Chapter 1

Type Checking of Tree Walking Transducers

Sebastian Maneth

NICTA and University of New South Wales, Sydney, Australia

Sylvia Pott & Helmut Seidl

Technische Universität München, Garching, Germany

Tree walking transducers are an expressive formalism for reasoning about XSLT-like document transformations. One of the useful properties of tree transducers is decidability of type checking: given a transducer and input and output types, it can be checked statically whether the transducer is type correct, i.e., whether each document adhering to the input type is necessarily transformed into documents adhering to the output type. Here, a "type" means a regular set of trees specified by a finite-state tree automaton. Usually, type checking of tree transducers is extremely expensive; already for simple top-down tree transducers it is known to be EXPTIME-complete. Are there expressive classes of tree transducers for which type checking can be performed in polynomial time? Most of the previous approaches are based on inverse type inference. The approach presented here goes the other direction: it uses forward type inference. This means to infer, given a transducer and an input type, the corresponding set of output trees. In general, this set is not a type, i.e., is not regular. However, its intersection emptiness with a given type can be decided. Using this approach it is shown that type checking can be performed in polynomial time, if (1) the output type is specified by a deterministic tree automaton and (2) the transducer visits every input node only a bounded number of times. If the tree walking transducer is additionally equipped with accumulating call-by-value parameters, then the complexity of type checking also depends (exponentially) on the number of such parameters. For this case a fast approximative type checking algorithm is presented, based on context-free tree grammars. Finally, the approach is generalized from trees to forest walking transducers which additionally support concatenation as a built-in output operation.

Contents

1.1. Introduction

The extensible markup language XML is the current standard format for exchanging structured data. Its widespread use has initiated lots of work to support processing of XML on many different levels: customized query languages for XML, such as XQuery, transformation languages like XSLT, and programming language support either in the form of special purpose languages like XDuce, or of binding facilities for mainstream programming languages like JAXB. A central problem in XML processing is the *(static) type checking problem*: given an input and output type and a transformation f, can we statically check whether all outputs generated by f on valid inputs conform to the output type? Since XML types are intrinsically more complex than the types found in conventional programming languages, the type checking problem for XML poses new challenges on the design of type checking algorithms. The excellent survey [MS05] gives an overview of the different approaches to XML type checking.

In its most general setting, the type checking problem for XML transformations is undecidable. Hence, general solutions are bound to be approximative, but seem to work well for practical XSLT transformations [MOS05]. Another approach is to restrict the types and transformations in such a way that type checking becomes decidable; we then refer to the problem as *exact XML type checking*. For the exact setting, types can be considered as regular or *recognizable* tree languages — thus, capturing the expressive strength of virtually all known type formalisms for XML [MLM00].

Even though the class of transformations for which exact type checking is possible is surprisingly large [EM03a; MSV03; MBPS05], the price to be paid for exactness is also extremely high. The design space for exact type checking comes as a huge "exponential wasteland": even for simple top-down transformations, exact type checking is exponential-time complete [MN05], and for more complex transformations such as the k-pebble transducers of [MSV03] the problem is non-elementary. For practical considerations, however, one is interested in useful subclasses of transformations for which exact type checking is provably tractable.

In general, we are interested in type checking of transformations formulated through tree walking transducers (2tts) and macro tree walking transducers (2mtts). A 2tt is similar to an attribute grammar which operates on derivation trees, and has trees as semantic domain (with tree top-concatenation as only semantic operation). The 2mtt generalizes the 2tt by adding formal context-parameters to the attributes, i.e., each attribute is seen as a function which can take parameters of type output tree. Such transducers are very expressive and can simulate most features of transformation languages such as XSLT. Given suitable descriptions (types) of admissible inputs and outputs for a 2tt M, type checking M means to test whether all outputs produced by M on admissible inputs are again admissible. Our main result is: if admissible outputs are described by *deterministic* tree automata, then exact type checking can be done in polynomial time for a large class of practically interesting transformations obtained by putting only mild restrictions on the transducers.

Related Work Approximative type checking for XML transformations is typically based

on (subclasses of) recognizable tree languages. Using XPath as pattern language, XQuery [BC03] is a functional language for querying XML documents. It is strongly-typed and type checking is performed via type inference rules computing approximative types for each expression. Approximative type inference is also used in XDuce [HP03] and its follow-up version CDuce [Fri04]; navigation and deconstruction are based on an extension of the pattern matching mechanism of functional languages with regular expression constructs. Recently, Hosoya et al. proposed a type checking system based on the approximative type inference of [HP02] for parametric polymorphism for XML [HFC05]. Type variables are interpreted as markings indicating the parameterized subparts. In [MOS05] a sound type checking algorithm is proposed (originally developed for the Java-based language XACT [KMS04]) based on an XSLT flow analysis that determines the possible outcomes of pattern matching operations; for the benefit of better performance the algorithm deals with regular approximations of possible outputs.

Milo et al. [MSV03] propose the k-pebble tree transducer (k-ptt) as a formal model for XML transformations, and show that exact type checking can be done for k-ptts using *inverse* type inference. The latter means to start with an output type O of a transformation f and then to construct the type of the inputs by backwards translating O through f. Each k-pebble tree transducer can be simulated by compositions of $k+1$ stay macro tree transducers (smtts) [EM03a], thus, type checking can be solved in time (iterated) exponential in the number of used pebbles. Intuitively, k-pebble tree transducers for $k = 0$ correspond to our 2tts. In [Eng08] it was shown that inverse type inference for 2tts can be done in exponential time, and can be done for k-fold compositions of 2tts in k-fold exponential time. In [MBPS05] it was shown that inverse type inference can be done for a transformation language providing all standard features of most XML transformation languages using a simulation by at most three smtts. Inverse type inference is used in [MN04; MN05] to identify subclasses of top-down XML transformation which have tractable exact type checking. We note that the classes considered there are incomparable to the ones considered in this paper.

1.2. Preliminaries

An XML document can be seen as a sequential representation of an unranked tree. Here is a small example document:

```
<department>
  <employee>
    <data><name>Charles Montgomery Burns</name>...</data>
    <subordinates>
      <employee>
        <data><name>Waylon Smithers</name>...</data>
      </employee>
      <employee> ... </employee> ...
    </subordinates>
  </employee>
  <employee> ... </employee> ...
</department>
```

This example represents a company structure, where each `employee` element has a `data`

element, with the personal data of the employee (e.g. the name). Additionally, it may have a `subordinates` element, which is a collection of further `employee` elements. The represented tree has a root node labeled `department`, which has an arbitrary number of children nodes which are labeled `employee`. An `employee`-node has as first child a `data`-node and has possibly a second child labeled `subordinates`. In the following we refer to this tree as t_B.

Formally, an unranked tree over an alphabet Σ consists of a root node labeled by a symbol $\mathtt{a}$ from Σ and a forest f, written $\mathtt{a}\langle f\rangle$. A forest is a sequence of an arbitrary number of unranked trees, written $t_1 t_2 \dots t_m$. The number m is called the length of the forest. The empty forest, i.e., a forest with length $m = 0$, is denoted by ϵ.

Definition 1.1 (Forests). *Let Σ be a an alphabet (i.e., a finite set). The set $\mathcal{F}_\Sigma$ of forests f over Σ is defined by the grammar rules* $f ::= \epsilon \mid tf, \quad t ::= \mathtt{a}\langle f\rangle\ , \quad$ *where* $\mathtt{a} \in \Sigma$.

Rather than on forests, *tree walking* transducers work on *ranked* trees. There, we assume that a fixed rank is given for every element of Σ, i.e. $\Sigma = \biguplus_{m\in\mathbb{N}} \Sigma^{(m)}$ where $\Sigma^{(m)}$ is the set of all symbols with rank m. We define *rank*$(\mathtt{a}) = m$ for all symbols $\mathtt{a} \in \Sigma^{(m)}$ for $m \geq 0$. The maximal rank *mr*(Σ) is the smallest number m such that $\Sigma^{(m)} \neq \emptyset$ and $\Sigma^{(m+i)} = \emptyset$ for all $i \geq 1$.

Definition 1.2 (Ranked Trees). *Let Σ be a ranked alphabet. The set $\mathcal{T}_\Sigma$ of ranked trees over Σ is defined by the grammar rules* $t ::= \mathtt{a}(\underbrace{t, \dots, t}_{m \text{ times}}) \mid \mathtt{b}$, *where* $\mathtt{a} \in \Sigma^{(m)}$ *and* $\mathtt{b} \in \Sigma^{(0)}$.

In the following we use the term 'tree' as a synonym for *ranked tree*. We fix the set $Y = \{y_1, y_2, \dots\}$ of formal *parameters*. These parameters are of rank 0. For a ranked alphabet Σ, where Σ and Y are disjoint, $\mathcal{T}_\Sigma(Y)$ denotes the set of trees over Σ and Y.

In order to define tree walking transducers on XML documents, we rely on ranked tree representations of forests, e.g., through binary trees. The empty forest then is represented by a leaf with label `e` (where `e` is a new symbol that does not appear in the document). The content of an element node `a` is coded as the left child of `a`, while the forest of right siblings of `a` is represented as the right child (this is the well-known "first-child next-sibling" encoding). Accordingly, the ranks of symbols are either zero or two. Figure 1.1 illustrates this relationship between unranked trees and their representation as binary trees. It shows the tree t_B in the left part and its binary encoding t'_B on the right.

