T U M

I N S T I T U T F Ü R I N F O R M A T I K

XML Type Checking with Macro Tree Transducers

S. Maneth, A. Berlea, T. Perst, H. Seidl

TUM-I0407
April 04

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

XML Type Checking with Macro Tree Transducers

S. Maneth[1], A. Berlea[2], T. Perst[2], and H. Seidl[2]

[1] Swiss Federal Institute of Technology, Lausanne, Switzerland
[2] Technische Universität München, Garching, Germany

Abstract. The tree transformation language TL is introduced which incorporates full MSO-pattern-matching, arbitrary navigation through the input, and named procedures with accumulating parameters. Thus, TL essentially captures all features offered by existing document processing languages such as XSLT, *fxt*, or XDUCE. It is proved that TL, despite its expressiveness, still allows for effective inverse type inference. This result is obtained by means of a translation of TL transformers into compositions of (stay) macro tree transducers.

1 Introduction

The extensible markup language (XML) has developed into the de facto standard for data exchange on the Internet [1]. Conceptually, an XML document is a textual representation of a finite labeled unranked tree. In a given application context, however, not every such (representation of a) tree is meaningful. Even more, certain applications may rely on inputs conforming to a specific "type". A description of the type of XML documents can either be given by a *document type definition* (DTD) or is provided by an XML Schema specification [16]. DTD's as well as schemas correspond to (certain) regular tree grammars [26, 28]. In this paper we adopt this view and use regular tree languages as XML types.

Since XML is used as a generic exchange format for data on the Internet, XML documents rather than being statically known, are often produced on-the-fly by applications such as, e.g., converters between document formats or search engines returning their results in XML. In this paper we address the problem of *statically* type checking XML processors. That is, given a transformation f, a type T_{in} of input documents and a type T_{out} of output documents, we want to guarantee that for all $t \in T_{\text{in}}$ the result $f(t)$ is in T_{out}. Conformance to an output type is especially important when the output becomes input for a subsequent XML processor. Thus, powerful static type checking allows to detect programming errors at an early stage. In general, type checking algorithms are specific to a given transformation language and strongly depend on the operations used by the language for querying and navigating the input document. Even though there are many different XML query and transformation languages, their "tree transformation core" can be subsumed by a small number of basic operations. Milo et al. [25] proposed the k-pebble tree transducer (k-ptt) as such a core model for tree transformations. Specifying a tree translation using a k-ptt, however, can be cumbersome because the model is quite low-level. Also, its expressiveness is not yet completely clear. Here we take a different approach: We choose our transformation model as expressive as possible ('high-level'), while still allowing for effective type inference. We introduce the small but powerful transformation language TL which comprises virtually all features present in existing domain specific languages for document processing. In particular, TL provides full MSO (Monadic Second-Order logic) pattern matching and arbitrary navigation through the input tree. This generalizes restricted pattern languages such as XPath (for XSLT [10, 21] and XQuery [8]) or *fxgrep* patterns (for *fxt* [6, 27]). Moreover, TL offers a notion of named procedures together with accumulating parameters. These make it straightforward to express powerful fold operations over forests which behave differently for different occurrences of the same elements as provided, e.g., by XDuce [20].

Let us take a look at an example of a TL transformation. Consider an Xml description of new mails where a spam filter has marked each dubious message header by a `spam` element:

```
<mailbox>
  <mail><subject>As seen on TV</subject></mail>
  <spam>99%</spam>
  <mail><subject>Adding to your spam load:-)</subject></mail>
  <mail><subject>Make money</subject></mail>
  <spam>95%</spam>
</mailbox>
```

Imagine that we want to obtain the following list of spam mails where the `spam` element has been moved inside of the mail:

```
<mail><spam>99%</spam><subject>As seen on TV</subject></mail>
<mail><spam>95%</spam><subject>Make money</subject></mail>
```

We can express this transformation in TL by the following rules:

$$\begin{array}{ll}
q_0(x_1 \in \mathsf{lab}_{\mathsf{mailbox}}) & \Rightarrow q_0(x_1/x_2) \\
q_0(x_1 \in \mathsf{lab}_{\mathsf{mail}} \wedge \mathit{spam}(x_1)) & \Rightarrow \mathsf{mail}\langle \mathit{copyT}(x_1;x_2)\ \mathit{copyF}(x_1/x_2)\rangle\ q_0(x_1;x_2) \\
q_0(x_1 \in \mathsf{lab}_{\mathsf{mail}} \wedge \neg\mathit{spam}(x_1)) & \Rightarrow q_0(x_1;x_2) \\
q_0(x_1 \in \mathsf{lab}_{\mathsf{spam}}) & \Rightarrow q_0(x_1;x_2) \\
q_0(x_1 \in \mathsf{lab}_{\epsilon}) & \Rightarrow \epsilon
\end{array}$$

Here, the MSO formulas to the left describe the nodes in the input tree where the corresponding rule is applicable. In particular, we have used the abbreviation $\mathit{spam}(x_1)$ for the sub-formula $\exists z.\ x_1;z \wedge z \in \mathsf{lab}_{\mathsf{spam}}$ denoting all nodes x_1 having a right sibling z labeled with `spam`. Similarly, x_1/x_2 in the right-hand side of a rule selects the first child of x_1 (and $x_1;x_2$ the right sibling); these nodes are further processed by the $\mathit{copyT}()$ and $\mathit{copyF}()$ transformations, respectively:

$$\begin{array}{ll}
\mathit{copyT}(x_1 \in \mathsf{lab}_{\alpha}) & \Rightarrow \alpha\langle \mathit{copyF}(x_1/x_2)\rangle \\
\mathit{copyF}(x_1 \in \mathsf{lab}_{\epsilon}) & \Rightarrow \epsilon \\
\mathit{copyF}(x_1 \notin \mathsf{lab}_{\epsilon}) & \Rightarrow \mathit{copyT}(x_1 = x_2)\ \mathit{copyF}(x_1;x_2)
\end{array}$$

for every possible input tag α. The same transformation can be expressed in XSLT [10, 21] as:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match='mail[name(following-sibling::*[1])="spam"]'>
  <mail>
    <xsl:copy-of select="following-sibling::spam[1]"/>
    <xsl:copy-of select="*"/>
  </mail>
</xsl:template>
<xsl:template match="spam|mail"/>
</xsl:stylesheet>
```

The first rule refers to `mail` elements which are immediately followed by `spam` elements and produces a `mail` element whose content is given by two copy-instructions. The first produces a copy of the `spam` sibling, while the second produces a copy of each of the children. The second rule prevents the processor from producing output for `spam` and other `mail` elements. A built-in default rule ensures that for all other elements output is produced by recursively processing their child nodes using the transformation rules.

Similarly in *fxt* [4], the transformation can be written as:

