartheta') \notin V$. All other positions are transformed into `other`.

The output DFA then works as follows. If a position is labeled `trigger1` then it rejects if there is an `error` occurring after the next `ver`. If a position is labeled `trigger2`, then it rejects if there is a `trigger2` or an `error` occurring after the next `ver`. Otherwise, it accepts that row.

By making use of the delimiters ver and hor, both above described automata can be combined into one automaton, taking care of the vertical and the horizontal constraints. This automaton resets to its initial state whenever it reads the delimiter symbol \$. Note that the output automaton is defined over the fixed alphabet $\{\mathtt{trigger}, \mathtt{trigger1}, \mathtt{trigger2}, \mathtt{error}, \mathtt{ok}, \mathtt{other}, \mathtt{hor}, \mathtt{ver}, \$\}$. □

Although the results in [23] were formulated in the context of variable schemas, the proofs for bounded copying, non-deleting tree transducers with DTD(SL) and with DTD(DFA) schemas actually used a fixed output schema. We can therefore sharpen these results as follows.

Theorem 18 *(1) $TC^o[\mathcal{T}_{nd,bc}$, DTD(SL)] is* coNP-*complete;*
(2) $TC^o[\mathcal{T}_{nd,bc}$, DTD(DFA)] is PTIME-*complete.*

4.4 Non-deleting: Fixed Input and Output Schema

We turn to the case where both input and output schemas are fixed. The following two theorems give us several new tractable cases.

Theorem 19 *(1) $TC^{io}[\mathcal{T}_{nd,bc}$, DTD(SL)] is* NLOGSPACE-*complete.*
(2) $TC^{io}[\mathcal{T}_{nd,bc}$, DTD(DFA)] is NLOGSPACE-*complete.*

PROOF. For both problems, membership in NLOGSPACE follows from Theorem 20. Indeed, every DTD(SL) can be rewritten into an equivalent DTD(NFA) in constant time as the input and output schemas are fixed.

We proceed by showing NLOGSPACE-hardness. We say that an NFA $N = (Q_N, \Sigma, \delta_N, I_N, F_N)$ has *degree of nondeterminism 2* if *(i)* I_N has at most two elements and *(ii)* for every $q \in Q_N$ and $a \in \Sigma$, the set $\delta_N(q, a)$ has at most two elements. We give a LOGSPACE reduction from the emptiness problem of an NFA with alphabet $\{0, 1\}$ and a degree of nondeterminism 2 to the typechecking problem. According to Lemma 24 in the Appendix, this problem is NLOGSPACE-hard. Intuitively, the input DTD will define all possible strings over alphabet $\{0, 1\}$. The tree transducer simulates the NFA and outputs "accept" if a computation branch accepts, and "error" if a computation branch rejects. The output DTD defines trees where all leaves are labeled with "error".

More concretely, let $N = (Q_N, \{0, 1\}, \delta_N, \{q_N^0\}, F_N)$ be an NFA with degree of nondeterminism 2. The input DTD (d_{in}, r) defines all unary trees, where the unique leaf is labeled with a special marker #. That is, $d_{\text{in}}(r) = d_{\text{in}}(0) =$

$d_{\text{in}}(1) = (0+1+\#)$ and $d_{\text{in}}(\#) = \varepsilon$. Note that these languages can be defined by SL-formulas or DFAs which are sufficiently small for our purpose.

Given a tree $t = r(a_1(\cdots(a_n(\#))\cdots))$, the tree transducer will simulate every computation of N on the string $a_1 \cdots a_n$. The tree transducer $T = (Q_T, \{r, \#, 0, 1, \text{error}, \text{accept}\}, q_T^0, R_T)$ simulates N's nondeterminism by copying the remainder of the input twice in every step. Formally, Q_T is the union of $\{q_T^0\}$ and Q_N, and R_T contains the following rules:

- $(q_T^0, r) \to r(q_N^0)$. This rule puts r as the root symbol of the output tree and starts the simulation of N.
- $(q_N, a) \to a(q_N^1, q_N^2)$, where $q_N \in Q_N$, $a \in \{0,1\}$ and $\delta_N(q_N, a) = \{q_N^1, q_N^2\}$. This rule does the actual simulation of N. By continuing in both states q_N^1 and q_N^2, we simulate *all* possible computations of N.
- $(q_N, a) \to \text{error}$ if $\delta_N(q_N, a) = \emptyset$. If N rejects, we output the symbol "error".
- $(q_N, \#) \to \text{error}$ for $q_N \notin F_N$; and
- $(q_N, \#) \to \text{accept}$ for $q_N \in F_N$. These last two rules verify whether N is in an accepting state after reading the entire input string.

Notice that T outputs the symbol "error" (respectively "accept") if and only if a computation branch of N rejects (respectively accepts).