The set $\mathcal{N}(t) \subseteq \mathbb{N}^*$ of all *nodes* v in a ranked tree t is defined as $\mathcal{N}(\mathtt{b}) = \{\epsilon\}$ and $\mathcal{N}(\mathtt{a}(t_1, \dots, t_m)) = \{\epsilon\} \cup \{iv \mid 1 \leq i \leq m, v \in \mathcal{N}(t_i)\}$. where $\mathbb{N}^*$ is the set of strings (including the empty string) over the alphabet of positive natural numbers and ϵ denotes the empty string. The *direction* $\eta(v)$ of a node v indicates whether v is the root of the tree or a particular child, i.e., we define $\eta(\epsilon) = 0$ and $\eta(v'j) = j$. The set $\mathcal{N}(f)$ of nodes of a forest f is defined as $\mathcal{N}(\epsilon) = \{0\}$ and $\mathcal{N}(\mathtt{a}\langle f_1\rangle f_2) = \{0v' \mid v' \in \mathcal{N}(f_1)\} \cup \{(i+1)v' \mid iv' \in \mathcal{N}(f_2)\}$. For a node v in a forest we define the *direction* $\eta(v)$ which now indicates whether v is at the top-level, has a left sibling or both. Thus, $\eta(0) = 0$, $\eta(i) = 1$ for $i > 0$, $\eta(v'0) = 2$ for $v' \neq \epsilon$ and $\eta(v) = 3$ otherwise.

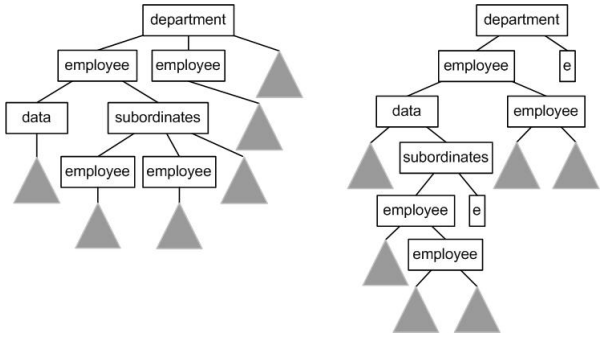


Fig. 1.1. The unranked tree t_B and its binary encoding t'_B.

Note that the definition of nodes of a ranked tree differs from the definition of nodes in a forest consisting of one tree only. Accordingly, also the definitions of *direction* differ. For a ranked tree t and a given node $v \in \mathcal{N}(t)$, $t[v]$ is called the *subtree of* t *located at* v and is defined as $t[\epsilon] = t$ and $\mathtt{a}(t_1, \ldots, t_m)[iv] = t_i[v]$ for $i = 1, \ldots, m$. For example in the left tree t_B in Figure 1.1 the `subordinates` element is the node 1.2. Here we write 1.2 instead of 12 for the second child of the first child of the root, to distinguish between this node and the twelfth son of the root. With $lab_t(v)$ we refer to the label of the node v in a tree t, or $lab(v)$, if t is given by the context. In the left example $lab_{t_B}(1.2) = \mathtt{subordinates}$ and in the right tree $lab_{t'_B}(1.2) = \mathtt{employee}$.

The height of a tree is recursively defined as $\mathit{height}(\mathtt{b}) = 1$ for $\mathtt{b}$ of rank 0 and $\mathit{height}(\mathtt{a}(t_1, \ldots, t_m)) = 1 + \mathit{max}(\mathit{height}(t_1), \ldots, \mathit{height}(t_m))$ for $\mathtt{a}$ of rank m. The height of a tree is the maximal length of a path from the root to a leaf. If we consider trees on right-hand sides of rules, we have to deal with *state calls*. In this exposition, we will first consider *tree walking transducers* (cf. Section 1.3) which do not support accumulating parameters. For tree walking transducers, state calls are of the form $q(op)$ (op stands for up, $stay$ or $down_i$). For these, we define $\mathit{height}(q(op)) = 1$ for all states q and all operations op. In Section 1.6 we then will add parameters to state calls to obtain *macro tree walking transducers*. Then a state call has the form $q(op, t_1, \ldots, t_n)$ where t_i $(1 \leq i \leq n)$ are ranked trees over $\Sigma \cup Y$ and further state calls ($Y = \{y_1, \ldots, y_k\}$ is a set of parameters). In this case, we define the height recursively by: $\mathit{height}(q(op, t_1, \ldots, t_n)) = 1 + \mathit{max}(\mathit{height}(t_1), \ldots, \mathit{height}(t_k))$. The *size* of a tree t is defined as the number of nodes, i.e., $|t| = |\mathcal{N}(t)|$. Similar notions also apply to forests. In particular, the subforest $f[v]$ at a node v in a forest is defined as f if $i = 0$ and as $f'[i-1]$ if $i > 0$ and $f = tf'$. For $v = iv'$ with $v' \neq \epsilon$ $f[v] = f_1[v']$ if $i = 0$ and $f = \mathtt{a}\langle f_1 \rangle f_2$, and $f[v] = f'[(i-1)v']$ if $i > 0$ and $f = tf'$. The *label* $lab_f(v)$ of v in the forest f is defined by $lab_f(0) = \epsilon$ if $f = \epsilon$, and $\mathit{lab}_f(iv') = \mathtt{a}$ if $i = 1$ and $v' = \epsilon$, $\mathit{lab}_f(iv') = \mathit{lab}_{f_1}(v')$ if $i = 1$ and $v' \neq \epsilon$, and $\mathit{lab}_f(iv') = \mathit{lab}_{f_2}((i-1)v')$ if $i > 0$ and $f = \mathtt{a}\langle f_1 \rangle f_2$. Note that the label at a node in a forest thus either is from Σ or equals the empty forest ϵ.

1.3. Tree Walking Transducers

Tree transducers describe transformations τ from trees to sets of trees over a ranked alphabet Σ, i.e., $\tau : \mathcal{T}_\Sigma \rightarrow 2^{\mathcal{T}_\Sigma}$. Consider for example a transformation which translates documents as in the example before into a collection of all employees which are listed under a new root node labeled `staff`. Besides a `name` element, these new `employee` elements now contain an element `boss` if the employee is the subordinate of someone. For our example document, the transformation produces:

```
<staff>
  <employee>
    <data> <name> Charles Montgomery Burns </name> ... </data>
  </employee>
  <employee>
    <data> <name> Waylon Smithers </name> ... </data>
    <boss> <name> Charles Montgomery Burns </name> ... </boss>
  </employee>
  <employee> ...  </employee> ...
</staff>
```

The corresponding tree is referred as s_B and its binary encoding as s'_B. A tree walking transducer starts at the root of the input tree. Depending on the label of the current node, the direction and the state, it produces a tree with leaves which again may contain state calls for nodes of the input tree. These recursively accessed nodes are determined according to the directives specified in the right-hand side of the applied rule: on directive up, the father of the current node is processed, on directive $down_i$, the i-th child and on directive $stay$ the current node itself. Tree walking transducers can be considered as generalizations of *top-down* tree transducers. While top-down tree transducers are only allowed to move downward in the input tree, tree walking transducers may also stay at the current node or move upward in the tree.

Example 1.1. Using our representation of forests by binary trees (Fig. 1.1), the transformation of our example is realized by a tree walking transducer M_{staff} with the following rules.

$$
\begin{aligned}
&1 \quad q_I(\texttt{department}) && \rightarrow \texttt{staff}(q(down_1), \texttt{e}),\\
&2 \quad q(\texttt{employee}) && \rightarrow \texttt{employee}(\texttt{data}(q_{data}(down_1), q_{boss}(stay)), q_{sub}(down_1)),\\
&3 \quad q(\texttt{e}) && \rightarrow q_{up}(up),
\end{aligned}
$$

together with a state q_{data} for copying the personal data

$$
4 \quad q_{data}(\texttt{data}) \rightarrow copy(down_1),
$$

as well as a state q_{boss} to find the boss

$$
\begin{aligned}
&5 \quad q_{boss}(\texttt{employee}) && \rightarrow q_{boss}(up),\\
&6 \quad q_{boss}(\texttt{department}) && \rightarrow \texttt{e},\\
&7 \quad q_{boss}(\texttt{subordinates}) && \rightarrow \texttt{boss}(q_{data}(up), \texttt{e}),
\end{aligned}
$$

and a state q_{sub}, which processes the subordinates

$$
\begin{aligned}
&8 \quad q_{sub}(\texttt{data}) && \rightarrow q_{sub}(down_2),\\
&9 \quad q_{sub}(\texttt{subordinates}) && \rightarrow q(down_1),\\
&10 \quad q_{sub}(\texttt{e}) && \rightarrow q_{next}(up).
\end{aligned}
$$

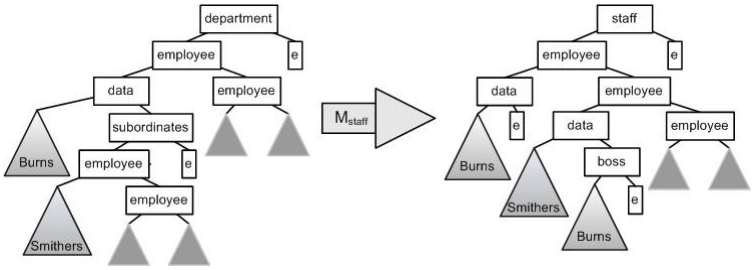


Fig. 1.2. The tree t'_B and its output tree s'_B of the transformation of the 2tt M_{staff}.