```
<fxt:spec>
  <fxt:pat>//*[# %spam]/mail</fxt:pat>
    <mail>
      <fxt:copyContent select="1"/>
      <fxt:copyContent/>
    </mail>
  <fxt:pat>//.</fxt:pat>
    <fxt:apply/>
</fxt:spec>
```

The pattern in the first rule specifies, analogously to the XSLT example, that the rule applies to spam mails. Here the pattern `//*[# %spam]/mail` selects `mail` elements followed by an element `spam`. Supplementary, the extra marker "%" at `spam` gives a reference to the next `spam` sibling for later use in the subsequent transformation. The child nodes of `mail` are copied by the `fxt:copyContent` instruction. The second rule is applicable to all other elements and is similar to the XSLT default rule. It specifies that output is produced by recursively applying the transformation rules to the child nodes of those elements.

Thus, abstracting away syntactic sugar such as various built-ins for copying parts of the input, both XSLT and *fxt* essentially are rule-based recursive transformer languages which use (more or less expressive) pattern languages for restricting the applicability of rules as well as for selecting the nodes to be further processed. These features easily can be simulated by TL.

The key idea of our type checking algorithm is to simulate TL transformations by compositions of macro tree transducers (mtt's). The macro tree transducer [14] is an extensively studied model of syntax directed semantics [18]. In particular, macro tree translations (and compositions thereof) can be type checked by using inverse type inference. The latter means that, given an output type T_{out}, the set $\tau^{-1}(T_{\text{out}})$ of possible input trees is again a regular tree language, and can be obtained effectively. Since regular tree languages are closed under complement and have decidable inclusion, we obtain a type checking algorithm. In fact, this approach was used by Engelfriet and Maneth in [13] in order to show that k-ptt's can be type checked: They prove that any k-ptt can be decomposed into a composition of $k+1$ mtt's. In order to compare this to our decomposition of TL-transformations, let us take a closer look at a k-pebble tree transducer. While traversing an input tree, a k-ptt can mark nodes using up to k different pebbles. In the proposed implementation of XPATH pattern matching through k-ptt's, the number of used pebbles corresponds to the number of nodes in the pattern. This means that the number of mtt's used in the decomposition of Engelfriet and Maneth increases when patterns get more complicated. The complexity, on the other hand, of inverse type inference of a composition of k mtt's depends sensitively on the number k. On the contrary, we show that every TL transformation can essentially be realized by a composition of three mtt's only — no matter how complicated the used patterns are.

This paper is organized as follows: In the next section we introduce the notion of 2-way macro tree transducers. In the third section we introduce the notion of a TL transformer and prove that every such transformation can be realized by a composition of (stay) mtt's. The following section investigates deterministic variants of TL. Then we show that TL transformers can be effectively type checked. Finally, we present related work and conclude.

2 Trees and Macro Tree Transducers

A *ranked alphabet* is a finite set Σ together with a mapping $\text{rank}_\Sigma : \Sigma \to \mathbb{N}$. For $\sigma \in \Sigma$ and $k \geq 0$ we also write $\sigma^{(k)}$ to denote that $\text{rank}_\Sigma(\sigma) = k$, and we define the set $\Sigma^{(k)} = \{\sigma \in \Sigma \mid \text{rank}_\Sigma(\sigma) = k\}$. The set $\mathcal{T}_\Sigma$ of *ranked trees over* Σ is recursively defined to consist of all

$$t ::= a(t_1, \ldots, t_k)$$

where $a \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$. If $k = 0$, we also write a instead of $a()$. Alternatively, each tree can be considered as a mapping $t : \mathcal{N}(t) \to \Sigma$ where $\mathcal{N}(t) \subseteq \mathbb{N}^*$ is the set of nodes of t. This set is recursively defined by:

$$\mathcal{N}(a(t_1, \ldots, t_k)) = \{\epsilon\} \cup \{iu \mid i = 1, \ldots, k \wedge u \in \mathcal{N}(t_i)\}.$$

Thus, the empty sequence ϵ denotes the root node of t, and ui denotes the i-th child of the node u. Now, for a node $u \in \mathcal{N}(t)$ and $t = a(t_1, \ldots, t_k)$ with $a \in \Sigma^{(k)}$, $k \geq 0$, and $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$, $t(u) = a$ if $u = \varepsilon$, and otherwise $t(u) = t_i(u')$ where $1 \leq i \leq k$ and $u' \in \mathcal{N}(t)$ such that $u = iu'$. The lexicographical order on $\mathbb{N}^*$ induces a total ordering "$<$" on the nodes of t which is also called *document order*. The *subtree* of t rooted at a node $v \in \mathcal{N}(t)$ (denoted by t/v) is defined by $t/\epsilon = t$ and $t/iv = t_i/v$ if $t = a(t_1, \ldots, t_k)$ and $1 \leq i \leq k$.

Definition 1 A *2-way macro tree transducer* (2-mtt for short) is a tuple $M = (Q, \Sigma, \Delta, q_0, R)$ where Q is a ranked alphabet of states, Σ and Δ are ranked input and output alphabets, respectively, $q_0 \in Q^{(1)}$ is the *initial state*, and R is a finite set of rules of the form

$$a \ : \quad q(\pi, y_1, \ldots, y_m) \to t$$

where $a \in \Sigma^{(k)}$, $q \in Q^{(m+1)}$, $m, k \geq 0$, and the right-hand side t is of the form:

$$\begin{aligned} t ::= \ & q'(\pi^-, t_1, \ldots, t_n) \\ & \mid q'(\pi, t_1, \ldots, t_n) \\ & \mid q'(\pi j, t_1, \ldots, t_n) \\ & \mid b(t_1, \ldots, t_l) \\ & \mid y_i \end{aligned}$$

where $q' \in Q^{(n+1)}$, $n \geq 0$, $j \in \{1, \ldots, k\}$, $b \in \Delta^{(l)}$, and $1 \leq i \leq m$.

Note that $q'(\pi j, \ldots)$ in a right-hand side denotes a transition into state q' at the j-th child of the current node; correspondingly π denotes the current node and π^- denotes the parent of the current node. The 2-mtt is called 0-pmtt in [13].

If R does not contain subterms of the form $q'(\pi^-, t_1, \ldots, t_n)$, then M is called *stay macro tree transducer* (s-mtt) [13]. If, additionally, there are also no subterms $q'(\pi, t_1, \ldots, t_n)$, then the definition is equivalent to the classical notion of a macro tree transducer (mtt). If no states have accumulating parameters, we obtain *2-way top-down tree transducers* (2-top), *stay top-down tree transducers* (s-top), and (ordinary) *top-down tree transducers* (top), respectively.

For a given input tree $t \in \mathcal{T}_\Sigma$, a state $q \in Q^{(k+1)}$ of the 2-mtt M can be considered as function $[\![q]\!]_t$ which takes a node $v \in \mathbb{N}^*$ of t and a list $\underline{T} = T_1, \ldots, T_k$ of sets $T_i \subseteq \mathcal{T}_\Delta$ as actual parameters and returns a set of output trees. This function is defined as the least fixpoint of the constraints:

$$\begin{aligned} [\![q]\!]_t(v, \underline{T}) &\supseteq [\![t']\!]_t \ v \ \eta && \text{if} \quad t(v) = a \wedge a : \ q(\pi) \to t' \in R \\ & && \text{and} \quad \eta(y_i) = T_i \text{ for } i = 1, \ldots, k \\ [\![y_j]\!]_t \ v \ \eta &\supseteq \eta(y_j) \\ [\![a(t_1, \ldots, t_m)]\!]_t \ v \ \eta &\supseteq a([\![t_1]\!]_t \ v \ \eta, \ldots, [\![t_m]\!]_t \ v \ \eta) \\ [\![p(\pi, t_1, \ldots, t_n)]\!]_t \ v \ \eta &\supseteq [\![p]\!]_t(v, [\![t_1]\!]_t \ v \ \eta, \ldots, [\![t_n]\!]_t \ v \ \eta) \\ [\![p(\pi^-, t_1, \ldots, t_n)]\!]_t \ vi \ \eta &\supseteq [\![p]\!]_t(v, [\![t_1]\!]_t \ v \ \eta, \ldots, [\![t_n]\!]_t \ v \ \eta) \\ [\![p(\pi i, t_1, \ldots, t_n)]\!]_t \ v \ \eta &\supseteq [\![p]\!]_t(vi, [\![t_1]\!]_t \ v \ \eta, \ldots, [\![t_n]\!]_t \ v \ \eta) && \text{if} \quad vi \in \mathcal{N}(t) \end{aligned}$$

where η is a mapping from symbols y_i, $i = 1, \ldots, k$ of formal parameters to actual parameters, i.e., sets $T_i \subseteq \mathcal{T}_\Delta$. Moreover for a constructor $a \in \Delta^{(k)}$, we have used the notation $a(T_1, \ldots, T_k)$ for sets $T_i \subseteq \mathcal{T}_\Delta$ to represent the set: $a(T_1, \ldots, T_k) = \{a(s_1, \ldots, s_k) \mid s_i \in T_i\}$. In case of 2-mtt's or s-mtt's, this definition might indeed be circular. Then the least fixpoint of these equations is obtained by starting with the empty set for each call $[\![q]\!]_t(v, \underline{T})$ and then successively iterating up to termination.

The *transduction* defined by M then is given by:

$$\tau_M = \{(t, s) \mid t \in \mathcal{T}_\Sigma, s \in [\![q_0]\!]_t \ (\epsilon)\}.$$

The class of all transductions induced by 2-mtt's (s-mtt's, mtt's, 2-top's, s-top's, top's) is denoted by 2-MTT (S-MTT, MTT, 2-TOP, S-TOP, TOP).

Note that we have not fixed any particular order in which recursive occurrences of $[\![..]\!]$ are to be evaluated. This is called the "unrestricted" derivation mode in [15]; the class of translations remains unchanged if we use the OI (outside-in, or lazy) derivation mode. If for every pair (a, q), we have exactly one rule and $[\![q_0]\!]_t(\epsilon) \neq \emptyset$ for every $t \in \mathcal{T}_\Sigma$, the transducer is called *total deterministic* which we indicate by the prefix D_t. The total deterministic variant is essentially equivalent to *macro attributed tree transducers* of [22, 18]. Note that total deterministic transducers in fact compute total *functions*, i.e., for every tree t, $[\![q_0]\!]_t(\epsilon)$ contains exactly one element. Moreover, in the total deterministic case the order of derivation does not influence the class of translations.

The type of tree substitution supported through the use of parameters can be captured by a class of functions called YIELD. The YIELD function interprets special leaf labels $\alpha_1, \ldots, \alpha_m$ as variables and the special symbols σ_m (of rank $m+1$) as formal substitution operations. Formally, YIELD is defined as YIELD_0; for $n \geq 0$, YIELD_n has n extra arguments $\underline{y} = y_1, \ldots, y_n$ and is defined as:

$$\text{YIELD}_n\ (\alpha_j, \underline{y}) = \begin{cases} y_j & \text{if} \quad j \leq n \\ \perp & \text{otherwise} \end{cases}$$

$$\text{YIELD}_n\ (\sigma_m(t_0, t_1, \ldots, t_m), \underline{y}) = \text{YIELD}_m(t_0, \text{YIELD}_n(t_1, \underline{y}), \ldots, \text{YIELD}_n(t_m, \underline{y}))$$

$$\text{YIELD}_n\ (a(t_1, \ldots, t_k), \underline{y}) = a(\text{YIELD}_n(t_1, \underline{y}), \ldots, \text{YIELD}_n(t_k, \underline{y}))$$

where $m \geq 0$, $1 \leq j \leq m$, $a \in \Sigma$ for some input ranked alphabet Σ, and $\perp$ is a new nullary symbol denoting the illegal use of a non-existing formal parameter (and hence it stands for "undefined"). Thus, given m and Σ, YIELD is a total function from trees over $\Sigma \cup \{\alpha_1^{(0)}, \ldots, \alpha_m^{(0)}\} \cup \{\sigma_1^{(1)}, \ldots, \sigma_m^{(m+1)}\}$ to trees over $\Sigma \cup \{\perp^{(0)}\}$. The function YIELD can be realized by a total deterministic mtt (in fact, even by a total deterministic 2-top; see, e.g., Lemma 36 of [13]).

Treating Up Moves for Parameters. In this section it is shown how a 2-top can be transformed into a transducer that does not use up instructions (π^-) in its right-hand sides, but which additionally uses parameters (i.e., a stay mtt). Since the result is well known, we merely point to the corresponding literature and give a short explanation of the proof.

Lemma 2 [13] 2-TOP $\subseteq$ S-MTT and D_t2-TOP $\subseteq$ D_tMTT.

The intuition behind the proofs is this one: Instead of moving up to a node u that has already been processed, we generate, at u, all possible state calls and pass them as parameters to any further calls. This is done in a recursive, stack like fashion. In this way an up move into a state q can be simulated by selecting the parameter y_j that corresponds to q. In the total deterministic case even the stay moves can be eliminated, by computing the corresponding tree (or deleting the corresponding rule if there is a circularity).

Dealing with Stay Moves. Stay moves can be eliminated if we process the input tree in the following way. We add above each symbol σ in the input tree an arbitrary number of monadic $\bar{\sigma}$'s. Let us denote this translation by MON_Σ, where Σ is a ranked alphabet, and denote by MON the class of all such translations (for all alphabets Σ). Obviously, MON_Σ can be realized by an s-top. Then, as shown in Lemma 27 of [13], every s-mtt (with input alphabet Σ) can be simulated by the composition of MON_Σ followed by an mtt. In fact, the proof given there preserves the number of parameters, and therefore we obtain a similar result for top-down tree transducers, as stated in the next lemma.

Lemma 3 [13] $\text{s-MTT} \subseteq \text{MTT} \circ \text{MON}$ and $\text{s-TOP} \subseteq \text{TOP} \circ \text{MON}$.

Eliminating Parameters. As mentioned before, YIELD mappings realize the type of tree substitution inherent in the use of parameters in a transducer. For macro tree transducers this relationship is expressed by the fact that $\text{D}_t\text{MTT} = \text{YIELD} \circ \text{D}_t\text{TOP}$ (proved in [15]). In the partial case, the YIELD function as defined before is not sufficient. But introducing a new "failure" symbol $\bullet$ which serves as right-hand side for any state and input symbol solves the problem. In this way, YIELD can evaluate a tree containing failure symbols, if these are in parameter positions which are deleted. For this to work we extend the input alphabet of YIELD with a new nullary symbol $\bullet$ without, however, adding rules for $\bullet$. The resulting function (and class of functions) is denoted $\text{YIELD}^\bullet$.

Lemma 4 $\text{2-MTT} \subseteq \text{YIELD}^\bullet \circ \text{2-TOP}$ and $\text{D}_t(\text{2-MTT}) \subseteq \text{YIELD} \circ \text{D}_t(\text{2-TOP})$.

Proof. The idea is essentially the same as in the proof of Theorem 3.26 in [15]. Let $M = (Q, \Sigma, \Delta, q_0, R)$ be a 2-mtt. We construct the 2-top $T = (Q', \Sigma, \Delta', q_0, R')$ such that $\text{YIELD}^\bullet \circ \tau_T = \tau_M$. For every $q \in Q^{(k+1)}$ and $a \in \Sigma$, let the rule $a \ : \ q(\pi) \to \bullet$ be in R' and for every rule $a \ : \ q(\pi, y_1, \ldots, y_k) \to t$ in R let the rule $a \ : \ q(\pi) \to [t]$ be in R' where $[.]$ is defined as:

$$\begin{aligned}
[a(t_1, \ldots, t_k)] &= a([t_1], \ldots, [t_k]) \\
[y_i] &= \alpha_i \\
[q(\pi, t_1, \ldots, t_n)] &= \sigma_n(q(\pi), [t_1], \ldots, [t_n]) \\
[q(\pi^-, t_1, \ldots, t_n)] &= \sigma_n(q(\pi^-), [t_1], \ldots, [t_n]) \\
[q(\pi i, t_1, \ldots, t_n)] &= \sigma_n(q(\pi i), [t_1], \ldots, [t_n])
\end{aligned}$$

Note that, due to the rules $a : q(\pi) \to \bullet$, the tree $\bullet$ is contained in $[\![q]\!]_t(v)$ for all $t \in \mathcal{T}_\Sigma$ and $v \in \mathcal{N}(t)$. We claim that for all $t \in \mathcal{T}_\Sigma, v \in \mathcal{N}(t), q \in Q, T_i \subseteq \mathcal{T}_\Delta$,

$$[\![q]\!]_t(v, T_1, \ldots, T_k) = \text{YIELD}_k([\![q]\!]_t\ (v), T_1, \ldots, T_k)$$

The proof is by fixpoint induction. For this, we simultaneously prove by structural induction on subterms of right-hand sides s:

$$[\![s]\!]_t\, v\, \eta = \text{YIELD}_k([\![[s]]\!]_t\ v\ \emptyset, \eta(y_1), \ldots, \eta(y_k))$$

In the case that M is total deterministic, the construction can be simplified by removing the rules $a : q(\pi) \to \bullet$ from the 2-top T. This makes T total deterministic and allows to use the original total function YIELD. $\diamond$

The combination of Lemma 4 with Lemma 2 gives us the following decompositions of 2-way macro tree transducers.

Lemma 5 $2\text{-}\textsc{mtt} \subseteq \textsc{yield}^\bullet \circ \textsc{s-mtt}$ and $\textsc{d}_t(2\text{-}\textsc{mtt}) \subseteq \textsc{yield} \circ \textsc{d}_t\textsc{mtt}$.

3 Forests, Queries and Transformations

An XML document is usually represented as *forest.* A forest (over Σ) consists of a sequence of *trees*, written $t_1 t_2 \cdots$, and a tree consists of a root node (labeled by some label $a \in \Sigma$) and a forest f (the children), written as $a\langle f\rangle$. A forest can also conveniently be viewed as a binary tree: Take the first child relation as left child and the next sibling relation as right child (thus, the forest $a\langle\epsilon\rangle\ \epsilon$ becomes the binary tree $a(\epsilon,\epsilon)$). In fact, in the sequel we will use this binary tree view of a forest. Then, for a forest f, $\mathcal{N}(f)$ is the set of nodes of the corresponding binary tree, and for a node u its label is $f(u)$. In the sequel, we therefore do no longer distinguish between forests and specific binary trees, namely those where all symbols are either binary (for labeling forest nodes) or equal to the nullary symbol ϵ (denoting the empty forest).

Forests, as sequences, support concatenation. We will later need a symbolic way to represent concatenation and introduce therefore the function APP which interprets the special symbol "@" (of rank 2) as concatenation. It is defined for binary trees. For better readability @ is written in infix notation. For a binary ranked input alphabet Σ, APP is the total function from trees t over $\Sigma \cup \{@^{(2)}\}$ to trees over Σ defined as $\textsc{app}(t) = \textsc{app}_1(t,\epsilon)$ where

$$\begin{aligned}
\textsc{app}_1(\epsilon, t) &= t\\
\textsc{app}_1(a(t_1,t_2),t) &= a(\textsc{app}_1(t_1,\epsilon), \textsc{app}_1(t_2,t))\\
\textsc{app}_1(t_1 @ t_2, t) &= \textsc{app}_1(t_1, \textsc{app}_1(t_2,t))
\end{aligned}$$

Obviously, the function APP can be realized by a total deterministic mtt.

An MSO formula ϕ (over alphabet Σ) is given by the grammar:

$$\begin{aligned}
\phi ::= {} & x;x' \mid x/x' \mid x \in \mathsf{lab}_a \mid x \in X \mid\\
& \phi_1 \vee \phi_2 \mid \neg\phi \mid\\
& \exists\, x.\ \phi \mid \exists\, X.\ \phi
\end{aligned}$$

Here, x and X are individual and set variables, respectively. Individual variables x range over nodes of a tree in $\mathcal{T}_\Sigma$ and set variables X over node sets. The binary relation symbols ";" and "/" denote *next sibling* and *first child* in the forest, respectively. The set lab_a denotes all nodes labeled with the symbol $a \in \Sigma$. As usual, we feel free to use abbreviations such as "$x_1 \wedge x_2$" and $\forall\, x.\ \phi$ in example formulas. For a given forest f, assignments ρ_1, ρ_2 of (supersets of) the individual and set variables occurring free in a formula ϕ, to nodes and node sets, respectively, the satisfiability assertion $\rho_1, \rho_2 \models_f \phi$ as usual is defined by:

$$\begin{array}{lll}
\rho_1, \rho_2 \models_f x; x' & \text{iff} & \rho_1(x') = \rho_1(x)\,2 \\
\rho_1, \rho_2 \models_f x/x' & \text{iff} & \rho_1(x') = \rho_1(x)\,1 \\
\rho_1, \rho_2 \models_f x \in \mathsf{lab}_a & \text{iff} & f(\rho_1(x)) = a \text{ and } a \in \Sigma \\
\rho_1, \rho_2 \models_f x \in X & \text{iff} & \rho_1(x) \in \rho_2(X) \\
\rho_1, \rho_2 \models_f \phi_1 \vee \phi_2 & \text{iff} & \rho_1, \rho_2 \models_f \phi_1 \text{ or } \rho_1, \rho_2 \models_f \phi_2 \\
\rho_1, \rho_2 \models_f \neg\phi & \text{iff} & \text{not } \rho_1, \rho_2 \models_f \phi \\
\rho_1, \rho_2 \models_f \exists\, x.\, \phi & \text{iff} & \rho_1 \oplus \{x \mapsto v\}, \rho_2 \models_f \phi \text{ for some } v \in \mathcal{N}(f) \\
\rho_1, \rho_2 \models_f \exists\, X.\, \phi & \text{iff} & \rho_1, \rho_2 \oplus \{X \mapsto V\} \models_f \phi \text{ for some } V \subseteq \mathcal{N}(f)
\end{array}$$