The output of T is always a tree in which only the symbols "error" and "accept" occur at the leaves. The output DTD then needs to verify that only the symbol "error" occurs at the leaves. Formally, $d_{\text{out}}(r) = d_{\text{out}}(0) = d_{\text{out}}(1) = \{0, 1, \text{error}\}^+$ and $d_{\text{out}}(\text{error}) = \varepsilon$. Again, these languages can be defined by sufficiently small SL-formulas or DFAs.

It is easy to see that the reduction only requires logarithmic space. □

Theorem 20 *$TC^{io}[\mathcal{T}_{nd,uc}$, DTD(NFA)] is* NLOGSPACE*-complete.*

PROOF. The NLOGSPACE-hardness of the problem follows from Theorem 19(b), where it is shown that the problem is already NLOGSPACE-hard when DTD(DFA)s are used as input and output schema.

We show that the problem is also in NLOGSPACE. Thereto, let $T = (Q_T, \Sigma, q_T^0, R_T)$ be the tree transducer, and let (d_{in}, r) and d_{out} be the input and output DTDs, respectively. As both d_{in} and d_{out} are fixed, we can assume without loss of generality that they are reduced.[5] For the same reason, we can also assume that the NFAs in d_{in} and d_{out} are determinized.

[5] In general, reducing a DTD(NFA) is PTIME-complete (Section 3.2).

The first half of the algorithm is similar to the one in Theorem 13. We guess a sequence of reachable state-label pairs $(p_0, a_0), (p_1, a_1), \ldots, (p_n, a_n)$ where $n < |Q_T||\Sigma|$ such that

- $(p_0, a_0) = (q_T^0, r)$; and
- for every pair (p_i, a_i), p_{i+1} occurs in $\text{rhs}(p_i, a_i)$ and a_{i+1} occurs in some string in $L(d_{\text{in}}(a_i))$.

Each time we guess a new pair in this sequence, we forget the previous one, so that we only keep a state, an alphabet symbol, a counter, and the binary representation of $|Q_T||\Sigma|$ on tape.

For simplicity, we write (p_n, a_n) as (p, a) in the remainder of the proof. We guess a node $u \in \text{Dom}(\text{rhs}(p, a))$. Let $b = \text{lab}^{\text{rhs}(p,a)}(u)$ and let $z_0 q_1 z_1 \cdots q_k z_k$ be the concatenation of u's children, where every $z_0, \ldots, z_k \in \Sigma^*$ and every $q_1, \ldots, q_k \in Q_T$, then we want to check whether there exists a string $w \in d_{\text{in}}(a)$ such that $z_0 \text{top}(T^{q_1}(w)) z_1 \cdots \text{top}(T^{q_k}(w)) z_k$ is not accepted by $d_{\text{out}}(b)$. Recall from Section 3.3 that, for a state $q \in Q_T$, we denote by $\text{top}(T^q(w))$ the homomorphic extension of $\text{top}(T^q(c))$ for $c \in \Sigma$, which is $\text{top}(\text{rhs}(q, c)))$ in the case of non-deleting tree transducers. We could do this by guessing w one symbol at a time and simulating k copies of $d_{\text{out}}(b)$ and one copy of $d_{\text{in}}(a)$ in parallel, like in the proof of Theorem 14. However, as k is not fixed, the algorithm would use superlogarithmic space.

So, we need a different approach. To this end, let $A = (Q_{\text{in}}, \Sigma, \delta_{\text{in}}, q_{\text{in}}^0, F_{\text{in}})$ and $B = (Q_{\text{out}}, \Sigma, \delta_{\text{out}}, q_{\text{out}}^0, F_{\text{out}})$ be the DFAs accepting $d_{\text{in}}(a)$ and $d_{\text{out}}(b)$, respectively. To every $q \in Q_T$, we associate a function

$$f_q : Q_{\text{out}} \times \Sigma \to Q_{\text{out}} : (p', c) \mapsto \hat{\delta}_{\text{out}}(p', \text{top}(T^q(c))),$$

where $\hat{\delta}_{\text{out}}$ denotes the canonical extension of δ_{out} to strings in Σ^*. Note that there are maximally $|Q_{\text{out}}|^{|Q_{\text{out}}||\Sigma|}$ such functions. Let K be the cardinality of the set $\{f_q \mid q \in Q_T\}$. Hence, K is bounded from above by $|Q_{\text{out}}|^{|Q_{\text{out}}||\Sigma|}$, which is a constant (with respect to the input). Let $f_1, \ldots, f_K$ an arbitrary enumeration of $\{f_q \mid q \in Q_T\}$.