The state q_{next} searches (in dfs-manner) the next employee

$$\begin{aligned} &11 \quad q_{next}(\texttt{data}) && \to q_{next}(up),\\ &12 \quad q_{next}(\texttt{employee}) && \to q(down_2), \end{aligned}$$

together with a state q_{up} for going to the boss, if there is no further subordinate

$$\begin{aligned} &13 \quad q_{up}(\texttt{employee}) && \to q_{up}(up),\\ &14 \quad q_{up}(\texttt{subordinates}) && \to q_{next}(up),\\ &15 \quad q_{up}(\texttt{department}) && \to \texttt{e}, \end{aligned}$$

where state *copy* in line 4 is meant to copy the content of `data` (i.e., the left child in the binary representation). The initial state is q_I, which means that we start with state q_I at the root of the tree. The output trees of this transformation are binary representations of the lists of all members of staff. The root, which is labeled `staff`, has a right child with label `e`. The left child of `staff` has label `employee` whose left child is a `data`-node (with the personal data and the boss) and whose right child is a chain of `employee`-nodes. Figure 1.2 illustrates this transformation for the binary example tree t'_B resulting in the tree s'_B. ◁

The example illustrates that the "first-child next-sibling" encoding of forests implies that the up-operation of the tree walking transducer may not necessarily access directly the father in the forest representation but may instead reach the *left* sibling – depending whether or not the current node is a left or right child (i.e., has direction 1 or 2). In the example this was no problem: the state q_{boss} simply proceeds upwards in the tree representation until a node with the right label is reached. A direct construction of forest walking transducers, which provides the operations up, $down$, $left$ and $right$ will be presented in Section 1.7. For the moment, we restrict ourselves to tree walking transducers on ranked trees (which perhaps are encodings of unranked forests).

Formally, the rules of a tree walking transducers are slightly more general than the ones shown in Example 1.1: additional to the label of the current node, the left-hand side of a rule also checks the direction of the current node, i.e., whether the current node is the root node (direction is zero), or whether it is the i-th child of its parent node. It is well-known that in the case of tree walking automata (viz. tree walking transducers with output

symbols $\{0, 1\}$ of rank zero), such direction tests (or "child number" test) are crucial: without them, the automaton cannot even realize a depth-first left-to-right traversal over the input tree, i.e., it cannot systematically search through every node of the input. For some translations, however, direction tests are not needed (such as our Example 1.1). In that example, we must think of every rule as existing in (at most) three incarnations, for direction zero (root node), direction one (left child), and direction two (right child). For instance, the q-rule for `employee`-nodes (rule number 2 of the example) is needed in the following two incarnations:

$$\begin{array}{ll} \text{2a} & q(\texttt{employee}, 1) \;\to \texttt{employee}(\texttt{data}(q_{data}(down_1), q_{boss}(stay)), q_{sub}(down_1)) \\ \text{2b} & q(\texttt{employee}, 2) \;\to \texttt{employee}(\texttt{data}(q_{data}(down_1), q_{boss}(stay)), q_{sub}(down_1)) \end{array}$$

Recall from the Preliminaries that the maximal rank of symbols in a ranked alphabet Σ is denoted by $mr(\Sigma)$.

Definition 1.3 (2tt). *A* tree walking transducer M *(*2tt *for short) is a tuple* (Q, Σ, R, Q_0) *where* Q *is a set of states,* Σ *is a ranked alphabet,* $Q_0 \subseteq Q$ *is a set of initial states, and* R *is a finite set of rules. A rule is of the form* $q(\mathtt{a}, \eta) \to \zeta$ *where* $q \in Q$, $\mathtt{a} \in \Sigma^{(m)}$, $m \geq 0$, $\eta \geq 0$ *and* ζ *is a tree generated by the grammar* $\zeta ::= \mathtt{b}(\underbrace{\zeta, \ldots, \zeta}_{m' \text{ times}}) \mid q'(op)$, *with* $\mathtt{b} \in \Sigma^{(m')}$, $m' \geq 0$, $q' \in Q$, *and* $op \in \{stay, up\} \cup \{down_i \mid 1 \leq i \leq m\}$.

Tree walking transducers are also called 2-way tree transducers, because they generalize to trees the well known concept of 2-way finite state transducer on words (see, e.g., [Gre78]).

Conventionally, tree transducers are defined over two ranked alphabets of input and output symbols. In Definition 1.3 of a 2tt M we only use one alphabet Σ which contains input and output symbols. If we want to distinguish the two, we say that $\mathtt{a} \in \Sigma$ is an *input symbol* if a appears on the left-hand side of a rule of M; we say that it is an *output symbol* if it appears in the right-hand side of a rule of M. In Example 1.1, `data` is an input and output symbol and `boss` is an output symbol of M_{staff}.

In practice, transducers also have to cope with *unknown* labels in the input such as, e.g., portions of text which then either are ignored or copied into the output. In order to deal with this, we could simply extend our formalism by an extra symbol $\bullet$ of any given rank which serves as a placeholder for unknown labels of this rank. This idea can be extended to placeholders for unknown elements of different atomic types, for instance `String`, `Number` or `Date`. Thus, we can describe the so called "Simple Types" of XML Schema (cf. [FW04]).

For a right-hand side ζ, we also write $\zeta = s[q_1(op_1), \ldots, q_c(op_c)]$ to refer to all occurrences of state calls in the right-hand side; there $s \in \mathcal{T}_\Sigma(X)$ is a tree which contains exactly one occurrence of the variable x_i for $i = 1, \ldots, c$. Note that s does not contain state calls. For example the right-hand side of the rule in line 2 in the Example 1.1 can be written as $s[q_{data}(down_1), q_{boss}(stay), q_{sub}(down_1)]$ where $s = \texttt{employee}(\texttt{data}(x_1, x_2), x_3)$.

A 2tt is called *deterministic* iff there is at most one state in the set Q_0 and for every triple $(q, \mathtt{a}, \eta)$ of a state, a symbol and a direction there is at most one rule with $q(\mathtt{a}, \eta)$ as left-hand side. The example 2tt M_{staff} is deterministic.

Intuitively, the meaning of the expressions of a right-hand side is as follows: the output can either be an element $\mathtt{b}$ whose content is recursively determined, or a recursive call to some state q' on the current input node, on its father or on its i-th subtree. The match patterns in the left-hand side of the rules are restricted to the form "$\mathtt{a}, \eta$", i.e., it is only allowed to check the label of the current input node and its direction. Thus, the transformation of a 2tt M starts at the root node of the input t with one of the initial states. A state q can be applied to an input node v with label $lab(v) = \mathtt{a}$ and direction $\eta = \eta(v)$ if there is a rule with left-hand side $q(\mathtt{a}, \eta)$. The evaluation continues on a child vi of v for each occurrence of a state call $q'(down_i)$, at v itself for each occurrence of a state call $q'(stay)$, and at the parent of v, for each occurrence of a state call $q'(up)$.

Hence, the *meaning* $[\![q]\!]_t$ of a state q of M with respect to an input tree t can be defined as a function from the nodes (of the input tree) to sets of trees, i.e., $[\![q]\!]_t : \mathcal{N}(t) \to 2^{\mathcal{T}_\Sigma}$. The values $[\![q]\!]_t$ for all q are jointly defined as the *least* functions satisfying: $[\![q]\!]_t(v) \supseteq ([\![\zeta]\!]_t(v))$ for rule $q(\mathtt{a}, \eta) \to \zeta$ where v is a node of t with $lab(v) = \mathtt{a}$ and $\eta = \eta(v)$ with

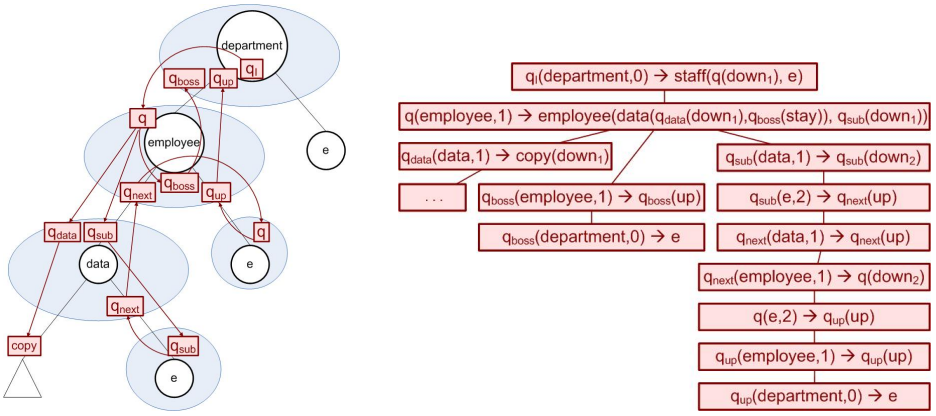
$$[\![\mathtt{b}(\zeta_1, \ldots, \zeta_m)]\!]_t(v) = \{\mathtt{b}(t'_1, \ldots, t'_m) \mid t'_i \in [\![\zeta_i]\!]_t(v)\}$$
$$[\![q'(op)]\!]_t(v) = [\![q']\!]_t([\![op]\!]_t(v)),$$

where op stands for $stay$, up or $down_i$ for $1 \leq i \leq rank(\mathtt{a})$, and $[\![op]\!]_t$ is defined by: $[\![stay]\!]_t(v) = v$, $[\![down_i]\!]_t(v) = vi$, and $[\![up]\!]_t(vi) = v$. The transformation τ_M realized by the 2tt M on an input tree t and sets T of input trees, respectively, is defined by $\tau_M(t) = \bigcup\{[\![q_0]\!]_t(\epsilon) \mid q_0 \in Q_0\}$ and $\tau_M(T) = \bigcup\{\tau_M(t) \mid t \in T\}$. For a deterministic 2tt M the transformation τ_M is a partial function $\tau_M : \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$. The *domain* of the transducer is the domain of the transformation, i.e., $dom(M) = dom(\tau_M) = \{t \mid \tau_M(t) \neq \emptyset\}$. As usual, the *size* $|M|$ of a 2tt M is the sum of the sizes of all its rules where the size of a rule $q(\mathtt{a}, i) \to \zeta$ is defined as $3 + |\zeta|$. Recall that $|\zeta|$ equals the number of nodes of ζ.

Applying the 2tt M_{staff} from before to t'_B we obtain the tree s'_B. The right-hand sides of rules in a 2tt may be arbitrarily large and contain arbitrarily many state calls. Dealing with such rules increases the complexity of some algorithms on 2tts. Thus, we give a normal form for 2tts where the number of state calls in right-hand sides is bounded by the maximal rank of output symbols. In the particular case where we consider binary representations of forests, the number of state calls in right-hand sides can be restricted to 2.