where "$\oplus$" is the operation which updates/extends an environment with the new bindings to the right. If ϕ does not contain free occurrences of set variables X, we also omit variable assignment ρ_2 in satisfiability assertions. If the only free variables are $x_1, \ldots, x_k$, ϕ can be considered as a k-ary *query* which, for each forest f, returns the set of all tuples $(v_1, \ldots, v_k) \in \mathcal{N}(f)^k$ such that $\{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\} \models_f \phi$. It is possible to implement k-ary queries by extending the classical construction of [30, 31] which constructs from a closed MSO formula (i.e., one without free variables) a finite tree automaton. For our applications, the most interesting queries are *unary* or *binary*, i.e., where $k = 1$ or $k = 2$, respectively. Unary queries are a suitable abstraction of *match* patterns which allow to determine which alternative action for a state q should be selected at a given node v [29]. Binary queries, on the other hand, are a powerful and convenient technique to select, depending on some given node (the current node), suitable next nodes where the transformation should proceed [5].

Before the result of [30] (adapted to unary and binary queries) is stated, we need to define the notion of finite (bottom-up) tree automaton. Here we are only interested in binary trees over $\Sigma = \Sigma^{(2)} \cup \{\epsilon^{(0)}\}$ for some set $\Sigma^{(2)}$ of binary symbols. Then, a (*bottom-up finite state*) *tree automaton* is a tuple $A = (S, \Sigma, \delta, F)$ where S is a set of *states*, $F \subseteq S$ is a set of *final states*, and $\delta \subseteq (\{\epsilon\} \times S) \cup \Sigma^{(2)} \times S \times S \times S$ is the transition relation. A transition (a, s, s_1, s_2) denotes that if A arrives in s_1, s_2 for two trees t_1, t_2, respectively, then it can arrive in state s for the tree $a(t_1, t_2)$. A *run* of A on a tree $t \in \mathcal{T}_\Sigma$ is a mapping $r : \mathcal{N}(t) \to S$ which is defined inductively for $u \in \mathcal{N}(t)$ as follows: if u is a leaf then $r(u) \in \{s \in S \mid (\epsilon, s) \in \delta\}$, and otherwise $r(u) \in \{s \in S \mid \exists s_1, s_2 \in S$ such that $(a, s_1, s_2, s) \in \delta$ and $s_1 \in r(u1)$ and $s_2 \in r(u2)\}$. The tree language accepted by A consists of the trees $t \in \mathcal{T}_\Sigma$ for which there is an accepting run, i.e., a run r with $r(\varepsilon) \in F$.

Lemma 6 Let $\phi_1, \ldots, \phi_k$ and $\psi_1, \ldots, \psi_m$ be sequences of unary and binary MSO formulas, respectively. Then a tree automaton $A = (S, \Sigma, \delta, F)$ can be constructed together with sequences $U_1, \ldots, U_k \subseteq S$ and $B_1, \ldots, B_m \subseteq S^2$ such that for every forest f the following holds:

1. $\{x_1 \mapsto v\} \models_f \phi_i$ iff there is an accepting run r of A on t such that $r(v) \in U_i$;
2. $\{x_1 \mapsto v_1, x_2 \mapsto v_2\} \models_f \psi_j$ iff there is an accepting run r of A on t such that $(r(v_1), r(v_2)) \in B_j$.

Definition 7 A TL transformer F is a tuple $(Q, \Sigma, \Delta, q_0, R)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ and Δ are the input and output alphabets, respectively. The finite set R contains rules of the form:

$$q(\phi, y_1, \ldots, y_k) \Rightarrow A$$

where an action A is recursively defined by:

$$A ::= \epsilon \mid A\ A \mid a\langle A\rangle\ A \mid y_j \mid q'(\psi, A_1, \ldots, A_m)$$

Here, ϕ and ψ are MSO formulas with free variables x_1 and x_1, x_2, respectively.

For a given input forest $f \in \mathcal{T}_\Sigma$, a state $q \in Q$ of a TL transformer can be considered as function $[\![q]\!]_f$ which takes a node $v \in \mathbb{N}^*$ of f and a list $\underline{F} = F_1, \ldots, F_k$ of sets $F_i \subseteq \mathcal{T}_\Delta$ of actual parameters and returns a set of output forests. This function is defined as the least fixpoint of:

$$[\![q]\!]_f(v, \underline{F}) \supseteq [\![A]\!]_f\ v\ \eta$$

if $v \models_f \phi$, $q(\phi, y_1, \ldots, y_k) \Rightarrow A$ and $\eta(y_i) = F_i$ for $i = 1, \ldots, k$ where

$$\begin{array}{rl}
[\![\epsilon]\!]_f\ v\ \eta & \supseteq \{\epsilon\} \\
[\![y_j]\!]_f\ v\ \eta & \supseteq \eta(y_j) \\
[\![A_1 A_2]\!]_f\ v\ \eta & \supseteq ([\![A_1]\!]_f\ v\ \eta)\ ([\![A_2]\!]_f\ v\ \eta) \\
[\![a\langle A_1\rangle\ A_2]\!]_f\ v\ \eta & \supseteq a\langle [\![A_1]\!]_f\ v\ \eta\rangle\ ([\![A_2]\!]_f\ v\ \eta) \\
[\![q'(\psi, A_1, \ldots, A_m)]\!]_f\ v\ \eta & \supseteq ([\![q']\!]_f(v_1, \underline{F}')) \ldots ([\![q']\!]_f(v_l, \underline{F}')) \\
& \quad \text{for} \quad \underline{F}' = [\![A_1]\!]_f\ v\ \eta, \ldots, [\![A_m]\!]_f\ v\ \eta \\
& \quad \text{if } \{v_1 < \ldots < v_l\} = \{v' \mid \{x_1 \mapsto v, x_2 \mapsto v'\} \models_f \psi\}
\end{array}$$

Note that we have denoted the concatenation of sets $F_1, \ldots, F_m$ by juxtaposition, i.e., $F_1 \ldots F_m = \{s_1 \ldots s_m \mid s_i \in F_i\}$. The translation induced by the TL transformer then is given by:

$$\tau_F = \{(f, s) \in \mathcal{T}_\Sigma \times \mathcal{T}_\Delta \mid s \in [\![q_0]\!]_f(\epsilon)\}.$$

According to this definition, a 2-mtt on forests is a (particularly simple) TL transformer where the match patterns ϕ only test for the label of the current node, i.e., are of the form $x_1 \in \mathsf{lab}_a$ for some a. Moreover, the navigation patterns only allow to stay at the same node ($x_1 = x_2$), to proceed to the first or second son (x_1/x_2 and $x_1; x_2$, respectively) or to the father of the current node ($x_2/x_1 \vee x_2; x_1$).

A 2-mtt is essentially able to implement a TL transformation — given that the input has been decorated so that the 2-mtt can decide at each node whether a pattern matches or not. For this decoration, we use a bottom-up followed by a top-down relabeling. A (total deterministic) top-down relabeling is a total deterministic top-down tree transducer, each rule of which is of the form $a : q(\pi) \rightarrow b(q_1(\pi 1), \ldots, q_k(\pi k))$ where a is an input symbol of rank k and b is an output symbol of rank k. The class of (total deterministic) top-down and bottom-up relabelings is denoted by T-REL and B-REL, respectively (see, e.g., [12]).

The idea of the implementation of a TL-transformation through (a composition of) stay mtt's is as follows Let $\mathcal{T}$ be a TL-transducer. According to Lemma 6, we assume that the unary and binary MSO queries contained in $\mathcal{T}$ have been compiled into one nondeterministic automaton $\mathcal{P}$ together with a set U_ϕ of states of $\mathcal{P}$ for each occurring unary query ϕ and a relation B_ψ on the states for each occurring binary query.

1. In a bottom-up relabeling each node v is decorated with the two sets S_1 and S_2 of states of $\mathcal{P}$ that can be assigned to the left and right descendent of v in all possible runs of $\mathcal{P}$.
2. Each node v additionally receives its child number together with the set $T(v)$ of states of $\mathcal{P}$ indicating which unary patterns ϕ match at v. This relabeling can be implemented by means of a top-down relabeling.
3. The 2-mtt works as follows: If a TL expression $q(\psi, \underline{A})$ is evaluated for a node v, the 2-mtt successively proceeds to the root where it returns a (representation of) the list of the forests obtained by recursively applying the transformation q to all v' where (v, v') satisfies ψ.

Lemma 8 TL $\subseteq$ APP $\circ$ 2-MTT $\circ$ T-REL $\circ$ B-REL

Proof. Let $\mathcal{T} = (Q, \Sigma, \Delta, q_0, R)$ be an TL-transducer. According to Lemma 6, we assume that the unary and binary MSO queries contained in $\mathcal{T}$ have been compiled into one nondeterministic automaton $\mathcal{P} = (S, \Sigma, \delta, F)$ together with one set $U_\phi \subseteq S$ for each occurring unary query ϕ and one relation $B_\psi \subseteq S^2$ for each occurring binary query. For each forest f, let $S(f) \subseteq S$ denote the set of states s such that there is a run r of $\mathcal{P}$ on f with $r(\epsilon) = s$. Recall that $S(f)$ can also be recursively defined by:

$$
\begin{aligned}
S(\epsilon) &= \{s \in S \mid (\epsilon, s) \in \delta\}\\
S(a\langle f_1\rangle\ f_2) &= \{s \in S \mid \exists\, s_i \in S(f_i) :\ (a, s, s_1, s_2) \in \delta\}
\end{aligned}
$$

The implementation of $\mathcal{T}$ is done in three steps:

1. Each node v of the input forest f labeled with symbol $a \in \Sigma$ is relabeled with $(a, S(f/v1), S(f/v2))$. According to the recursive definition of the sets $S(f')$, this can be done by means of a total deterministic bottom-up relabeling.
2. Each node v additionally receives its child number together with the set

$$T(v) = \{r(v) \mid r \text{ accepting run for } f\}$$

as label. Thus, now internal nodes v with original label a should have labels $(a, j, T(v), S(v1), S(v2))$ whereas leaf nodes v receive the labels $(\epsilon, j, T(v))$. Here, $j = 0$ if $v = \epsilon$, i.e., if v is the root of f. Otherwise, $v = v'j$ for some v', i.e., j equals the last direction in v. Since the resulting trees use a richer alphabet of leaf labels (not just "ϵ") than forests, we write the resulting terms as binary trees again. The second relabeling This relabeling can be implemented by means of a total deterministic top-down relabeling. $\mathit{Top} = (B, A, \Sigma', b_{0,F}, R_{\mathit{Top}})$ where $B = \{0,1,2\} \times 2^S$ with the rules:

$$b_{j,T_0}((a, S_1, S_2)(x_1, x_2)) \;\rightarrow\; (a, j, T, S_1, S_2)(b_{1,T_1}(x_1), b_{2,T_2}(x_2))$$

where $T = \{s \in T_0 \mid (a, s, s_1, s_2) \in \delta, s_i \in S_i\}$ and $T_i = \{s_i \in S_i \mid (a, s, s_1, s_2) \in \delta, s \in T, s_{3-i} \in S_{3-i}\}$, and

$$b_{j,T_0}(\epsilon) \;\rightarrow\; (\epsilon, j, T)$$

where $T = \{s \in T_0 \mid (\epsilon, s) \in \delta\}$. Note, that our bottom-up relabeling followed by a top-down relabeling can jointly be considered as top-down relabeling with regular look-ahead, which is semantically equivalent to a MSO definable relabeling [12, 7].