The typechecking algorithm continues as follows. We start by writing the $(1+K\cdot|Q_{\text{out}}|)$-tuple $(q_{\text{in}}^0, q'_1, \ldots, q'_{|Q_{\text{out}}|}, \ldots, q'_1, \ldots, q'_{|Q_{\text{out}}|})$ on tape, where $Q_{\text{out}} = \{q'_1, \ldots, q'_{|Q_{\text{out}}|}\}$. We will refer to this tuple as the tuple $\bar{p} := (p'_0, \ldots, p'_{K\cdot|Q_{\text{out}}|})$. We explain how we update $\bar{p}$ when guessing w symbol by symbol. Every time when we guess the next symbol c of w, we overwrite the tuple $\bar{p}$ by

$$\big(\delta_{\text{in}}(p'_0, c), f_1(p'_1, c), \ldots, f_1(p'_{|Q_{\text{out}}|}, c), \ldots \\ \ldots, f_K(p'_{(K-1)\cdot|Q_{\text{out}}|+1}, c), \ldots, f_K(p'_{K\cdot|Q_{\text{out}}|}, c)\big).$$

Notice that there are at most $|Q_{\text{in}}| \cdot K \cdot |Q_{\text{out}}|^2$ different $(K \cdot |Q_{\text{out}}|+1)$-tuples of this form. We nondeterministically determine when we stop guessing symbols of w.

It now remains to verify whether w was indeed a string such that $w \in d_{\text{in}}(a)$ and $z_0 \text{top}(T^{q_1}(w))z_1 \cdots \text{top}(T^{q_k}(w))z_k \notin d_{\text{out}}(b)$. The former condition is easy to test: we simply have to test whether $p'_0 \in F_{\text{in}}$. To test the latter condition, we read the string $z_0 q_1 z_1 \cdots q_k z_k$ from left to right while performing the following tests. We keep a state of $d_{\text{out}}(b)$ in memory and refer to it as the "current state".

(1) The initial current state is q^0_{out}.
(2) If the current state is p' and we read z_j, then we change the current state to $\hat{\delta}_{\text{out}}(p', z_j)$.
(3) If the current state is p' and we read q_j, then we change the current state to p'_i in $\bar{p}$, where for i, the following condition holds. Let $\ell, m = 1, \ldots, K \cdot |Q_{\text{out}}|$ be the smallest integers such that
   - $p' = q'_\ell$ in Q_{out}, and
   - $f_{q_j} = f_m$.

   Then $i = (m-1)K + \ell$.

   Note that deciding whether $p' = q'_\ell$ and $f_{q_j} = f_m$ can be done deterministically in logarithmic space, as the output schema is fixed. Consequently, i can also be computed in constant time and space.
(4) We stop and accept if the current state is a non-accepting state after reading z_k.

□

Theorem 21 *$TC^{io}[\mathcal{T}_{nd,bc}$, DTA(DFA)$]$ is* EXPTIME*-complete.*

PROOF. The proof is quite analogous to the proof of Theorem 11. As deletion is now disallowed, whereas it was allowed in Theorem 11, we need to define the rules of the transducer $T = (Q_T, \Sigma_T, q^\varepsilon_{\text{copy}}, R_T)$ differently.

The language defined by the input schema is exaclty the same as in Theorem 11. The transition rules in R_T are defined as follows:

- $(q^\varepsilon_{\text{copy}}, s) \to s(q^0_{\text{copy}} q^1_{\text{copy}})$;
- $(q^i_{\text{copy}}, \#) \to \#(q^{i0}_{\text{copy}} q^{i1}_{\text{copy}})$ for $i \in D(\lceil \log n \rceil - 1) - \{\varepsilon\}$;
- $(q^i_{\text{copy}}, \#) \to \#(\text{start}^\ell_k)$, where $i \in D(\lceil \log n \rceil) - D(\lceil \log n \rceil - 1)$, and i is the binary representation of k;
- $(q^i_{\text{copy}}, a) \to \text{error}$ for $a \in \Sigma$ and $i \in D(\lceil \log n \rceil)$;
- $(\text{start}^\ell_k, \#) \to \text{error}$ for all $k = 1, \ldots, 2^{\lceil \log n \rceil}$;
- $(q^\ell, a_r) \to \varepsilon$ and $(q^r, a_\ell) \to \varepsilon$ for all $q \in Q_j$, $j = 1, \ldots, n$;

- $(q^\ell, a_\ell) \to a_\ell(q_1^\ell q_2^r)$ and $(q^r, a_r) \to a_r(q_1^\ell q_2^r)$, for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = q_1 q_2$, and a is an internal symbol;
- $(q^\ell, a_\ell) \to a_\ell(q_1^\ell)$ and $(q^r, a_r) \to a_r(q_1^\ell)$, for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = q_1$ and a is an internal symbol;
- $(q^\ell, a_\ell) \to \varepsilon$ and $(q^r, a_r) \to \varepsilon$ for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a) = \varepsilon$ and a is a leaf symbol; and
- $(q^\ell, a_\ell) \to$ error and $(q^r, a_r) \to$ error for every $q \in Q_i$, $i = 1, \ldots, n$, such that $\delta_i(q, a)$ is undefined.