Lemma 1.1. *For every 2tt M a 2tt M' can be constructed in time $\mathcal{O}(|M|)$ such that (i) $\tau_{M'} = \tau_M$ and (ii) the right-hand side of each rule of M' contains at most k occurrences of states where k is the maximal rank of the output symbols of M.*

Proof. Let $M = (Q, \Sigma, R, Q_0)$. Intuitively, the idea of the construction is to introduce auxiliary states for all proper subtrees which contain more than 1 state call. For a symbol $\mathtt{a} \in \Sigma$ and direction η, let $Z_{\mathtt{a},\eta}$ denote the set of all subterms with more than one state call in right-hand sides of rules for $\mathtt{a}, \eta$. For each $\zeta \in Z_{\mathtt{a},\eta}$, we introduce a fresh state $q_{\mathtt{a},\eta,\zeta}$. Assume that $\zeta = \mathtt{b}(\zeta_1, \ldots, \zeta_m)$. Then we introduce the new rule $q_{\mathtt{a},\eta,\zeta}(\mathtt{a}, \eta) \to \mathtt{b}(\zeta'_1, \ldots, \zeta'_m)$ where $\zeta'_j = \zeta_j$ if ζ_j contains at most one occurrence of a state, and $\zeta'_j = q_{\mathtt{a},\eta,\zeta_j}(stay)$ otherwise. We construct $M' = (Q', \Sigma, R', Q_0)$ as follows. The set of rules R' of the new transducer consists of all these newly constructed rules. Additionally, we add for every rule



(a) The behavior of 2tt M_{staff} on tree t_e. (b) A run of M_{staff} on the tree t_e.

Fig. 1.3. The 2tt M_{staff} on the tree t_e.

$q(\mathtt{a},\eta) \to \mathtt{b}(\zeta_1,\ldots,\zeta_m)$ of M a new rule $q(\mathtt{a},\eta) \to \mathtt{b}(\zeta'_1,\ldots,\zeta'_m)$ where for every j, $\zeta'_j = \zeta_j$ if ζ_j contains at most one occurrence of a state, and $\zeta'_j = q_{\mathtt{a},\eta,\zeta_j}(stay)$ otherwise. The set of states Q' contains all states of Q and additionally the new states $q_{\mathtt{a},\eta,\zeta}$ for every symbol $\mathtt{a} \in \Sigma$, direction η and every term $\zeta \in Z_{\mathtt{a},\eta}$.

The resulting transducer M' has a new state at most for every non-leaf node of a right-hand side of a rule in M. Thus, in the worst case, we have at most $|M|$ new states. In the new rules the right-hand side of the original rule of M is split in its subtrees. Thereby, we have $|M'| \in \mathcal{O}(|M|)$. □

In order to describe the behavior of the 2tt $M = (Q, \Sigma, R, Q_0)$ on a fixed input tree t, we are also going to define *runs* of M. A run can itself be described by a ranked tree over the set of rules. Here, the rank of a rule $q(\mathtt{a},\eta) \to \zeta$ is given by the number of occurrences of calls $q'(op)$ in ζ to states q' in Q.

Definition 1.4 (Run). *Let q denote a state of M and v a node in the input tree t of direction η which is labeled with* $\mathtt{a}$. *Assume that $r : q(\mathtt{a},\eta) \to \zeta$ is a rule in R with $\zeta = s[q_1(op_1),\ldots,q_m(op_m)]$. Then the tree $\rho = r(\rho_1,\ldots,\rho_m) \in \mathcal{T}_R$ is a (q,v)-*run *of the 2tt M on the tree t, if for every $1 \le i \le m$, ρ_i is a (q_i,v_i)-run of M on t where v_i is obtained from v by operation op_i. The output $\tau(\rho)$ produced by a run ρ is defined by $\tau(\rho) = s[\tau(\rho_1),\ldots,\tau(\rho_m)]$. A (q_0,ϵ)-run for an initial state q_0 is also called* accepting run *of M on t.*

If M is deterministic, then there exists at most one accepting run on every tree.

Example 1.2. Figure 1.3(a) shows the behavior of the (deterministic) example 2tt M_{staff} on the tree $t_e = \mathtt{department}(\mathtt{employee}(\mathtt{data}(\ldots,\mathtt{e}),\mathtt{e}),\mathtt{e})$ which describes a department with one employee. All states in an oval around a node are applied to this node. The picture includes the dependences of the states. For example, consider the employee node $v = 1$

(with label $lab(v) = \texttt{employee}$). There, we have the state q. In the 2tt, there is just one rule with left-hand side $q(\texttt{employee}, 1)$:

$$2a \quad q(\texttt{employee}, 1) \to \texttt{employee}(\texttt{data}(q_{data}(down_1), q_{boss}(stay)), q_{sub}(down_1))$$

Thus, we have a $(q, 1)$-run $\rho = r_{2a}(\rho_1, \rho_2, \rho_3)$ where ρ_1, ρ_3 are $(q_{data}, 1.1)$- and $(q_{sub}, 1.1)$-runs, respectively, and ρ_2 is a $(q_{boss}, 1)$-run. This is illustrated by the three arrows starting at q at 1. The Figure 1.3(b) shows a (q_I, ϵ)-run $\rho' = r_1(\rho)$. The state $copy$ was not detailed in Example 1.1. Accordingly, the $(copy, 1.1.1)$-run here is not complete. The output $\tau(\rho)$ of this run is the tree $\texttt{staff}(\texttt{employee}(\texttt{data}(\ldots, \texttt{e}), \texttt{e}), \texttt{e})$. ◁

Accepting runs are another approach to define the semantics of a 2tt. Indeed, this operational semantics of a 2tt coincides with the denotational semantics provided first.

Theorem 1.1. *For a tree t and a 2tt M the following two statements are equivalent. (1) There is an accepting run ρ of M for t with $\tau(\rho) = s$ and (2) $s \in \tau_M(t)$.*

Theorem 1.1 can be proved by fixpoint induction. The denotational view on the semantics of a 2tt allows us to use fixpoint arguments for proving the correctness of constructions, whereas the operational view is better suited for combinatorial arguments.

1.3.1. *Notes and References*

Top-down tree transducers were invented by Rounds and Thatcher [Rou70; Tha69]. Top-down tree transducers terminate for every input tree, because they process the input tree strictly top-down. While the height increase of a top-down tree transducer is at most linear, the size increase is at most exponential (viz. the translation of a monadic tree with n nodes into a full binary tree of height n). A nondeterministic top-down tree transducer can associate at most double exponentially many output trees to a given input tree; e.g. the transducer with the three rules $q(\texttt{a}, \eta) \to \texttt{b}(q(down_1), q(down_1))$, $q(\texttt{a}, \eta) \to \texttt{c}(q(down_1), q(down_1))$, and $q(\texttt{e}, 1) \to \texttt{e}$ for $\eta \in \{0, 1\}$. Tree walking transducers with output strings were invented in [AU71]; by adding the ability to generate output trees rather than strings, we obtain the tree walking transducer of this paper. It can be seen as the k-pebble tree transducer of [MSV03], for the case that $k = 0$. In [KS81] it was shown that tree walking transducers without child number test are not useful: they cannot even check whether all leaves of input trees are labeled by some symbol a. As mentioned in [EM03a], in the total deterministic case the tree walking transducer is essentially the same as the attribute grammar [Knu68]. Similar to the fact that circularity of attribute grammars is decidable, it is possible to change any deterministic tree walking transducer in such a way that all runs are terminating [EM03a]. This is not possible for nondeterministic tree walking transducers, because they can associate infinitely many output trees to a given input tree (viz. the transducer with the two rules $q(\texttt{a}, 0) \to \texttt{b}(q(stay))$, and $q(\texttt{a}, 0) \to \texttt{e}$). The normal form of Lemma 1.1 is similar to the one for pebble macro tree transducers given in Theorem 16 of [EM03a]. Attribute grammars with tree output are also called "attributed tree transducers" [Fül81]; for total deterministic such transducers (which coincide with our

tree walking transducers when they are total deterministic) it is known that the size-to-height relationship of input tree to output tree is linear, and that the number of different output subtrees in an output tree is linear in the size of the corresponding input tree (see, e.g., [FV98]).

1.4. Type Checking

In this section, we present general techniques for certifying that all outputs produced by a transducer M for trees of a given input type are *well-formed*, i.e., comply with some given output type O. This problem is called *type checking* of the transducer M. Here, a type is just a set of trees, i.e., a *tree language*. Clearly, the tractability of type checking heavily depends on the class of languages used as types, and the class of transformations.

The tree s_B is an example for the output language of the 2tt M_{staff}. Such output trees are binary trees with a root labeled with `staff` and a right-comb of `employee` nodes as left subtree. It is the first-child next-sibling representation of a tree which has a root labeled with `staff` and arbitrary many `employee` nodes as children. A DTD describing this type (not the binary representation) is the following (where `content` stands for further personal data which are not specified here):

```
<!ELEMENT staff (employee)*>
<!ELEMENT employee (data, boss)>
<!ELEMENT data (name, content)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT boss (name, content)>
<!ELEMENT content ... >
```

1.4.1. *Type Checking by Forward Type Inference*

Type checking a transducer M means to verify that all trees produced by M for input trees in the given input type I are necessarily contained in the given output type O. If τ is the transformation induced by the transducer M, we want to check whether or not $\tau(I) \subseteq O$, where $\tau(I) = \{\tau(t) \mid t \in I\}$. If this check succeeds, then we say that M *type checks w.r.t.* I *and* O. We solve this problem by forward inference, i.e., we determine whether $\tau(I) \cap \overline{O} = \emptyset$, where $\overline{O}$ is the complement of the type O. In order to decide emptiness of this intersection, we proceed in two steps. First, we construct from M a transducer $M_{\overline{O}}$ which produces only those outputs of M which are from $\overline{O}$. This construction is presented in Section 1.5.1. Then we present methods for deciding emptiness of transducers (w.r.t. I).