3. We construct from the TL transformer $\mathcal{T}$ a 2-mtt $M = (Q', \Sigma', \Delta, R')$ where $\Sigma' \subseteq \Sigma \times \{0,1,2\} \times 2^S \times 2^S \times 2^S$ as above and $Q' = Q \cup \text{Down} \cup \text{Up}$. Besides the states of the TL transformer, we introduce sets Down and Up of auxiliary states (procedures) which implement the matching of the TL transformer's navigation patterns.
For each TL rule

$$q(\phi, y_1, \ldots, y_k) \Rightarrow A$$

there will be rules

$$\begin{aligned} (a, j, T, S_1, S_2) : q(\pi, y_1, \ldots, y_k) &\rightarrow [A]_j \\ (\epsilon, j, T) : q(\pi, y_1, \ldots, y_k) &\rightarrow [A]_j \end{aligned}$$

in R' whenever $U_\phi \cap T \neq \emptyset$ where the new right-hand sides are determined by means of the transformation $[\,.\,]_j,\ j \in \{0,1,2\}$, which is recursively defined by:

$$\begin{aligned}
[\epsilon]_j &= \epsilon \\
[y_i]_j &= y_i \\
[a\langle A_1\rangle\ A_2]_j &= a([A_1]_j, [A_2]_j) \\
[A_1\ A_2]_j &= [A_1]_j \;@\; [A_2]_j \\
[q'(\psi, A_1, \ldots, A_k)]_0 &= \mathsf{down}^{q'}_{\psi,h}(\pi, [A_1]_0, \ldots, [A_k]_0) \\
[q'(\psi, A_1, \ldots, A_k)]_j &= \mathsf{up}^{j,q'}_{\psi,h}(\pi^-, [A_1]_j, \ldots, [A_k]_j, \\
&\qquad \mathsf{down}^{q'}_{\psi,h}(\pi, [A_1]_j, \ldots, [A_k]_j)) \quad \text{for } j \in \{1,2\}
\end{aligned}$$

where the relation $h \subseteq S^2$ is defined by:

$$h = \{(s, s) \mid s \in T\}.$$

In general, we have found a match iff $h \cap B_\psi \neq \emptyset$. Here a state $\mathsf{down}^{q'}_{\psi,h}$ is meant to traverse the subtree at the current node v and return (a representation of) the list of results obtained by applying the state q' to all nodes $v_1, \ldots, v_k$ in document order which complete the binary match. A state $\mathsf{up}^{j,q'}_{\psi,h}$

is meant to visit the ancestor of the current node v and to traverse from there the left or right sibling of v — depending on the node at which the up state has been called.

$$\begin{aligned}
(a,0,T,S_1,S_2) : {} & \mathsf{up}^{1,q'}_{\psi,h}(\pi, y_1, \ldots, y_k, y_{k+1}) \to \\
& q'(\pi, y_1, \ldots, y_k) \mathbin{@} y_{k+1} \mathbin{@} \mathsf{down}^{q'}_{\psi,g_1}(\pi 2, y_1, \ldots, y_k) \\
(a,j,T,S_1,S_2) : {} & \mathsf{up}^{1,q'}_{\psi,h}(\pi, y_1, \ldots, y_k, y_{k+1}) \to \mathsf{up}^{j,q'}_{\psi,h_1}(\pi^-, y_1, \ldots, y_k, \\
& q'(\pi, y_1, \ldots, y_k) \mathbin{@} y_{k+1} \mathbin{@} \mathsf{down}^{q'}_{\psi,g_1}(\pi 2, y_1, \ldots, y_k)) \\
(a,0,T,S_1,S_2) : {} & \mathsf{up}^{2,q'}_{\psi,h}(\pi, y_1, \ldots, y_k, y_{k+1}) \to \\
& q'(\pi, y_1, \ldots, y_k) \mathbin{@} \mathsf{down}^{q'}_{\psi,g_2}(\pi 1, y_1, \ldots, y_k) \mathbin{@} y_{k+1} \\
(a,j,T,S_1,S_2) : {} & \mathsf{up}^{2,q'}_{\psi,h}(\pi, y_1, \ldots, y_k, y_{k+1}) \to \mathsf{up}^{j,q'}_{\psi,h_2}(\pi^-, y_1, \ldots, y_k, \\
& q'(\pi, y_1, \ldots, y_k) \mathbin{@} \mathsf{down}^{q'}_{\psi,g_2}(\pi 1, y_1, \ldots, y_k) \mathbin{@} y_{k+1})
\end{aligned}$$

given that $h_1 \cap B_\psi \neq \emptyset$ for $\mathsf{up}^{1,q'}_{\psi,h}$ or $h_2 \cap B_\psi \neq \emptyset$ for $\mathsf{up}^{2,q'}_{\psi,h}$, respectively. Here,

$$\begin{aligned}
h_i &= \{(s,s') \in S \times T \mid \exists\, (s,s_i) \in h, s_{3-i} \in S_{3-i} \ : \ (a,s',s_1,s_2) \in \delta\} \\
g_i &= \{(s,s_{3-i}) \in S \times S_{3-i} \mid \exists\, s' \in T, (s,s_i) \in h \ : \ (a,s',s_1,s_2) \in \delta\}
\end{aligned}$$

Assume that the formal location π refers to the node v' in the input. Assume further that some node v (where the currently simulated TL-rule is to be applied) is a descendent of the i-th child v_i of v'. Then h_i computes the relation between the states at v and the states at v' — given the corresponding relation h between the states at v and the states at v_i. In particular, (v,v') is a match of ψ iff $h_i \cap B_\psi \neq \emptyset$. Likewise depending on h, g_i computes the relation between the states at v and the states at the sibling v_{3-i} of v_i.

If $h_i \cap B_\psi = \emptyset$, i.e., the pair (v,v') is not a match of ψ, we have:

$$\begin{aligned}
(a,0,T,S_1,S_2) : {} & \mathsf{up}^{1,q'}_{\psi,h}(\pi, y_1, \ldots, y_k, y_{k+1}) \to \\
& y_{k+1} \mathbin{@} \mathsf{down}^{q'}_{\psi,g_1}(\pi 2, y_1, \ldots, y_k) \\
(a,j,T,S_1,S_2) : {} & \mathsf{up}^{1,q'}_{\psi,h}(\pi, y_1, \ldots, y_k, y_{k+1}) \to \mathsf{up}^{j,q'}_{\psi,h_1}(\pi^-, y_1, \ldots, y_k, \\
& y_{k+1} \mathbin{@} \mathsf{down}^{q'}_{\psi,g_1}(\pi 2, y_1, \ldots, y_k)) \\
(a,0,T,S_1,S_2) : {} & \mathsf{up}^{2,q'}_{\psi,h}(\pi, y_1, \ldots, y_k, y_{k+1}) \to \\
& \mathsf{down}^{q'}_{\psi,g_2}(\pi 1, y_1, \ldots, y_k) \mathbin{@} y_{k+1} \\
(a,j,T,S_1,S_2) : {} & \mathsf{up}^{2,q'}_{\psi,h}(\pi, y_1, \ldots, y_k, y_{k+1}) \to \mathsf{up}^{j,q'}_{\psi,h_2}(\pi^-, y_1, \ldots, y_k, \\
& \mathsf{down}^{q'}_{\psi,g_2}(\pi 1, y_1, \ldots, y_k) \mathbin{@} y_{k+1})
\end{aligned}$$

If $h_i \cap B_\psi = \emptyset$, i.e., the pair (v,v') is not a match of ψ, we have the same rule as above with the only difference that $q'(\pi, y_1, \ldots, y_k)$ is not concatenatet to the result.

The down transitions concatenate the matches (transformed with q') to the result and the transformation continues with the children:

$$\begin{array}{rl}(\epsilon, j, T) : \mathsf{down}^{q'}_{\psi,h}(\pi, y_1, \dots, y_k) \to & q'(\pi, y_1, \dots, y_k) \\ (a, j, T, S_1, S_2) : \mathsf{down}^{q'}_{\psi,h}(\pi, y_1, \dots, y_k) \to & q'(\pi, y_1, \dots, y_k) \mathbin{@} \\ & \mathsf{down}^{q'}_{\psi,g'_1}(\pi 1, y_1, \dots, y_k) \mathbin{@} \\ & \mathsf{down}^{q'}_{\psi,g'_2}(\pi 2, y_1, \dots, y_k)\end{array}$$

if the current node is a match, i.e. $h \cap B_\psi \neq \emptyset$. Here,

$$g'_i = \{(s, s_i) \in S \times S_i \mid \exists\, (s, s') \in h, s_{3-i} \in S_{3-i} \;:\; (a, s', s_1, s_2) \in \delta\}$$

Thus, assume that the node v is no proper descendent of v' where h describes a relation between the states at v and the states at v'. Then g'_i describes the corresponding relation between the states at v and the states at the i-th child of v'.
If no match is found, i.e., $h \cap B_\psi = \emptyset$, the transformation continues with the children:

$$\begin{array}{rl}(\epsilon, j, T) : \mathsf{down}^{q'}_{\psi,h}(\pi, y_1, \dots, y_k) \to & \epsilon \\ (a, j, T, S_1, S_2) : \mathrm{down}^{q'}_{\varphi,h}(\pi, y_1, \dots, y_k) \to & \mathsf{down}^{q'}_{\psi,g'_1}(\pi 1, y_1, \dots, y_k) \mathbin{@} \\ & \mathsf{down}^{q'}_{\psi,g'_2}(\pi 2, y_1, \dots, y_k)\end{array}$$

Correctness:
Let $t \in \mathcal{T}_{\Sigma'}$ be an arbitrary but fixed input tree. For two nodes v_1 and v_2 of t labeled with $t(v_i) = (\_, \_, T_i, \_, \_)$ for $i = 1, 2$, the relation $g_t(v_1, v_2) \subseteq T_1 \times T_2$ relates states in T_1 at v_1 and T_2 at v_2 which are possibly connected through a common run r on t of the match automaton $\mathcal{P}$. This relation is defined by

$$(s_1, s_2) \in g_t(v_1, v_2) \quad \text{iff} \quad \exists \text{ accepting run } r : s_1 = r(v_1) \text{ and } s_2 = r(v_2)$$

Claim 1:

$$\textsc{app}([\![\mathsf{down}^{q}_{\psi,h}]\!]_t(v, \underline{T})) = \textsc{app}([\![q]\!]_t(u_1, \underline{T})) \;\dots\; \textsc{app}([\![q]\!]_t(u_l, \underline{T}))$$

for all $t \in \mathcal{T}_{\Sigma'},\ v, u_1, \dots, u_l \in \mathcal{N}(t),\ q \in Q$, and a list $\underline{T} = T_1, \dots, T_k$ of actual parameters $T_i \in \mathcal{T}_\Delta$ where $u_1 < \ldots < u_l$ are exactly the nodes u' in document order occurring in the subtree t/v of t rooted at v such that

$$g_t(v, u') \circ h \;\cap\; B_\psi \neq \emptyset$$

The proof is by induction on the depth n of the subtree t/v. For $n = 0$, consider the node $u' = v$. Then $[\![\mathsf{down}^{q}_{\psi,h}]\!]_t(v, \underline{T})$ either equals $[\![q]\!]_t$ applied to v and $\underline{T}$ given that $h \cap B_\psi \neq \emptyset$, or equals ϵ otherwise. Since $g_t(v, u')$ is the identity relation, the assertion follows. If $n > 0$, v is not a leaf of t and therefore has two children $v_i = vi$, $i = 1, 2$. Let us first consider the case where $h \cap B_\phi = \emptyset$. Then, all descendents u' of v with $g_t(v,u')\circ h \;\cap\; B_\psi \neq \emptyset$ either are descendents of v_1 or of v_2. Accordingly, the sequence of nodes $u_1,\dots,u_l$ is the concatenation of two sequences $u_1,\dots,u_{l'}$ and $u_{l'+1},\dots,u_l$ where the nodes in the first sequence are descendents of v_1 and the others are descendents of v_2. By inspecting the definition of g'_i and g_t, we verify that $g_t(v,u')\circ h = g_t(v_i,u')\circ g'_i$ whenever u' is a descendent of v_i. Therefore by induction hypothesis,

$$\begin{array}{l}\textsc{app}([\![\mathsf{down}^q_{\psi,g'_1}]\!]_t(v_1,\underline{T})) = \textsc{app}([\![q]\!]_t(u_1,\underline{T}))\ \dots\ \textsc{app}([\![q]\!]_t(u_{l'},\underline{T})) \quad \text{and} \\ \textsc{app}([\![\mathsf{down}^q_{\psi,g'_2}]\!]_t(v_2,\underline{T})) = \textsc{app}([\![q]\!]_t(u_{l'+1},\underline{T}))\ \dots\ \textsc{app}([\![q]\!]_t(u_l,\underline{T}))\end{array}$$

Since by definition,

$$[\![\mathsf{down}^q_{\psi,h}]\!]_t(v,\underline{T}) = [\![\mathsf{down}^q_{\psi,g'_1}]\!]_t(v_1,\underline{T}) \ @\ [\![\mathsf{down}^q_{\psi,g'_2}]\!]_t(v_2,\underline{T})$$

the assertion of the claim follows. It remains to consider the case where $h\cap B_\phi \neq \emptyset$. Then $v = u_1$. For the remaining sequence $u_2,\dots,u_l$ we may argue as before that

$$\begin{array}{l}\textsc{app}([\![\mathsf{down}^q_{\psi,g'_1}]\!]_t(v_1,\underline{T}) \ @\ [\![\mathsf{down}^q_{\psi,g'_2}]\!]_t(v_2,\underline{T})) \\ \qquad\qquad\qquad\qquad = \textsc{app}([\![q]\!]_t(u_2,\underline{T}))\ \dots\ \textsc{app}([\![q]\!]_t(u_l,\underline{T}))\end{array}$$

Since now by definition,

$$[\![\mathsf{down}^q_{\psi,h}]\!]_t(v,\underline{T}) = [\![q]\!]_t(v,\underline{T}) \ @\ [\![\mathsf{down}^q_{\psi,g'_1}]\!]_t(v_1,\underline{T}) \ @\ [\![\mathsf{down}^q_{\psi,g'_2}]\!]_t(v_2,\underline{T})$$