It is straightforward to verify that, on input $s(\#(\#(\cdots\#(t))))$, T performs the identity transformation if and only if there are $\lceil \log n \rceil$ #-symbols in the input and $t \in L(A_1) \cap \cdots \cap L(A_n)$. All other outputs contain at least one leaf labeled "error".

Finally, the output tree automaton accepts all trees with at least one leaf that is labeled "error". So the only counterexamples for typechecking are those trees that are accepted by all automata $A_1, \ldots, A_n$.

It is easy to see that the reduction can be carried out in deterministic logarithmic space, that T has copying width 2, and that the input and output schemas do not depend on $A_1, \ldots, A_n$. $\square$

5 Conclusion

We considered the complexity of typechecking in the presence of fixed input and/or output schemas. We have settled an open question in [23], namely that TC$[\mathcal{T}_{nd,bc}, DTA]$ is EXPTIME-complete.

In comparison with the results in [23], fixing input and/or output schemas only lowers the complexity in the presence of DTDs and when deletion is disallowed. Here, we see that the complexity is lowered when

(1) the input schema is fixed, in the case of DTD(SL)s;
(2) the input schema is fixed, in the case of DTD(DFA)s;
(3) the output schema is fixed, in the case of DTD(NFA)s; and
(4) both input and output schema are fixed, in all cases.

In all of these cases, the complexity of the typechecking problem is in polynomial time.

It is striking, however, that in many cases, the complexity of typechecking does not decrease significantly by fixing the input and/or output schema, and most cases remain intractable. We have to leave the precise complexity (that is, the PTIME-hardness) of $TC^i[\mathcal{T}_{nd,uc}, \text{DTD(SL)}]$ as an open problem.

Acknowledgments

We thank Giorgio Ghelli for raising the question about the complexity of type-checking in the setting of a fixed output schema. We also thank Joos Heintz for providing us with a useful reference to facilitate the proof of Proposition 27.

Appendix: Definitions and Basic Results

The purpose of this Appendix is to prove some lemmas that we use in the body of the paper. We first introduce some notations and definitions needed for the propositions and proofs further on in this Appendix. We also survey some complexity bounds on decisions problems concerning automata that are used throughout the paper.

We show that the complexities of the classical decision problems of string and tree automata are preserved when the automata operate over fixed alphabets. We will consider the following decision problems for string automata:

Emptiness: Given an automaton A, is $L(A) = \emptyset$?
Universality: Given an automaton A, is $L(A) = \Sigma^*$?
Intersection emptiness: Given the automata $A_1, \ldots, A_n$, is $L(A_1) \cap \cdots \cap L(A_n) = \emptyset$?

The corresponding decision problems for tree automata are defined analogously.

We associate to each label $a \in \Sigma$ a unique binary string $\text{enc}(a) \in \{0,1\}^*$ of length $\lceil \log |\Sigma| \rceil$. For a string $s = a_1 \cdots a_n$, $\text{enc}(s) = \text{enc}(a_1) \cdots \text{enc}(a_n)$. This encoding can be extended to string languages in the obvious way.

We show how to extend the encoding "enc" to trees over alphabet $\{0, 1, 0', 1'\}$. Here, 0 and 1 are internal labels, while $0'$ and $1'$ are leaf labels. Let $\text{enc}(a) = b_1 \cdots b_k$ for $a \in \Sigma$. Then we denote by tree-enc(a) the unary tree $b_1(b_2(\cdots(b_k)))$, if a is an internal label, and the unary tree $b_1(b_2(\cdots(b'_k)))$, otherwise. Then, the enc-fuction can be extended to trees as follows: for $t = a(t_1 \cdots t_n)$,

$$\text{enc}(t) = \text{tree-enc}(a)(\text{enc}(t_1) \cdots \text{enc}(t_n)).$$

Note that we abuse notation here. The hedge $\text{enc}(t_1) \cdots \text{enc}(t_n)$ is intended to be the child of the leaf in tree-enc(a). The encoding can be extended to tree languages in the obvious way.

Proposition 22 *Let B be a TDBTA. Then there is a TDBTA B' over the*

alphabet $\{0,1,0',1'\}$ *such that* $L(B') = enc(L(B))$. *Moreover,* B' *can be constructed from* B *in* LOGSPACE.