1.4.2. *Tree Automata*

There are several specification formalisms for XML types, such as DTD, XML Schema, or RELAX NG. For our purpose, the particular type formalisms is not essential, as all of these formalisms can be abstracted by recognizable (or: regular) tree languages. Thus, each type definition can be translated into a finite tree automaton. XML Schema specifications, e.g.,

can be considered as simple classes of deterministic top-down automata. We briefly recall crucial definitions of finite tree automata.

Definition 1.5 (bta). *Let Σ be a ranked alphabet. A* bottom-up finite state tree automaton *A (over Σ),* bta *for short, is a tuple (P, Σ, δ, F) where P is a finite set of states, $F \subseteq P$ is a set of accepting states, and δ is a finite set of transitions $(p, \mathtt{a}, p_1 \dots p_m)$ where $\mathtt{a} \in \Sigma^{(m)}$ and $p, p_1, \dots, p_m \in P$.*

A transition $(p, \mathtt{a}, p_1 \dots p_m)$ denotes that if, for all $1 \leq i \leq m$, A arrives in state p_i after processing some tree t_i, then it can assign state p to the tree $\mathtt{a}(t_1, \dots, t_m)$. Technically, a p-run ρ of A on a tree $t = \mathtt{a}(t_1, \dots, t_m) \in \mathcal{T}_\Sigma$ is a tree $\rho = r(\rho_1, \dots, \rho_m) \in \mathcal{T}_\delta$ where r is a transition $(p, \mathtt{a}, p_1 \dots p_m) \in \delta$ and ρ_i is a p_i-run of A for t_i. The tree language $\mathcal{L}(A)$ accepted by A consists of the trees $t \in \mathcal{T}_\Sigma$ by which A can reach an accepting state, i.e. it exists a p-run ρ of A for t with $p \in F$; the latter run is called *accepting run of A on t*. A bottom-up tree automaton $A = (P, \Sigma, \delta, F)$ is *deterministic* (dbta) if for each symbol $\mathtt{a} \in \Sigma^{(m)}$ and every tuple $p_1 \dots p_m$ of states, there is at most one state p with $(p, \mathtt{a}, p_1 \dots p_m) \in \delta$, i.e., δ induces a partial function of type $\Sigma \times P^* \to P$. A bta is called *total* if there is at least one rule $(p, \mathtt{a}, p_1 \dots p_m) \in \delta$ for all $m \geq 0$, $\mathtt{a} \in \Sigma^{(m)}$, and $p_1, \dots, p_m \in P$.

We may also interpret the transitions of a bta in a *top-down* fashion. Then we obtain the known top-down tree automaton (tta) which starts at the input root node and assigns states to the children of a node, depending on the label of the node and the current state.

Definition 1.6 (dtta). *A bta is called* deterministic top-down *(*dtta *for short), if the set of final states is a singleton set, and the transition relation δ induces a partial function $P \times \Sigma \to P^*$, i.e., for each state $p \in P$ and each symbol $\mathtt{a} \in \Sigma^{(m)}$, there is at most one sequence of states $p_1 \dots p_m \in P^m$ with $(p, \mathtt{a}, p_1 \dots p_m) \in \delta$.*

As usual, the *size* $|A|$ of a finite state tree automaton A is the sum of sizes of all its transitions. A transition $(p, \mathtt{a}, p_1 \dots p_m)$ has size $m + 2$. Let BTA, DBTA, and DTTA denote the classes of all languages definable by btas, dbtas, and dttas, respectively. It is known that BTA $=$ DBTA equals the class of regular tree languages, and that DTTA is properly contained in this class.

Example 1.3. Coming back to the transformation from Example 1.1, the set of valid output documents should be lists of staff members, more precisely: `staff` should contain a possibly empty sequence of `employee` elements; Each `employee` element should contain a `data` element and optionally, a `boss` element.

A bta describing (the binary representations of) this set is given by $A_{\text{staff}} = (P, \Sigma, \delta, F)$ where $P = \{r_{\text{staff}}, r_{\text{empl}}, r_{\text{data}}, r_{\text{name}}, r_{\text{boss}}, r_{\mathtt{e}}, \dots\}$ and $\delta = \{(r_{\text{staff}}, \mathtt{staff}, r_{\text{empl}}\ r_{\mathtt{e}}), (r_{\text{empl}}, \mathtt{employee}, r_{\text{data}}\ r_{\text{empl}}), (r_{\text{empl}}, \mathtt{e}), (r_{\text{data}}, \mathtt{data}, r_{\text{name}}\ r_{\text{boss}}), (r_{\text{boss}}, \mathtt{boss}, r_{\text{name}}\ r_{\mathtt{e}}), (r_{\text{boss}}, \mathtt{e}), (r_{\text{name}}, \mathtt{name}, r_{\text{content}}\ r_{\mathtt{e}}), (r_{\text{boss}}, \mathtt{boss}, r_{\text{content}} r_{\mathtt{e}}), (r_{\mathtt{e}}, \mathtt{e})\}$, where r_{content} is the state characterizing valid personal data of employees. The set of accepting states is, thus, given by $F = \{r_{\text{staff}}\}$. Note that this bta is in fact deterministic top-down. ◁

In the previous example, the bta ran on the first-child next-sibling encoding of forests as binary ranked trees. For convenience, we call such a finite tree automaton also *finite forest automaton* (short: bfa). Again, if it is deterministic or deterministic top-down, then we abbreviate it with dbfa and dtfa, respectively.

1.4.3. *Basic Properties of BTAs*

The approach which we advocate here is called *forward type checking* (cf. Section 1.4.1). Assume that O is the type of all valid output trees. In order to check that the transducer M produces only outputs in O, we construct a transducer which for every input t, only produces those output trees of M which are *not* valid, i.e., which are in $\overline{O}$ (the complement of O). Thus, type correctness for M is reduced to emptiness of the auxiliary transducer $M_{\overline{O}}$. For this idea to work, it is useful to have effective constructions which take the specification of a type and return a specification for its complement. For a *total* dbta $A = (P, \Sigma, \delta, F)$ this construction is simple: we need to exchange accepting and non-accepting states, i.e., replace F with $P \backslash F$. Since every regular tree language can be accepted by a total dbta, this construction implies that the complement of a regular tree language is a regular tree language, too. The complement of a type described by a deterministic top-down tree automaton is a regular language as well, but not necessarily in DTTA. The obvious technique for constructing an automaton for the complement therefore is to transform the deterministic top-down automaton into a total deterministic bottom-up automaton and then apply the complement construction for total dbtas. This first construction, however, possibly incurs an exponential blow-up in the number of states. Therefore, we approve a different approach: instead of constructing a *deterministic* automaton for the complement, we construct a *non-deterministic* automaton. The latter can be achieved by only moderately increasing the size.

Lemma 1.2. *For a dtta A over the ranked alphabet Σ there is a bta A' over Σ with $\mathcal{L}(A') = \mathcal{T}_\Sigma \backslash \mathcal{L}(A)$ and $|A'| \in \mathcal{O}((|A| + |\Sigma|) \cdot mr(\Sigma))$.*

Proof. Intuitively, the automaton A' guesses a path in the input tree to some node where the original automaton A fails. Formally, let $A = (P, \Sigma, \delta, \{p_0\})$ and define $A' = (P', \Sigma, \delta', \{p'_0\})$ with $P' = \{p' \mid p \in P\} \cup \{\bullet\}$ for a new state $\bullet \notin P$. A state p' is meant to generate only trees for which there is *no* p-run of A. The state $\bullet$ describes arbitrary trees, i.e., the language $\mathcal{T}_\Sigma$. The set δ' of transitions of the new bta is defined as follows:

- for every state $p \in P$ and $\mathtt{a} \in \Sigma^{(0)}$, $(p', \mathtt{a}, \epsilon) \in \delta'$ whenever $(p, \mathtt{a}, \epsilon) \notin \delta$
- for every transition $(p, \mathtt{a}, p_1 \dots p_m) \in \delta$ with $rank(\mathtt{a}) \geq 1$, and for every $i \in \{1, \dots, m\}$ let $(p', \mathtt{a}, \bullet^{i-1} p'_i \bullet^{m-i}) \in \delta'$
- for every state $p \in P$ and $\mathtt{a} \in \Sigma^{(m)}$, $(p', \mathtt{a}, \bullet^m) \in \delta'$ whenever $\forall p_1, \dots, p_m \in P$: $(p, \mathtt{a}, p_1 \dots p_m) \notin \delta$
- for every $\mathtt{a} \in \Sigma^{(m)}$, $(\bullet, \mathtt{a}, \bullet^m) \in \delta'$.

For the correctness of the construction, we claim that for every state p of A, and every input tree t, A' has a p'-run on t iff A has no p-run on t. This claim can be proven by induction on the height of input trees.

Now, let $k = mr(\Sigma)$ be the maximal rank of symbols in Σ. For each transition in δ we get at most k new transitions in δ' (one for each successor state). Additionally, we require a new rule of length at most $k + 2$ for each symbol in Σ. Thus, the size of the automaton A' is in $\mathcal{O}((|A| + |\Sigma|) \cdot mr(\Sigma))$. □

Example 1.4. For the dtta $A_{\text{staff}} = (P, \Sigma, \delta, \{r_{\text{staff}}\})$ in Example 1.3, the bta $A'_{\text{staff}} = (P', \Sigma, \delta', \{r'_{\text{staff}}\})$ for the complement has the following transitions for the label `staff`: $(r'_{\text{staff}}, \texttt{staff}, r'_{\text{empl}}\bullet)$, $(r'_{\text{staff}}, \texttt{staff}, \bullet r'_{\texttt{e}})$, and $(r', \texttt{staff}, \bullet\bullet)$ for all $r' \in P' \setminus \{r'_{\text{staff}}\}$ ◁

1.4.4. *Notes and References*

XML type definition languages such as DTDs [W3C00], XML Schema [Fal01], or RELAX NG [CM] are closely related to the regular tree languages [MLM00; Nev02], that is, to the class of tree languages recognized by finite tree automata.