the assertion of Claim 1 again follows.

Assume that the node v of t is a descendent of the node v_j which in turn is the j-th child of the node v'. Assume further that $h = g_t(v,v_j)$ and that the value of the parameter T_{k+1} satisfies:

$$\textsc{app}(T_{k+1}) = \textsc{app}([\![q]\!]_t(u_1,\underline{T}))\dots\textsc{app}([\![q]\!]_t(u_l,\underline{T}))$$

where $u_1 < \dots < u_l$ are the descendents u' of v_j with

$$g_t(v,u')\cap B_\psi \neq \emptyset$$

Thus the sequence $u_1 < \dots < u_l$ consists of all descendents u' of v_j in document order such that (v,u') is a match of ψ in t. Assume $t(v') = (a,i,T,S_1,S_2)$. Consider the call $[\![\mathsf{up}^{j,q}_{\psi,h}]\!]_t(v',\underline{T},T_{k+1})$. Now let h_j, g_j be defined according to the possible right-hand side of $\mathsf{up}^{j,q}_{\psi,h}$ and the label of $t(v')$. Moreover, define T'_{k+1} as the new value of the $k+1$-st accumulating parameter in the recursive call for the father of v' (in case $i>0$) or the complete right-hand side (in case $i=0$). Thus,

$$T'_{k+1} = \begin{cases} [\![q]\!]_t(v',\underline{T}) \ @\ T_{k+1} \ @\ [\![\mathsf{down}^q_{\psi,g_1}]\!]_t(v_2,\underline{T}) & \text{if} \quad j=1 \ \wedge\ h_1\cap B_\psi \neq \emptyset \\ T_{k+1} \ @\ [\![\mathsf{down}^q_{\psi,g_1}]\!]_t(v_2,\underline{T}) & \text{if} \quad j=1 \ \wedge\ h_1\cap B_\psi = \emptyset \\ [\![q]\!]_t(v',\underline{T}) \ @\ T_{k+1} \ @\ [\![\mathsf{down}^q_{\psi,g_2}]\!]_t(v_1,\underline{T}) & \text{if} \quad j=2 \ \wedge\ h_2\cap B_\psi \neq \emptyset \\ T_{k+1} \ @\ [\![\mathsf{down}^q_{\psi,g_2}]\!]_t(v_1,\underline{T}) & \text{if} \quad j=2 \ \wedge\ h_2\cap B_\psi = \emptyset \end{cases}$$

We claim:

Claim 2:

1. $h_j = g_t(v, v')$ and $g_j = g_t(v, v_{3-j})$ for $j = 1, 2$.
2. T'_{k+1} satisfies the same property as T_{k+1} – but now for the whole subtree rooted at v', i.e.,

$$\textsc{app}(T'_{k+1}) = \textsc{app}([\![q]\!]_t(u_1, \underline{T})) \ldots \textsc{app}([\![q]\!]_t(u_m, \underline{T}))$$

where $u_1 < \ldots < u_m$ are the descendents u' of v' with

$$g_t(v, u') \cap B_\psi \neq \emptyset$$

The first item of the claim follows from the property of h by inspection of the definitions of the h_j and g_j. For the second item, we recall from Claim 1 that the call $[\![\mathsf{down}^q_{\psi,g_j}]\!]_t(v_{3-j}, \underline{T})$ returns (the representation of) the list of all calls $[\![q]\!]_t(u', \underline{T})$ where $g_t(v_{3-j}, u') \circ g_j \cap B_\psi \neq \emptyset$. From the definitions of g_j and g_t, we verify that

$$g_t(v_{3-j}, u') \circ g_j = g_t(v, u')$$

We conclude that the sequence of u' in the subtree rooted at v_{3-j} precisely equals the sequence of all $v' \leq v_{3-j}$ such that (v, u') is a match of ψ. This completes the proof of Claim 2.

The correctness proof for our translation now proceeds by fixpoint induction. Within the induction step, we have to prove by structural induction on the right-hand sides in TL-rules that:

$$[\![A]\!]_t(v, \underline{T}) = \textsc{app}([\![[A]_j]\!](v, \underline{T}))$$

The only difficult case is the simulation of calls $q'(\psi, \underline{A})$. For that case, we iteratively may apply Claim 2 to the ancestors of the node v. Thus, we deduce for $h = \{(s, s) \mid s \in T\}$ that the calls: $[\![\mathsf{down}^{q'}_{\psi,h}]\!]_t(v, \underline{T})$ (if v is the root of t) or $[\![\mathsf{up}^{j,q'}_{\psi,h}]\!]_t(v^-, \underline{T}, [\![\mathsf{down}^{q'}_{\psi,h}]\!]_t(v, \underline{T}))$ (if v^- is the father of v) result in a representation of

$$[\![q]\!]_t(u_1, \underline{T}) \ldots [\![q]\!]_t(u_n, \underline{T})$$

where $u_1 < \ldots < u_n$ is the sequence of nodes u' such that (v, u') are precisely all matches of ψ in t, i.e., $g_t(v, u') \cap B_\psi \neq \emptyset$ — which we needed to verify. If no match is found, i.e., $h \cap B_\psi = \emptyset$, the down transition only continues with the children. Thus, the rules of the first form return ϵ and that of the second form compute $\mathsf{down}^{q'}_{\psi,g'_1}(\pi 1, y_1, \ldots, y_k)$ @ $\mathsf{down}^{q'}_{\psi,g'_2}(\pi 2, y_1, \ldots, y_k)$. $\diamond$

We obtain our main result, namely, that every TL transformation can be realized by the composition of one stay mtt and two fixed deterministic mtt's that only depend on the involved ranked alphabets.

Theorem 9 $\textsc{tl} \subseteq \textsc{app} \circ \textsc{yield}^\bullet \circ \textsc{s-mtt}$.

Proof. By Lemma 8, $\text{TL} \subseteq \text{APP} \circ 2\text{-MTT} \circ \text{T-REL} \circ \text{B-REL}$. We now apply Lemma 5 and replace the 2-MTT by $\text{YIELD}^\bullet \circ \text{s-MTT}$. Further, we can apply Lemma 3 in order to get rid of the stay mtt. Altogether we obtain that TL is included in $\text{APP} \circ \text{YIELD}^\bullet \circ \text{MTT} \circ \text{MON} \circ \text{T-REL} \circ \text{B-REL}$. We observe that MON can be moved through the relabelings onto the beginning of this composition. First, it should be clear that $\text{MON} \circ \text{T-REL} \subseteq \text{T-REL} \circ \text{MON}$: For every rule $a : q(\pi) \to b(q_1(\pi 1), \ldots, q_k(\pi k))$ of the top-down relabeling, a new rule $\bar{a} : q(\pi) \to \bar{a}(q(\pi 1))$ is added. In this way the new transducer can take as input a tree that was processed by MON, and will relabel the unary symbols $\bar{a}$ by the correct symbol. Similarly it can be shown that $\text{MON} \circ \text{B-REL} \subseteq \text{B-REL} \circ \text{MON}$. Given a B-REL we have to add for every output symbol δ and every state q, a new state q_δ. Then, each right-hand side of the B-REL is changed from $q(\delta(\ldots))$ into $q_\delta(\delta(\ldots))$. Thus, we remember in the state the most recently relabeled symbol. Now if a barred symbol is encountered in state q_δ then it is relabeled by $\bar{\delta}$. The application of the two mentioned inclusions gives us $\text{TL} \subseteq \text{APP} \circ \text{YIELD}^\bullet \circ \text{MTT} \circ \text{T-REL} \circ \text{B-REL} \circ \text{MON}$. It follows from Theorem 6.14 and Corollary 4.10 of [14] that $\text{MTT} \circ \text{D}_t\text{T-REL} \subseteq \text{MTT}$. Similarly, from Theorem 6.14 and Theorem 4.12 of [14] it follows that $\text{MTT} \circ \text{D}_t\text{B-REL} \subseteq \text{MTT}$. Hence, we obtain that $\text{TL} \subseteq \text{APP} \circ \text{YIELD}^\bullet \circ \text{MTT} \circ \text{MON}$. For the inclusion: $\text{MTT} \circ \text{MON} \subseteq \text{s-MTT}$, it suffices to change every rule $\bar{\sigma} : l \to r$ of Lemma 3 into $\sigma : l \to r'$, where r' is obtained from r by changing $\pi 1$ into π. Replacing $\text{MTT} \circ \text{MON}$ by s-MTT, gives $\text{TL} \subseteq \text{APP} \circ \text{YIELD}^\bullet \circ \text{s-MTT}$. $\diamond$

Recall that APP can be realized by a total deterministic mtt, and that $\text{YIELD}^\bullet$ can be realized by a deterministic mtt. Since $\text{s-MTT} \subseteq \text{MTT} \circ \text{MON}$, we obtain from Theorem 9 the following corollary.

Corollary 10 $\text{TL} \subseteq \text{D}_t\text{MTT} \circ \text{DMTT} \circ \text{MTT} \circ \text{MON}$.

4 Type Checking TL Transformers

Recall from the introduction that, in the context of XML transformations, a type is a regular forest language. It is a known and obvious fact that if we view the forests of a regular forest language as binary trees, then we obtain a (binary) regular tree language (and vice versa; see, e.g., [25]). Denote the class of all (binary) regular tree languages by REGT. It is the class of tree languages accepted by the tree automata defined in Section 3 (above Lemma 6).

Given a transformer T, an input type $I \in \text{REGT}$, and an output type $O \in \text{REGT}$, we say that T *type checks with respect to I and O* if $\tau_T(I) \subseteq O$. One way of solving the type checking problem is to use inverse type inference: given T and O, *inverse type inference* determines the tree language $P = \tau_T^{-1}(\overline{O}) = \{t \mid \tau_T(t) \cap \overline{O} \neq \emptyset\}$ where $\overline{O}$ denotes the complement of O. Clearly, T type checks w.r.t. I and O iff $I \cap P = \emptyset$. We now show that Corollary 10 implies that inverse type inference is possible for TL transformers. We first recall:

Lemma 11 If O is a regular tree language, then so is $\text{MON}^{-1}(O)$.

Proof. Intuitively, the inverse of a translation in MON deletes all barred monadic symbols of the form $\bar{\sigma}$. This can be realized by a top-down transducer: For every non-barred input symbol σ of rank k it has a rule $\sigma : q_0(\pi) \to \sigma(q_0(\pi 1), ..., q_0(\pi k))$ which simply copies the node, and for every barred symbol $\bar{\sigma}$ (of rank 1) it has a rule $\sigma : q_0(\pi) \to q_0(\pi)$ which deletes the node. Note that any πj, $j \geq 1$ occurs at most once in the right-hand side of each rule. Such a transducer is called *linear* and it is well-known that linear top-down tree transducers preserve the regular tree languages (cf., e.g., Proposition 20.2 of [19]). $\diamond$

By Theorem 7.4 of [14], the inverse image of an mtt (and thus also of compositions of mtt's) applied to a regular tree language is again regular. Therefore, by Corollary 10 and Lemma 11 we obtain:

Theorem 12 If τ is a composition of TL transformations and O is a regular tree language, then $\tau^{-1}(O)$ is a regular tree language.

With O the complement $\overline{O}$ is regular and therefore, by Theorem 12, also $P = \tau^{-1}(\overline{O})$ for every TL transformation τ. Since regular tree languages are closed under intersection and have decidable emptiness, we conclude:

Theorem 13 Type checking for (compositions of) TL transformers is decidable.

Complexity What is the complexity of the type checking problem for TL transformers? Recall from Corollary 10 that