PROOF. Let $B = (Q_B, \Sigma_B, \delta_B, F_B)$ be a TDBTA. Let $k := \lceil \log |\Sigma_B| \rceil$. We define $B' = (Q_{B'}, \{0,1,0',1'\}, \delta_{B'}, F_{B'})$. Set $Q_{B'} = \{q_x \mid q \in Q_B$ and x is a prefix of $\text{enc}(a)$, where $a \in \Sigma_B\}$ and $F_{B'} = \{q_\varepsilon \mid q \in F_B\}$. To define the transition function, we introduce some notation. For each $a \in \Sigma$ and $i,j = 1,\ldots,\lceil \log |\Sigma_B| \rceil$, denote by $a[i:j]$ the substring of $\text{enc}(a)$ from position i to position j (we abbreviate $a[i:i]$ by $a[i]$). For each transition $\delta_B(q,a) = q^1 q^2$, add the transitions $\delta_{B'}(q_\varepsilon, a[1]) = q_{a[1]}$, $\delta_{B'}(q_{a[1]}, a[2]) = q_{a[1:2]}, \ldots, \delta_{B'}(q_{a[1:k-1]}, a[k]) = q^1_\varepsilon q^2_\varepsilon$. Other transitions are defined analogously. Clearly, B' is a TDBTA, $L(B') = \text{enc}(L(B))$, and B' can be constructed from B in LOGSPACE. □

It is straightforward to show that Proposition 22 also holds for NFAs and DFAs (the proofs are analogous). It is immediate from Proposition 22, that lower bounds of decision problems for automata over arbitrary alphabets [16,30,32] carry over to automata working over fixed alphabets. Hence, we obtain the following corollary to Proposition 22:

Corollary 23 *Over the alphabet* $\{0,1\}$*, the following statements hold:*

(1) Intersection emptiness of an arbitrary number of DFAs is PSPACE*-hard [16].*
(2) Universality of NFAs is PSPACE*-hard [32].*

Over the alphabet $\{0,1,0',1'\}$*, the following statement holds:*

(3) Intersection emptiness of an arbitrary number of TDBTAs is EXPTIME*-hard [30].*

Lemma 24 now immediately follows from NLOGSPACE-hardness of the reachability problem on graphs with out-degree 2 [15].

Lemma 24 *The emptiness problem for an NFA with alphabet* $\{0,1\}$ *degree of nondeterminism 2 is* NLOGSPACE*-hard.*

We now aim at proving Proposition 27, which states that we can find integer solutions to arbitrary Boolean combinations of linear (in)equalities in polynomial time, when the number of variables is fixed. To this end, we use a well-known result by Ferrante and Rackoff.

First, we need some definitions. We define logical formulas with variables $x_1, x_2, \ldots$ and linear equations with factors in $\mathbb{Q}$. A *term* is an expression of the form a_1/b_1, $a_1/b_1 x_1 + \cdots + a_n/b_n x_n$, or $a_1/b_1 x_1 + \cdots + a_{n-1}/b_{n-1} x_{n-1} + a_n/b_n$

where $a_i, b_i \in \mathbb{N}$ for $i = 1, \ldots, n$. An *atomic formula* is either the string "true", the string "false", or a formula of the form $\vartheta_1 = \vartheta_2$, $\vartheta_1 < \vartheta_2$, or $\vartheta_1 > \vartheta_2$. A *formula* is built up from atomic formulas using conjunction, disjunction, negation, and the symbol $\exists$ in the usual manner. Formulas are interpreted in the obvious manner over $\mathbb{Q}$. For instance, the formula $\neg\exists x_1, x_2\ (x_1 < x_2) \wedge \neg\big(\exists x_3\ (x_1 < x_3 \wedge x_3 < x_2)\big)$ states that for every two different rational numbers, there exists a third rational number that lies strictly between them.

The *size* of a formula Φ is the sum of the number of brackets, Boolean connectives, the sizes of the variables, and the sizes of all rational constants occurring in Φ. Here, we assume that all rational constants are written as a/b, where a and b are integers, written in binary notation. We assume that variables are written as x_i, where i is written in binary notation.

Lemma 25 (Lemma 1 in [13]) *Let $\Phi(x_1, \ldots, x_n)$ be a quantifier-free formula. Then there exists a* PTIME *procedure for obtaining another quantifier-free formula, $\Phi'(x_1, \ldots, x_{n-1})$, such that*

$$\Phi'(x_1, \ldots, x_{n-1}) \text{ is equivalent to } \exists x_n \Phi(x_1, \ldots, x_n).$$

The following proposition is implicit in the work by Ferrante and Rackoff [13]. A full proof can be found as the proof of Proposition 3.4 in [21].

Proposition 26 *Let $\Phi(x_1, \ldots, x_n)$ be a quantifier-free formula. If n is fixed, then satisfiability of Φ over $\mathbb{Q}$ can be decided in* PTIME*. Moreover, if Φ is satisfiable, we can find $(v_1, \ldots, v_n) \in \mathbb{Q}^n$ such that $\Phi(v_1, \ldots, v_n)$ is true in polynomial time.*

The following proposition is a generalization of a well-known theorem by Lenstra which states that there exists a polynomial time algorithm to find an integer solution for a *conjunction* of linear (in)equalities with rational factors and a fixed number of variables [18].