Tree automata are a well studied formalism in computer science, dating back to the late 1960s. For surveys on tree automata, please see [GS84; GS97; CDG+07]. Tree automata inherit most of the good properties of finite automata on strings, such as effective closure under Boolean operations and decidability of emptiness. An important property which will be used later for type checking, is that emptiness of btas can be decided in linear time (see, e.g., Theorem 1.7.4 in [CDG+07]).

Theorem 1.2. *Given a bta A it can be decided in linear time whether or not $\mathcal{L}(A) = \emptyset$.*

Just as in the string case, nondeterministic bottom-up tree automata can be determinized (with a potential and sometimes unavoidable exponential blow up in automaton size). This is not the case for top-down tree automata: the class DTTA of languages accepted by deterministic top-down tree automata is a strict subclass of BTA which does not even contain all finite languages; a famous example of a language not in DTTA is the set $U = \{\texttt{f}(\texttt{a}, \texttt{b}), \texttt{f}(\texttt{b}, \texttt{a})\}$. Note that for a given bta, it is decidable if its language is in DTTA; this is due to the fact that DTTA languages can be characterized by the "path-closed" property [Cou78; Vir81]; the latter means that the trees in the languages are exactly obtained by combining all paths of the corresponding path language. The language U for instance is not path-closed. Using a similar example, it is easily shown that DTTA is not closed under complementation (and neither under union).

1.5. Type Checking of Tree Walking Transducers

1.5.1. *Intersecting Tree Walking Transducers with Output Types*

In this section we present techniques to type check 2tts against regular tree languages. For a given 2tt we build a second 2tt which produces only output trees in the *complement*

of the output type, and otherwise realizes the same transformation as the original 2tt. If the output type is described by a total dbta $A = (P, \Sigma, \delta, F)$, the complement will be recognized by the total dbta $\bar{A} = (P, \Sigma, \delta, P \setminus F)$. For a given dtta there exists a bta describing the complement (cf. Lemma 1.2). Thus, it is sufficient to construct a 2tt M_A for a 2tt M and a bta A (which may be the complement automaton of a total dbta or dtta) with $\tau_{M_A}(t) = \tau_M(t) \cap \mathcal{L}(A)$ for every tree t.

Theorem 1.3. *For every 2tt M and every bta A there is a 2tt M_A with*

$$\tau_{M_A}(t) = \tau_M(t) \cap \mathcal{L}(A)$$

for all $t \in \mathcal{T}_\Sigma$. The size $|M_A|$ of M_A is in $\mathcal{O}(|M| \cdot |A|^{d+1})$, where d is the maximal number of occurrences of states in right-hand sides of M.

Proof. Let $M = (Q, \Sigma, R, Q_0)$ and $A = (P, \Sigma, \delta, F)$. For each state q in Q and all states $p \in P$ we generate new states for M_A of the form $\langle q, p\rangle$. Such a state is meant to generate only trees $t \in \mathcal{T}_\Sigma$ for which there is a run of A starting at the leaves and reaching the root of t in state p. The rules of the new 2tt M_A are $\langle q, p\rangle(\mathtt{a}, \eta) \to \zeta'$ for every rule $q(\mathtt{a}, \eta) \to \zeta$ of M and $\zeta' \in \tau^p[\zeta]$. The sets $\tau^p[.]$ are inductively defined by:

$$\begin{array}{ll} \tau^p[\mathtt{b}(\zeta_1, \ldots, \zeta_m)] & = \{\mathtt{b}(\zeta'_1, \ldots, \zeta'_m) \mid (p, \mathtt{b}, p'_1 \ldots p'_m) \in \delta \wedge\ \forall i : \zeta'_i \in \tau^{p'_i}[\zeta_i]\} \\ \tau^p[q'(op)] & = \{\langle q', p\rangle(op)\}. \end{array}$$

The set of initial states of M_A is $Q'_0 = Q_0 \times F$. By fixpoint induction, we verify for every state q, every input tree $t \in \mathcal{T}_\Sigma$, every node $v \in \mathcal{N}(t)$ and every state p that:

$$[\![\langle q, p\rangle]\!]_t(v) = [\![q]\!]_t(v) \cap \{s \in \mathcal{T}_\Sigma \mid \exists \text{ run } \rho \text{ on } s \text{ with } \rho(\epsilon) = p\}$$

For each state in M we have at most $|A|$ new states in M_A. If we have c occurrences of state calls in the right-hand side of a rule r of M, with the state on the left-hand side, we obtain at most $|A|^{c+1}$ new rules for r in M_A. Therefore, the new 2tt is of size $\mathcal{O}(|M| \cdot |A|^{d+1})$ where d is the maximal number of occurrences of state calls in right-hand sides in M. □

Considering only binary trees, we obtain size $\mathcal{O}(|M| \cdot |A|^3)$ for the intersection 2tt (with Lemma 1.1). The last step is to decide whether $\tau_{M_A} \neq \emptyset$. Thereto, we build a bta describing the domain of M_A. This will be done after completing the example.

Example 1.5. Let us try to type check the 2tt $M_{\text{staff}} = (Q, \Sigma, R, Q_0)$ via forward type inference. According to Lemma 1.1, we restrict the maximal number of state calls in right-hand sides to 2. In our example 2tt, the rule $q(\mathtt{employee}, \eta) \to \mathtt{employee}(\mathtt{data}(q_{data}(down_1), q_{boss}(stay)), q_{sub}(down_2))$ has three state calls. We obtain the new rules: $q(\mathtt{employee}, \eta) \to \mathtt{employee}(q'(stay), q_{sub}(stay))$ and $q'(\mathtt{employee}, \eta) \to \mathtt{data}(q_{data}(down_1), q_{boss}(stay))$. According to the proof of Lemma 1.1, the new state q' is $q_{\mathtt{employee},\mathtt{data}(q_{data}(down_1),q_{boss}(stay))}$. Consider again the dtta $A_{\text{staff}} = (P, \Sigma, \delta, \{r_{\text{staff}}\})$ as output type. The complement bta $A'_{\text{staff}} = (P', \Sigma, \delta', \{r'_{\text{staff}}\})$ is given in Example 1.4. The intersection 2tt $(M_{\text{staff}})_{A'_{\text{staff}}}$ is given by $(Q \times P', \Sigma, R', Q_0 \times \{r'_{\text{staff}}\})$. In what follows we show how the first few rules in R' are constructed from R and δ' as follows. For $q_I(\mathtt{department}, 0) \to \mathtt{staff}(q(down_1), \mathtt{e})$ and r'_{staff} we obtain the rule $\langle q_I, r'_{\text{staff}}\rangle(\mathtt{department},$

$0) \to \texttt{staff}(\langle q, r'_{\text{empl}}\rangle(down_1), \texttt{e})$. For $q(\texttt{employee}, \eta) \to \texttt{employee}(q'(stay), q_{sub}(down_2))$ and r'_{empl} we obtain $\langle q, r'_{\text{empl}}\rangle(\texttt{employee}, \eta) \to \texttt{employee}(\langle q', r'_{\text{data}}\rangle(stay), \langle q_{sub}, \bullet\rangle(down_2))$ and $\langle q, r'_{\text{empl}}\rangle(\texttt{employee}, \eta) \to \texttt{employee}(\langle q', \bullet\rangle(stay), \langle q_{sub}, r'_{\text{empl}}\rangle(down_2))$. For $q(\texttt{e}, \eta) \to q_{up}(up)$ and all $r \in P'$ we obtain $\langle q, r\rangle(\texttt{e}, \eta) \to \langle q_{up}, r\rangle(up)$. ◁

1.5.2. *Deciding Emptiness of 2TTs*

In order to check the emptiness of a tree walking transducer w.r.t. a given input type, we construct a nondeterministic finite state automaton (cf. Section 1.4.2) which then is checked for emptiness. First, for a 2tt M and an input type I we define an *alternating tree walking automaton* M' which ignores the output of the 2tt, but apart from that imitates the behavior of M on trees in I. For M', we then construct a nondeterministic bta $A_{M'}$ accepting all trees t such that $\tau_M(t) \neq \emptyset$. The right-hand sides of transitions of an alternating tree walking automaton are conjunctions. Whereas the empty conjunction, i.e., $\bigwedge \emptyset$, equals **true**.

Definition 1.7 (atwa). *An* alternating tree walking automaton *(*atwa *for short) is a tuple* $M = (Q, \Sigma, \delta_M, Q_0)$ *where* Q *is a finite set of states,* Σ *a ranked input alphabet,* $Q_0 \subseteq Q$ *a set of initial states, and* δ_M *a finite set of rules of the form* $q(\texttt{a}, \eta) \to q_1(op_1) \wedge \ldots \wedge q_c(op_c)$ *with* $c \geq 0$*,* $q, q_1, \ldots, q_c \in Q$*,* $\texttt{a} \in \Sigma^{(m)}$*,* $m \geq 0$*,* $\eta \geq 0$*, and for* $1 \leq i \leq c$*,* $op_i \in \{stay, up\} \cup \{down_j \mid j = 1, \ldots, m\}$.

A transition $q(\texttt{a}, \eta) \to H$ is also called q-rule. An atwa traverses a tree like a 2tt, but produces no output. The language $\mathcal{L}(M)$ of an atwa M is defined as the set of all trees for which there exists an accepting run of M.