$$\text{TL} \subseteq \text{D}_t\text{MTT} \circ \text{DMTT} \circ \text{MTT} \circ \text{MON}.$$

Now, how efficiently can a regular tree language O be inverse translated by a composition on the right? In fact, let us consider only the complexity of inverse type inference for a mtt composition on the right of the above inclusion. The combined complexity for the type checking problem can then be obtained by first constructing a mtt composition from the TL transformer, as shown in this paper, and then applying the standard algorithms on regular tree languages.

Let B be the minimal tree automaton accepting O. We define the size $|B|$ of an automaton B to be the number of its states. It should be clear that a tree automaton for $O' = \text{MON}^{-1}(\overline{O})$ can be constructed in time linear in $|B|$ (cf. Lemma 11), and hence also the size of an automaton for O' is linear in $|B|$.

For simplicity, let us first determine how the size of a tree automaton B (with set of states P) changes when being inverse-translated by a *deterministic* mtt $M = (Q, \Sigma, \Delta, q_0, R)$. A direct construction for this task is given in Section 7 of [13]. The (input) automaton A constructed there has as states mappings d such that for every $q \in Q^{n+1}$ and $n \geq 0$, $d(q)$ is a mapping from P^n to P. Thus, the number of states of A is $|Q \to P^n \to P|$ which equals $(|P|^{|P|^n})^{|Q|} = |P|^{|Q||P|^n}$. Let the size m of the mtt M be defined in such a way that $|Q| \leq m$ and $n \leq m$. Then, taking b as the number of states of B ($= |P|$) we can approximate $|A|$ by

$b^{mb^m} = 2^{2^{\mathcal{O}(m\ \log(b))}}$. For the composition of two deterministic mtt's M and $\bar{M}$ (with maximal size m) we obtain as size of the corresponding input automaton

$$2^{2^{2^{\mathcal{O}(m\ \log(b))}}}$$

It remains to consider the nondeterministic mtt that occurs in the composition above. The inverse type inference construction is similar to the one given in Section 7 of [13]. Taking into account the inductive characterization of OI mtt's, given in Definition 3.16 of [14], we obtain an automaton with set of states $Q \to (2^P)^m \to 2^P$ which has cardinality $\kappa = 2^{bm2^{bm}} = 2^{2^{\mathcal{O}(b \cdot m)}}$. We are now ready to consider the full composition of the three mtt's above, and to determine an approximation of the size of the input automaton. Let m be the maximum of the sizes of the three mtt's, and b is the size of the automaton for O. We simply have to plug κ into the above formula for two mtt's.

$$\text{Size of tree automaton for } \tau^{-1}(\overline{O}) \quad \approx \quad 2^{2^{2^{2^{\mathcal{O}(b \cdot m)}}}}.$$

From a practical point of view, these numbers do not suggest efficient type checking. The problem is however difficult, and we should compare our result to the complexity of type checking k-pebble tree transducer. In Theorem 4.8 of [24] it is shown that if T is a k-pebble tree transducer (of size n), then even for fixed types I and O, the type checking problem for arbitrary k is non-elementary. Hence, using MSO to describe patterns, in place of pebbles, and describing MSO patterns by some less expensive formalism like grammars or *fxt* patterns, brings the complexity from non-elementary down to a tower of exponents of bounded height. It remains a challenging engineering problem whether a practical type checking algorithm can be designed and implemented along the lines we have outlined here.

5 Deterministic TL Transformers

Since deterministic XML transformations are of great practical importance, we investigate in this section deterministic variants of our transformation language TL.

Intuitively, determinism means that at each moment of the computation and for each function call there shall only be at most one rule that is applicable. However, since the applicability of a rule of a TL transformer is determined by the match pattern in the rule's left-hand side, we should first require that all match patterns for a particular input symbol and state are pairwise disjoint. The resulting definition of determinism for TL becomes quite technical; it is considered at the end of this section. Now we choose a different approach.

Clearly, one of the aims of a definition of determinism is to force the corresponding translations to be *functions*. Similarly, for total determinism we want to obtain *total* functions. This is precisely our definition of this notion for TL.

Definition 14 A TL transformer T is *total deterministic*, if τ_T is a total function. The class of translations realized by total deterministic TL's is denoted $\mathrm{D}_t\mathrm{TL}$.

Clearly, such a "semantic" definition can be problematic. For instance, given a TL transformer T, it might not even be decidable whether or not τ_T is a total function. In the sequel of this section we show that this property suffices to give a characterization in terms of a composition of total deterministic macro tree transducers.

Recall from Corollary 10 that $\mathrm{TL} \subseteq \mathrm{D}_t\mathrm{MTT} \circ \mathrm{DMTT} \circ \mathrm{MTT} \circ \mathrm{MON}$. As mentioned in Section 2 (in the part about stay moves), any translation in MON can be realized by an s-top. By Theorem 7.6 of [14] (there MTT is denoted by $\mathrm{MTT}_{\mathrm{OI}}$), $\mathrm{MTT} = \mathrm{TOP} \circ \mathrm{D}_t\mathrm{MTT}$ and $\mathrm{DMTT} = \mathrm{DTOP} \circ \mathrm{D}_t\mathrm{MTT}$. Thus, TL can be decomposed into

$$\mathrm{D}_t\mathrm{MTT} \circ \mathrm{DTOP} \circ \mathrm{D}_t\mathrm{MTT} \circ \mathrm{TOP} \circ \mathrm{D}_t\mathrm{MTT} \circ \text{S-TOP}.$$

Our goal is to restrict the above class of translations to total functions, and to obtain a composition of total deterministic transducers only. Since all the mtt's above are total deterministic already, we can restrict our attention to the involved top-down tree transducers. The main idea is to show that it is possible to generate a "deterministic cut" of any (stay) top-down tree transducer. For tops this was shown in the lemma of [11]; we now generalize this result to s-tops. An s-top with regular look-ahead is an s-top composed with a bottom-up relabeling.

Lemma 15 For each s-top T there exists a deterministic s-top T' with regular look-ahead such that $\tau_{T'} \subseteq \tau_T$ and $\mathrm{dom}(T') = \mathrm{dom}(T)$.

Proof. For a state p of T denote by T_p the s-top obtained from T by making p the initial state. The basic fact needed for the proof is that the domain of T_p is a regular tree language. Thus, we need to show that the domain of any s-top M is regular. Clearly, $\mathrm{dom}(M) = \tau_M^{-1}(\mathcal{T}_\Delta)$, where Δ is the output alphabet of M. By Lemma 3, $\text{S-TOP} \subseteq \mathrm{TOP} \circ \mathrm{MON}$. It is well known that inverses of top-down tree transducers preserve the regular tree languages (see, e.g., Theorem 7.4 of [14]). Since also MON^{-1} preserves the regular tree languages, by Lemma 11, we obtain that $\mathrm{dom}(M)$ is regular.

The remainder of the proof goes along the lines of the proof of the Lemma in [11]. Let q be a state of T, σ an input symbol of rank n, and $r_1, \dots, r_k$ the right-hand sides of all rules for q and σ. For every $1 \leq j \leq k$ and $1 \leq i \leq n$ define

$$D_j(i) = \bigcap\{\mathrm{dom}(T_p)) \mid p(\pi i) \text{ occurs in } r_j\} \\ \cap \bigcap\{\mathrm{dom}(T_p)/i \mid p(\pi) \text{ occurs in } r_j\}.$$

Clearly, if $t_i \in D_j(i)$ for $1 \leq i \leq n$, then the rule r_j can be applied at the root of $\sigma(t_1, \dots, t_n)$, and the computation can finish with a terminal tree. Since the sets $D_j(i)$ are regular, we can use a bottom-up relabeling in order to add to each input node a mapping d which assigns to each i the set of all j such that the i-th subtree belongs to the set $D_j(i)$. The rule of T' for q and (σ, d) is r_j if there is a j such that for all $1 \leq i \leq n$, $j \in d(i)$ (i.e., the i-th input subtree is in $D_j(i)$), and is undefined otherwise. It should be clear that T' has the same domain as T', and that $\tau_{T'} \subseteq \tau_T$. $\diamond$

Thus, we characterize the total deterministic TL transformers in terms of total deterministic mtt's.

Theorem 16 $\text{D}_t\text{TL} \subseteq \text{D}_t\text{MTT}^3$.

Proof. As mentioned above, every TL transformation can be decomposed into

$$\tau = \tau_{M_3} \circ \tau_{T_3} \circ \tau_{M_2} \circ \tau_{T_2} \circ \tau_{M_1} \circ \tau_{T_1}.$$

where T_1 and T_2 are nondeterministic top's (with and without stay moves, respectively), and all other transducers are deterministic. First, we can change each T_i, $1 \leq i \leq 3$ in such a way that it only generates output trees which are in the domain of the remaining composition. To see this, note first that these domains are regular tree languages (by Theorem 7.4 of [14] and because the domain of a s-top is regular as shown in the proof of Lemma 15). Restricting the output of an s-top T to a regular tree language R can be realized by a straightforward product construction: Let A be a (nondeterministic) top-down tree automaton accepting R. Let $\sigma : q(\pi) \to r$ be a rule of T and let p be a state of A. Then $\langle q, p\rangle$ is a state of the new s-top T'. To obtain a rule of T' we try to run A, starting in state p, on the tree r. If A arrives in state p' at some $q'(\pi i)$, then q' is replaced by $\langle q', p'\rangle$. For each possible run of A on r we add a corresponding rule to T'. Clearly, $\tau_{T'} = \{(s,t) \in \tau_T \mid t \in R\}$. Note that for top-down tree transducers the result already follows e.g., from the proof of Lemma 2.10(1) in [11], taking the trivial look-ahead.