Proposition 27 *There exists a* PTIME *algorithm that decides whether a Boolean combination of linear (in)equalities with rational factors and a fixed number of variables has an integer solution.*

PROOF. Note that we cannot simply put the Boolean combination into disjunctive normal form, as this would lead to an exponential increase of its size.

Let $\Phi(x_1, \ldots, x_n)$ be a Boolean combination of formulas $\varphi_1, \ldots, \varphi_m$ with variables $x_1, \ldots, x_n$ that range over $\mathbb{Z}$. Here, n is a constant integer greater than

zero. Without loss of generality, we can assume that every φ_i is of the form

$$k_{i,1} \times x_1 + \cdots + k_{i,n} \times x_n + k_i \geq 0,$$

where $k_i, k_{i,1}, \ldots, k_{i,n} \in \mathbb{Q}$.

We describe a PTIME procedure for finding a solution for $x_1, \ldots, x_n$, that is, for finding values $v_1, \ldots, v_n \in \mathbb{Z}$ such that $\Phi(v_1, \ldots, v_n)$ evaluates to true.

First, we introduce some notation and terminology. For every $i = 1, \ldots, m$, we denote by φ'_i the formula $k_{i,1} \times x_1 + \cdots + k_{i,n} \times x_n + k_i = 0$. In the following, we freely identify φ'_i with the hyperplane it defines in $\mathbb{R}^n$. For an n-tuple $\overline{y} = (y_1, \ldots, y_n) \in \mathbb{Q}^n$, we denote by $\varphi'_i(\overline{y})$ the rational number $k_{i,1} \times y_1 + \cdots + k_{i,n} \times y_n + k_i$.

Given a set of hyperplanes H in $\mathbb{R}^n$, we say that $C \subseteq \mathbb{R}^n$ is a *cell* of H when

(i) for every hyperplane φ'_i in H, and for every pair of points $\overline{y}$, $\overline{z} \in C$, we have that $\varphi'_i(\overline{y})\ \theta\ 0$ if and only if $\varphi'_i(\overline{z})\ \theta\ 0$, where, θ denotes "<", ">", or "="; and
(ii) there exists no $C' \supsetneq C$ with property (i).

Let H be the set of hyperplanes $\{\varphi'_i \mid 1 \leq i \leq m\}$.

We now describe the PTIME algorithm. The algorithm iterates over the following steps:

(1) Compute $(v'_1, \ldots, v'_n) \in \mathbb{Q}^n$ such that $\Phi(v'_1, \ldots, v'_n)$ is true.[6] If no such $(v'_1, \ldots, v'_n)$ exists, the algorithm rejects.
(2) For every $\varphi'_i \in H$, let $\theta_i \in \{<, >, =\}$ be the relation such that

$$k_{i,1} \times v'_1 + \cdots + k_{i,n} \times v'_n + k_i\ \theta_i\ 0.$$

For every $i = 1, \ldots, m$, let $\varphi''_i = k_{i,1} \times x_1 + \cdots + k_{i,n} \times x_n + k_i\ \theta_i\ 0$. So, for every $i = 1, \ldots, m$, φ''_i defines the half-space or hyperplane that contains the point $(v'_1, \ldots, v'_n)$.

Let $\Phi'(x_1, \ldots, x_n)$ be the conjunction

$$\bigwedge_{1 \leq i \leq n} \varphi''_i.$$

Notice that the points satisfying $\Phi'(x_1, \ldots, x_n)$ are precisely the points in the cell C of H that contains $(v'_1, \ldots, v'_n)$.
(3) Solve the *integer programming problem* for $\Phi'(x_1, \ldots, x_n)$. That is, find a $(v_1, \ldots, v_n) \in \mathbb{Z}^n$ such that $\Phi'(v_1, \ldots, v_n)$ evaluates to true.

[6] Note that we abuse notation here, as the variables in Φ range over $\mathbb{Z}$ and not $\mathbb{Q}$.

(4) If $(v_1, \ldots, v_n) \in \mathbb{Z}^n$ exists, then write $(v_1, \ldots, v_n)$ to the output and accept.
(5) If $(v_1, \ldots, v_n) \in \mathbb{Z}^n$ does not exist, then overwrite $\Phi(x_1, \ldots, x_n)$ with

$$\Phi''(x_1, \ldots, x_n) = \Phi(x_1, \ldots, x_n) \wedge \neg\Phi'(x_1, \ldots, x_n)$$

and go back to step (1).