Definition 1.8 (Run). *For an atwa* $M = (Q, \Sigma, \delta_M, Q_0)$*, assume that* $r : q(\texttt{a}, \eta) \to H$ *is a rule in* δ_M *with* $H = q_1(op_1) \wedge \ldots \wedge q_c(op_c)$*. Then the tree* $\rho = r(\rho_1, \ldots, \rho_c) \in \mathcal{T}_{\delta_M}$ *is a* (q, v)*-run of the atwa* M *on the tree* t*, if* v *is a node of* t *with direction* η *and label* a *and for all* i*,* ρ_i *is a* (q_i, v_i)*-run of* M *on* t *where* v_i *is obtained from* v *by operation* op_i*. A* (q_0, ϵ)*-run* ρ *with* $q_0 \in Q_0$ *is also called* accepting.

A subtree $\rho[w]$ of a run ρ on a node w which is a (q, v)-run for some state q and some node v of t is called (q, v)*-subrun*. The set of rules which are applied to one node v during a run ρ on t is the set $\mathit{rules}_\rho(v) = \{r \mid \exists \rho_1, \ldots, \rho_c : r(\rho_1, \ldots, \rho_c)$ is a (q, v)-subrun of $\rho\}$.

Lemma 1.3. *Assume that* M *is a 2tt. Then an atwa* M' *can be constructed in linear time such that for every input tree* t*,* M' *has an accepting run for* t *iff* M *has an accepting run for* t.

Proof. The atwa M' has the same set of states as M. The rules of M' are obtained from those of M by replacing every right-hand side ζ of M with the conjunction of all $q(op)$ occurring in ζ. Formally, let $M = (Q_M, \Sigma, R, Q_0)$ be a 2tt. Then, the atwa M' is defined by $M' = (Q_M, \Sigma, \delta', Q_0)$ for a set δ' of transitions $\delta' = \{[r] \mid r \in R\}$ where $[q(\texttt{a}, \eta) \to \zeta] = q(\texttt{a}, \eta) \to [\zeta]$ and $[\zeta] = q_1(op_1) \wedge \ldots \wedge q_c(op_c)$ for a right-hand side

$\zeta = s[q_1(op_1), \ldots, q_c(op_c)]$. In order to prove the correctness of this construction, we first extend the translation [.] from rules to trees of rules. For $\rho = r(\rho_1, \ldots, \rho_c) \in \mathcal{T}_R$, $[\rho]$ is inductively defined as the tree $[\rho] = [r]([\rho_1], \ldots, [\rho_c])$. Then we claim that ρ is a (q, v)-run of M iff $[\rho]$ is a (q, v)-run of M'. The proof is by structural induction on ρ. Since for every (q, v)-run ρ' of M', some $\rho \in \mathcal{T}_R$ exists with $[\rho] = \rho'$, we conclude that $\tau_M(t) \neq \emptyset$ iff $t \in \mathcal{L}(M')$. □

As an example for this translation of 2tts into atwas, consider the 2tt M_{staff} (Example 1.1). For the second rule, we get: $r'_2 : q(\texttt{employee}, \eta) \to q_{data}(down_1) \wedge q_{boss}(stay) \wedge q_{sub}(down_1)$. In order to accept only trees of an input type I, we enlarge an atwa M. Let I be given by a finite state tree automaton (P, Σ, δ_A, F) and $M = (Q, \Sigma, \delta_M, Q_0)$. For every rule $q_0(\mathtt{a}, \eta) \to H$ with $q_0 \in Q_0$ and every $p_f \in F$ we enhance the atwa with the rule $q_0(\mathtt{a}, \eta) \to p_f(stay) \wedge H$. Additionally for every $(p, \mathtt{a}, p_1 \ldots p_m) \in \delta_A$, we add the rule $p(\mathtt{a}, \eta) \to \bigwedge_{i \leq m} p_i(down_i)$.

In an accepting run of an atwa, subruns for the same state and node are interchangeable. Thus, we define *uniform* runs as runs where for each state q and each node v, the (q, v)-subruns are the same.

Definition 1.9 (Uniform Run). *A (q, v)-run ρ of an atwa M on a tree t is called* uniform *if, for every state q, every node v in t, and every two (q, v)-subruns ρ_1, ρ_2 of ρ, $\rho_1 = \rho_2$.*

If an atwa is deterministic, i.e., for each state q, direction η and label $\mathtt{a}$, there is at most one rule of the form $q(\mathtt{a}, \eta) \to H$, then *every* (q', v)-run is uniform. In general, this may not be the case. However, we have:

Lemma 1.4. *For an atwa M and an input tree t the following two statements are equivalent. (1) M has an accepting run for t and (2) M has a uniform accepting run for t.*

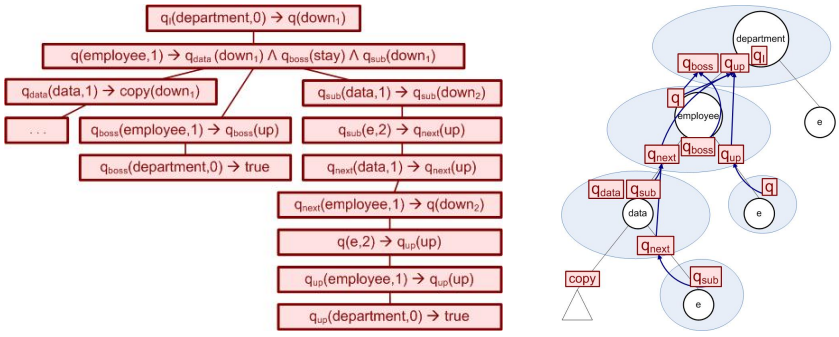
Proof. Every uniform accepting run is an accepting run. For the reverse direction, it suffices to prove that for every (q, v)-run ρ of M on t, there is also a uniform (q, v)-run of M on t. For that, we consider the set $B(\rho)$ of all pairs (q', v') for which ρ contains more than one (q', v')-run as a subtree. We proceed by induction on the cardinality of the set $B(\rho)$. If the set $B(\rho)$ is empty, ρ is already uniform. Now assume $B(\rho)$ is non-empty. Then ρ contains a subtree ρ_1 with the following two properties:

(1) ρ_1 is a (q_1, v_1)-run with $(q_1, v_1) \in B$;

(2) all subtrees ρ'_1 of ρ_1 are (q', v')-runs with $(q', v') \notin B$.

Then we construct from ρ a tree ρ' by replacing every occurrence of a subtree in ρ which is a (q_1, v_1)-run with ρ_1. Then ρ' is again a (q, v)-run on t, but now the set $B(\rho') \subseteq B(\rho) \backslash \{(q_1, v_1)\}$ contains at least one element less. Thus, by induction hypothesis applied to ρ', there is a (q, v)-run on t which is uniform. □

Figure 1.3(a) shows the behavior of the 2tt M_{staff} on the tree t_e. The corresponding atwa M'_{staff} yields the same behavior. The Figure 1.3(b) shows an accepting run ρ of M_{staff} on t_e. The corresponding run $[\rho]$ of M'_{staff} is illustrated in Figure 1.4(a). This run is uniform.



(a) A uniform run of the atwa M'_{staff} on the tree t_e. (b) The mappings μ for that run.

Fig. 1.4. Behavior of M'_{staff} on t_e.

In order to decide emptiness of an atwa M and accordingly of a 2tt, we construct a *nondeterministic* bottom-up finite state tree automaton A_M. In order to accept the domain of M, the bta A_M guesses uniform accepting runs. Since A_M visits each node in the input tree at most once, it *guesses* at every node all transitions which are applied at this node during a uniform run of M.

Technically, the states of the bta A_M consist of guessed directions together with *partial* mappings $\mu : Q \to 2^Q$ of states to sets of states: $\mu(q) = B$ at a given node v means that the (q, v)-run on the input tree will cause calls $q'(up)$ at v only for states q' from B. For a mapping μ we refer to the domain as $\text{dom}(\mu) = \{q \mid \mu(q) \text{ is defined}\}$. Furthermore, to each node v in the input tree we implicitly attach the set $T_v = \mathit{rules}_\rho(v)$ collecting the atwa rules which are applied to the node v in an accepting run ρ of the atwa on the input tree t. Since ρ is uniform, the set $\mathit{rules}_\rho(v)$ contains at most one q-rule for every q.
A pair $\langle \mu, \eta \rangle$ is accepting if $\eta = 0$ and for the partial mapping μ, $q_0 \in \text{dom}(\mu)$ for some accepting state $q_0 \in Q_0$ of the atwa M and $\mu(q) = \emptyset$ for all $q \in \text{dom}(\mu)$.

Now assume that $\mathtt{a}$ is a label of arity m, η is a direction and $\mu, \mu_i : Q \to 2^Q$ are partial mappings ($i = 1, \ldots, m$). Then $(\langle \mu, \eta \rangle, \mathtt{a}, \langle \mu_1, 1 \rangle, \ldots, \langle \mu_m, m \rangle)$ is a transition of A_M iff there is a set T of rules of the atwa M with the following properties. Let Q_s, Q_u and $Q_{d,i}$ ($i = 1, \ldots, m$) denote the set of states q for which there is a q-rule in T, the set of states q' with a recursive call $q'(up)$, and the sets of states q' with a recursive call $q'(down_i)$ in some right-hand side of rules in T, respectively. Then the set T of rules should have the following properties:

(1) All rules in T have a left-hand side of the form $q(\mathtt{a}, \eta)$, where $q \in Q$.
(2) Assume $q(\mathtt{a}, \eta) \to q_1(op_1) \wedge \ldots \wedge q_c(op_c) \in T$. Then we have for every $j \leq c$:
   - If $op_j = stay$, then T also contains a q_j-rule, i.e., $q_j \in Q_s$;
   - If $op_j = down_i$, then $q_j \in \text{dom}(\mu_i)$.
(3) Whenever $q' \in \text{dom}(\mu_i)$ for i, then $\mu_i(q') \subseteq Q_s$.
(4) Consider the following graph G with set of vertices $V = \{q(stay) \mid q \in Q_s\} \cup \{q(down_i) \mid i = 1, \ldots, m, q \in Q_{d,i}\} \cup \{q(up) \mid q \in Q_u\}$ and the following set E of

edges:

- If a q-rule in T contains a call $q'(op)$, then $(q(stay), q'(op)) \in E$;
- If $\mu_i(q) = B_i$ is defined, then $(q(down_i), q'(stay)) \in E$ for all $q' \in B_i$.