In the new composition (with T_1, T_2 replaced by T_1', T_2', respectively), each intermediate tree is translated by the remainder to a final output tree. This implies that if a computation "splits", i.e., due to nondeterminism there are more than one output tree for a particular input tree, then all these trees must be translated by the remainder of the composition into the *same* output tree. We now apply Lemma 15 to T_1', T_2'. This gives us the deterministic s-top and top T_1'', T_2'' which generate one particular output tree of the original transducer, whenever there is a choice. By the above this implies that the translation has remained the same and hence τ is now represented by deterministic transducers only. It remains to remove the stay moves of T_1''. This can be done by taking the parameterless case of the proof of Theorem 31 of [13]. Treating partiality of the transducers is now trivial: since τ is a total function we know that no 'missing rule' ever becomes applicable. Thus in order to obtain total transducers we can simply add a dummy rule for each missing rule. Since D_tMTT is closed under composition with D_tTOP by [14] we obtain the desired result. $\diamond$

To show that it is decidable for a TL transformer whether or not it is total deterministic, is left for further research. Independent of this result, however, Theorem 16 can be used already, because it is straightforward to find (decidable) restrictions on TL transformers which guarantee that the translations are total functions.

As an example, consider the restriction that at each step of a successful computation of a TL transformer T there is exactly one rule applicable to each configuration $q(u)$. Clearly this implies that τ_T is a function. This restriction can be decided as follows. For a state q, let $\phi_1, \ldots, \phi_k$ all MSO formulas that appear in left-hand sides. Make a new transducer T' as follows. Let ψ be a conjunction of all ϕ_i (positive or negated). The root node of the right-hand side is labeled by a unary symbol I which is a set containing the indices of the ϕ_i which occur positive in ψ. Below the I we add a tree over some new binary output symbol b which has at its leaves all state calls that appear in right-hand side of the positive ϕ_i's of ψ. Now, let R be the regular tree language over b and sets I such that there is at least one I with cardinality > 1. By Theorem 12, $I = \tau_T^{-1}(R)$ is regular. If I is empty, then M is deterministic. This can be decided, because emptiness of regular tree languages is decidable.

6 Related Work

The most used and well-known transformation languages are those released or being in preparation by W3C, aiming at standardizing the way that XML documents are manipulated. XPATH 1.0 [9] and 2.0 [3] are languages for addressing parts of an XML document. An XPATH expression specifies how to locate a set of nodes from the input document. Basically, this is done by successively selecting nodes by means of patterns depending on one another. XSLT [10, 21] is a *rule-based* language for transforming XML documents. The transformation is specified as a set of rules each of which consists of a *match pattern* and a corresponding *action*. While the match pattern specifies the subtrees to which the rule is applicable, the action part defines the transformation result as XML fragment consisting of statically specified content, parts of the subtree being transformed and by recursively applying the transformation rules on subtrees matching the *select patterns*. XSLT uses XPATH expressions both for match and select patterns. Like XSLT, *fxt* [6] is a rule-based transformation language which, however, uses *fxgrep* [27] as pattern language. The syntax of *fxgrep* is similar to that of XPATH but based on tree automata. Selection of nodes for further processing is by means of binary patterns [5]. Neither XSLT nor *fxt* provide support for type checking.

Opposed to that, XQUERY [8] is a strongly-typed functional language for querying XML documents which is again based on XPATH. Type checking is performed via *forward* type inference rules and thus possibly leads to dynamic type errors or may fail to type check correct programs. Type inference is also used in the XDUCE [20] and CDUCE [2, 17] family of functional languages. These languages extend the pattern matching mechanism of functional languages by regular expression constructs. Patterns may contain variables which are bound to parts of the matched structure. The bound forests then can be addressed in subsequent processing. Another functional approach to XML processing is XMλ [23]. Here, the type information is mapped onto (extended) Haskell types. Like XQUERY and XDUCE/CDUCE, type inference is approximate.

Besides in [25, 13], *inverse* type inference has also been used by Tozawa for a subset of XSLT which essentially consists of top-down tree transformations [32].

7 Conclusion

We have introduced the small but very expressive tree transformation language TL which subsumes the tree transformation core of most existing XML transformation languages. We have shown that every TL transformation can be effectively decomposed into three (stay) macro tree transducers, independent of the complexity of the transformation or the used patterns. This improves, e.g., on a similar simulation of k-pebble tree transducers where the number of mtt's in the composition is linear in k. Applying the known result that regularity is preserved by inverse images of (stay) mtt's, we obtain an elegant procedure for type checking all transformers expressible in TL. It remains a challenging engineering problem whether this decomposition gives rise to an implementation which is sufficiently efficient on real world transformations.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kaufmann, 2000.
2. V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 51–63. ACM Press, 2003.
3. A. Berglund and S. Boag et.al., editors. XML Path Language (XPath) 2.0. W3C Working Draft, World Wide Web Consortium, November 2003. Available online http://www.w3.org/TR/xpath20.
4. A. Berlea and H. Seidl. fxt - A Transformation Language for XML Documents. In *XML Conference And Exposition 2001, Orlando, USA*, December 2001.
5. A. Berlea and H. Seidl. Binary queries. In *Proc. Extreme Markup Languages 2001, Montreal, Canada*, August 2002.
6. A. Berlea and H. Seidl. fxt – A Transformation Language for XML Documents. *Journal of Computing and Information Technology*, 10(1):19–35, 2002.
7. R. Bloem and J. Engelfriet. Monadic Second Order Logic and Node Relations on Graphs and Trees. In *Structures in Logic and Computer Science*, pages 144–161. LNCS 1261, Springer-Verlag, 1997.
8. S. Boag and D. Chamberlin et.al., editors. XQuery 1.0: An XML Query Language. W3C Working Draft, World Wide Web Consortium, November 2003. Available online http://www.w3.org/TR/xquery/.
9. J. Clark and S. DeRose, editors. XML Path Language (XPath) 1.0. W3C Recommendation, World Wide Web Consortium, November 1999. Available online http://www.w3.org/TR/xpath.

10. J. Clark, editor. XSL Transformations (XSLT) 1.0. W3C Recommendation, World Wide Web Consortium, November 1999. Available online http://www.w3.org/TR/xslt.
11. J. Engelfriet. Top-down Tree Transducers with Regular Look-Ahead. *Math. Systems Theory*, 10:289–303, 1977.
12. J. Engelfriet and S. Maneth. Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. *Inform. and Comput.*, 154:34–91, 1999.
13. J. Engelfriet and S. Maneth. A Comparison of Pebble Tree Transducers with Macro Tree Transducers. *Acta Inf.*, 39:613–698, 2003.
14. J. Engelfriet and H. Vogler. Macro Tree Transducers. *J. of Comp. Syst. Sci.*, 31:71–146, 1985.
15. J. Engelfriet and H. Vogler. Pushdown Machines for the Macro Tree Transducer. *Theoret. Comp. Sci.*, 42:251–368, 1986.
16. D.C. Fallside, editor. XML Schema. W3C Recommendation, World Wide Web Consortium, 2 May 2001. Available online http://www.w3.org/TR/xmlschema-0/.
17. A. Frisch. Regular Tree Language Recognition with Static Information, 2004. PLAN-X 2004.
18. Z. Fülöp and H. Vogler. *Syntax-Directed Semantics; Formal Models Based on Tree Transducers*. Springer-Verlag, 1998.
19. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3*, chapter 1. Springer-Verlag, 1997.
20. H. Hosoya and B.C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.
21. M. Kay, editor. XSL Transformations (XSLT) 2.0. W3C Working Draft, World Wide Web Consortium, November 2003. Available online http://www.w3.org/TR/xslt20.
22. A. Kühnemann and H. Vogler. Synthesized and Inherited Functions. A new Computational Model for Syntax-Directed Semantics. *Acta Inf.*, 31:431–477, 1994.
23. E. Meijer and M. Shields. XMλ: A Functional Language for Constructing and Manipulating XML Documents. 1999. Available online http://www.cse.ogi.edu/~mbs/pub/xmlambda/.
24. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *19th ACM Symposium on Principles of Database Systems (PODS)*, pages 11–22, 2000.
25. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. *J. of Comp. Syst. Sci.*, 66:66–97, 2003.
26. M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Proc. Extreme Markup Languages 2000*, 2000.
27. A. Neumann and A. Berlea. fxgrep 4.0. Source Code, 2004.
28. F. Neven. Automata Theory for XML Researchers. *SIGMOD Record*, 31(3):39–46, 2002.
29. T. Schwentick. On Diving in Trees. In *25th Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 660–669. LNCS 1893, Springer-Verlag, 2000.
30. J.W. Thatcher and J.B. Wright. Generalized Finite Automata with an Application to a Decision Problem of Second Order Logic. *Mathematical Systems Theory*, 2:57–82, 1968.
31. W. Thomas. Automata on Infinite Objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–191. Elsevier Science, 1990.
32. A. Tozawa. Towards Static Type Inference for XSLT. In *ACM Symp. on Document Engineering*, pages 18–27, 2001.