We show that the algorithm is correct. Clearly, if the algorithm accepts, Φ has a solution. Conversely, suppose that Φ has a solution. Hence, the algorithm computes a value $(v_1', \ldots, v_n') \in \mathbb{Q}^n$ in step (1) of its first iteration. It follows from the following two observations that the algorithm accepts:

(i) If the algorithm computes $(v_1', \ldots, v_n') \in \mathbb{Q}^n$ in step (1), and the cell C of H containing $(v_1', \ldots, v_n')$ also contains a point in $\mathbb{Z}^n$, then step (3) finds a solution $(v_1, \ldots, v_n) \in \mathbb{Z}^n$; and,
(ii) If the algorithm computes $(v_1', \ldots, v_n') \in \mathbb{Q}^n$ in step (1), and the cell C of H containing $(v_1', \ldots, v_n')$ does *not* contain a point in $\mathbb{Z}^n$, then step (3) does *not* find a solution. By construction of Φ'' in step (5), the solutions to the formula Φ'' are the solutions of Φ, minus the points in C. As C did not contain a solution, we have that Φ has a solution if and only if Φ'' has a solution. Moreover, there exists no $(v_1'', \ldots, v_n'') \in C$ such that $\Phi''(v_1'', \ldots, v_n'')$ evaluates to true.

To show that the algorithm can be implemented to run in polynomial time, we first argue that there are at most a polynomial number of iterations. This follows from the observation in step (2) that the points satisfying $\Phi'(x_1, \ldots, x_n)$ are precisely all the points in a cell C of H. Indeed, when we do not find a solution to the problem in step (3), we adapt Φ to exclude all the points in cell C in step (5). Hence, in the following iteration, step (1) cannot find a solution in cell C anymore. It follows that the number of iterations is bounded by the number of cells in H, which is $\Theta(m^n)$ (see, e.g. [7], or Theorem 1.3 in [11] for a more recent reference).

Finally, we argue that every step of the algorithm can be computed in PTIME.

Step (1) can be solved by the quantifier elimination method of Ferrante and Rackoff (Lemma 25). Proposition 26 states that we can find $(v_1', \ldots, v_n')$ in polynomial time.

Step (2) is easily to be seen to be in PTIME: we only have to evaluate every φ_i' once on $(v_1', \ldots, v_n')$.

Step (3) can be executed in PTIME by Lenstra's algorithm for integer linear programming with a fixed number of variables [18].

Step (4) is in PTIME (trivial).

Step (5) replaces $\Phi(x_1, \ldots, x_n)$ by the formula $\Phi(x_1, \ldots, x_n) \wedge \neg\Phi'(x_1, \ldots, x_n)$. As the size of $\Phi'(x_1, \ldots, x_n)$ is bounded by n plus the sum of the sizes of φ_i'' for $i = 1, \ldots, n$, the formula Φ only grows by a linear term in each iteration. As the number of iterations is bounded by a polynomial, the maximum size of Φ is also bounded by a polynomial.

It follows that the algoritm is correct, and can be implemented to run in polynomial time. □

Corollary 28 *There exists a* PTIME *algorithm that decides whether a Boolean combination of linear (in)equalities with rational factors and a fixed number of variables has a solution of positive integers.*

PROOF. Given a Boolean combination $\Phi(x_1, \ldots, x_n)$ of linear (in)equalities with rational factors, we simply apply the algorithm of Proposition 27 to the formula

$$\Phi'(x_1, \ldots, x_n) = \Phi(x_1, \ldots, x_n) \wedge \bigwedge_{1 \leq i \leq n} x_i \geq 0.$$

□

In the following proposition, we treat the emptiness problem for DTDs: given a DTD d, is $L(d) = \emptyset$? Note that $L(d)$ can be empty even when d is not. For instance, the trivial grammar $a \to a$ generates no finite trees.

Proposition 29 *The emptiness problem is (1)* PTIME*-complete for DTD(NFA) and DTD(DFA), and (2)* coNP*-complete for DTD(SL).*

PROOF. (1) The upper bound follows from a reduction to the emptiness problem for NTA(NFA)s, which is in PTIME (cf. Theorem 19(1) in [23])

For the lower bound, we reduce from PATH SYSTEMS [10], which is known to be PTIME-complete. PATH SYSTEMS is the decision problem defined as follows: given a finite set of propositions P, a set $A \subseteq P$ of axioms, a set $R \subseteq P \times P \times P$ of inference rules and some $p \in P$, is p *provable from* A *using* R? Here, *(i)* every proposition in A is provable from A using R and, *(ii)* if $(p_1, p_2, p_3) \in R$ and if p_1 and p_2 are provable from A using R, then p_3 is also provable from A using R.

In our reduction, we construct a DTD (d, p) such that (d, p) is not empty if and only if p is provable. Concretely, for every $(a, b, c) \in R$, we add the string ab to $d(c)$; for every $a \in A$, $d(a) = \{\varepsilon\}$. Clearly, (d, p) satisfies the requirements.

(2) We provide an NP algorithm to check whether a DTD(SL) (d, r) defines a non-empty language. Intuitively, the algorithm computes the set $S = \{a \in \Sigma \mid L((d, a)) \neq \emptyset\}$ in an iterative manner and accepts when $r \in S$.

Let k be the largest integer occurring in any SL-formula in d. Initially, S is empty.

The iterative step is as follows. Guess a sequence of different symbols $b_1, \ldots, b_m$ in S. Then guess a vector $(v_1, \ldots, v_m) \in \{0, \ldots, k+1\}^m$, where k is the largest integer occurring in any SL-formula in d. Intuitively, the vector $(v_1, \ldots, v_m)$ represents the string $b_1^{v_1} \cdots b_m^{v_m}$. From Lemma 12 it follows that any SL-formula in d is satisfiable if and only if it is satisfiable by a string of the form $a_1^{u_1} \cdots a_n^{u_n}$, where $\Sigma = \{a_1, \ldots, a_n\}$, and for all $i = 1, \ldots, n$, $u_i \in \{0, \ldots, k+1\}$. Now add to S each $a \in \Sigma$ for which $b_1^{v_1} \cdots b_m^{v_m} \models d(a)$. Note that this condition can be checked in PTIME. Repeat the iterative step at most $|\Sigma|$ times and accept when $r \in S$.

The coNP-lowerbound follows from an easy reduction of non-satisfiability. Let ϕ be a propositional formula with variables $x_1, \ldots, x_n$. Let Σ be the set $\{a_1, \ldots, a_n\}$. Let (d, r) be the DTD where $d(r) = \phi'$, where ϕ' is the formula ϕ in which every x_i is replaced by $a_i^{=1}$. Hence, (d, r) defines the empty tree language if and only if ϕ is unsatisfiable. □

Reducing a grammar is the act of finding an equivalent reduced grammar.

Corollary 30 *Reducing a DTD(NFA) is* PTIME*-complete; and reducing a DTD(SL) is* NP*-complete.*

PROOF. We first show the upper bounds. Let (d, s) be a DTD(NFA) or DTD(SL) over alphabet Σ. In both cases, the algorithm performs the following steps for each $a \in \Sigma$:

(i) Test whether $L((d, a)) \neq \emptyset$. If this is not the case, remove a from the definition of the DTD.
(ii) Test whether a is *reachable* from s. That is, test whether there is a sequence of Σ-symbols $a_1, \ldots, a_n$ such that
- $a = s$ and $a_n = a$; and
- for every $i = 2, \ldots, n$, there exists a string $w_1 a_i w_2 \in d(a_{i-1})$, for $w_1, w_2 \in \Sigma^*$.

If this is not the case, remove a from the definition of the DTD.

Removing a symbol a from the definition of the DTD is done as follows. In the case of SL, every atom $a^{\geq i}$ and $a^{=i}$ is replaced by true when $i = 0$ and false otherwise. In the case of NFAs, every transition mentioning a is removed.

Further, the definition of $d(a)$ is removed and a is deleted from the alphabet of the DTD.

In the case of a DTD(NFA), step (i) is in PTIME, and step (ii) is in NLOGSPACE. In the case of a DTD(SL), both tests (i) and (ii) are in NP.

For the lower bound, we argue that

(1) if there exists an NLOGSPACE-algorithm for reducing a DTD(NFA), then emptiness of a DTD(NFA) is in NLOGSPACE; and,
(2) if there exists a PTIME-algorithm for reducing a DTD(SL), then emptiness of a DTD(SL) is in PTIME.

Statements (1) and (2) are easy to show: one only has to observe that an emptiness test of a DTD can be obtained by reducing the DTD and verifying whether the alphabet of the DTD still contains the start symbol. □

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] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems*, 29(4):710–751, 2004.

[4] V. Benzaken, A. Frisch, and G. Castagna. 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.

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

[6] 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.

[7] R. Buck. Partition of space. *American Mathematical Monthly*, 50(9):541–544, 1943.

[8] 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.

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

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

[11] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

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

[13] J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM Journal on Computing*, 4(1):69–76, 1975.

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

[15] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, A. R. Meyer, N. Nivat, M. S. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume A, chapter 2, pages 67–161. North-Holland, 1990.

[16] 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.

[17] 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.

[18] H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.

[19] S. Maneth and F. Neven. Structured document transformations based on XSL. In R. Connor and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming (DBPL 1999)*, volume 1949 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2000.

[20] S. Maneth, T. Perst, A. Berlea, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of the 24th Symposium on Principles of Database Systems (PODS 2005)*, pages 283–294. ACM Press, 2005.

[21] W. Martens. *Static Analysis of XML Transformation and Schema Languages*. PhD thesis, Hasselt University, 2006.

[22] 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.

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

[24] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. *Journal of Computer and System Sciences*, 73(3):362–390, 2007.

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

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

[27] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

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

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

[30] H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

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

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

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

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

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