The resulting directed graph $G = (V, E)$ should be acyclic, and the mapping μ is obtained from G as follows:

- $q \in \text{dom}(\mu)$ iff $q \in Q_s$ and
- $\mu(q) = B$ iff the set B equals the set of all vertices $q'(up)$ which are reachable in G from $q(stay)$.

The size of A_M is exponential in the size of M. We give a detailed example for this construction in Example 1.6. Now, we state the correlation of runs of A_M and of M.

Lemma 1.5. *For a tree t the following statements are equivalent. (1) There is a uniform accepting run of M on t and (2) there is an accepting run of A_M on t.*

Proof. $(1) \Rightarrow (2)$: Let ρ be a uniform accepting run of M on a tree t. For a node v of t with label $\mathtt{a}$ and direction η, let T_v denote the set of all atwa rules applied at the root in subruns of ρ starting at v, i.e., $T_v = \mathit{rules}_\rho(v)$. We then construct for every node v of t with label $\mathtt{a} \in \Sigma^{(m)}$ and direction η, a state μ_v and a transition $r_v = (\langle \mu_v, \eta\rangle, \mathtt{a}, \langle \mu_{v1}, 1\rangle, \ldots, \langle \mu_{vm}, m\rangle)$ of the bta A_M.

The sets T_v allow us to construct a directed graph G_t. The set V_t of vertices of G_t are given by the set of all pairs (q, v) for nodes v of t and states q for which there is a q-rule in T_v. The set E_t of edges consists of:

- all pairs $((q, v), (q', v))$ where the q-rule in T_v contains a call $q'(stay)$;
- all pairs $((q, v), (q', vi))$ where the q-rule in T_v contains a call $q'(down_i)$;
- all pairs $((q, v), (q', v'))$ where the q-rule in T_v contains a call $q'(up)$ and $v = v'i$ for some i.

The graph G_t is acyclic. Moreover since the uniform run ρ is accepting, every vertex in V_t is reachable from some vertex (q_0, ϵ) with $q_0 \in Q_0$.

The graph G_t allows to construct partial mappings μ_v for every node v. $q \in \text{dom}(\mu_v)$ iff (q, v) is a vertex of G_t. Assume (q, v) is a vertex in G_t. We consider two cases. If $v = \epsilon$, then T_ϵ cannot contain any q-rule which has an up-call. In this case, $\eta(v) = 0$, and we set $\mu_\epsilon(q) = \emptyset$. Otherwise, assume that $v = v'i$. Then $\eta(v) = i$ and $q' \in \mu_v(q)$ iff there is an edge $((q_1, v), (q', v'))$ in G_t where (q_1, v) is reachable from (q, v) by a path which contains only vertices (q_2, v_2) referring to nodes v_2 from the subtree at v, i.e., to nodes which have v as a prefix. If no such state q' exists, then $\mu_v(q) = \emptyset$.

It now can be verified for every node v with label $\mathtt{a} \in \Sigma^{(m)}$ and direction η that $(\langle \mu_v, \eta\rangle, \mathtt{a}, \langle \mu_{v1}, 1\rangle, \ldots, \langle \mu_{vm}, m\rangle)$ constitutes a transition of A_M (with T_v as set of rules of the atwa). Since by construction, $\langle \mu_\epsilon, 0\rangle$ is an accepting state of A_M, we have, thus, constructed an accepting run of A_M for t.

$(2) \Rightarrow (1)$: Let ρ' be an accepting run of A_M on the tree t, and let $\langle \mu_v, \eta\rangle$ and r_v denote the state and transition of A_M attained for the node v in t. We can find sets T_v

conforming to the properties of the transition relation of M. These allow to construct a graph G'_t analogously to the graph G_t above. By the definition of the transition relation of A_M, G'_t is acyclic. This allows us to define for every vertex (q, v) in G'_t, the number $h(q, v)$ as the maximal length of a path in G'_t to a leaf, i.e., a vertex with out-degree 0. Using the sets T_v of atwa rules, we now construct for every node v and atwa rule $r : q(\mathtt{a}, \eta) \to H$ from T_v with $H = q_1(op_1) \wedge \ldots \wedge q_c(op_c)$, a tree $\rho[q, v]$ by $\rho[q, v] = H(\rho[q_1, v_1], \ldots, \rho[q_c, v_c])$, where $v_j = [\![op_j]\!](v)$. Note that all these trees are well-defined, since the height of $\rho[q, v]$ precisely equals $h(q, v)$. Moreover, the tree $\rho[q, v]$ is a (q, v)-run of the atwa M on t. Since every (q', v')-subrun of this tree equals the (q', v')-run $\rho[q', v']$, this run is also uniform. In particular, the tree $\rho[q_0, \epsilon]$ constitutes a uniform accepting run of the atwa M. □

By Lemmas 1.3, 1.4, and 1.5, the bta A_M recognizes the domain of the given 2tt M, which gives us Theorem 1.4. Note that this implies, by Theorem 1.2, that also emptiness of M's domain (and hence of M's translation τ_M) can be decided in exponential time.

Theorem 1.4. *Assume M is a 2tt. Then a bta A can be constructed in exponential time such that $\mathcal{L}(A) = dom(M)$. Thus, emptiness for a 2tt can be decided in deterministic exponential time.*

Example 1.6. In this example we consider again the 2tt M_{staff} and its corresponding atwa M'_{staff}. The size of the bta $A_{M'_{\text{staff}}}$ is exponential in the size of the atwa or 2tt. Therefore, we only construct states occurring in a run of $A_{M'_{\text{staff}}}$ on the tree t_e in Figure 1.3(a). In Figure 1.4(a), a uniform accepting run ρ of the atwa is illustrated. The run yields the sets T_v. For instance, for the node labeled with `employee` we get $T_1 = \{q(\mathtt{employee}, 1) \to q_{boss}(stay) \wedge q_{sub}(down_1) \wedge q_{data}(down_1),\ \ q_{boss}(\mathtt{employee}, 1) \to q_{boss}(up),\ \ q_{up}(\mathtt{employee}, 1) \to q_{up}(up),\ \ q_{next}(\mathtt{employee}, 1) \to q(down_2)\}$. The graph G_{t_e} in the proof of Lemma 1.5 is similar to the graph in Figure 1.3(a). There, it spans the tree t_e. To obtain the graph G_{t_e} we have to replace a vertex q, which is located at a node v of t_e, by (q, v). The mappings μ_v are illustrated in Figure 1.4(b). For each state q fixed at a node v, $\mu_v(q)$ is defined. If q has no outgoing edges then $\mu_v(q) = \emptyset$. Otherwise, it is the set of all direct successors of q in this figure. For example

$$\begin{aligned}
\mu_\epsilon &= \{q_I \mapsto \emptyset, q_{up} \mapsto \emptyset, q_{boss} \mapsto \emptyset\}\\
\mu_1 &= \{q \mapsto \{q_{up}, q_{boss}\}, q_{next} \mapsto \{q_{up}\}, q_{boss} \mapsto \{q_{boss}\}, q_{up} \mapsto \{q_{up}\}\}\\
\mu_{1.1} &= \{q_{data} \mapsto \emptyset, q_{sub} \mapsto \{q_{next}\}, q_{next} \mapsto \{q_{next}\}\}\\
\mu_{1.2} &= \{q \mapsto \{q_{up}\}\}
\end{aligned}$$

Note that $q_{up} \notin \mu_{1.2}(q_{next})$ although $(q_{up}, 1)$ is reachable from $(q_{next}, 1.1)$ in G_t. But the path contains $(q, 1.2)$ and 1.1 is not a prefix of 1.2. In order to illustrate a transition of bta $A_{M'_{\text{staff}}}$, consider, e.g., the transition $(\langle\mu_1, 1\rangle, \mathtt{employee}, \langle\mu_{1.1}, 1\rangle, \langle\mu_{1.2}, 2\rangle) \in \delta_A$ with the set T_1. All rules in T_1 agree in the input label `employee` and the direction 1 (condition 1). Also it contains a q_{boss}-rule for the call $q_{boss}(stay)$. For all states q_{sub}, q_{data}

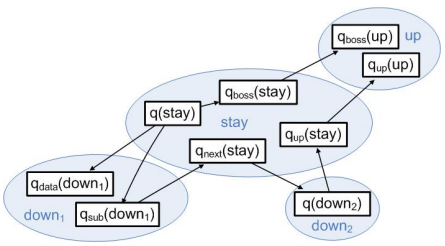


Fig. 1.5. Graph G for transition $(\langle\mu_1, 1\rangle, \mathtt{employee}, \langle\mu_{1.1}, 1\rangle, \langle\mu_{1.2}, 2\rangle)$.

of occurring $down_1$-calls, the mapping $\mu_{1.1}$ is defined. Likewise, for q with a $down_2$-call, $q \in \text{dom}(\mu_{1.2})$. Thus, T_1 also conforms with condition 2. For condition 3, we verify that T_1 has rules both for q_{next} and q_{up}. For the last property, we construct the graph G. The set of vertices is

$$\begin{aligned} V = \{&q(stay), q_{boss}(stay), q_{next}(stay), q_{up}(stay),\\ &q_{data}(down_1), q_{sub}(down_1), q(down_2),\\ &q_{boss}(stay), q_{up}(stay)\} \end{aligned}$$

The edges are illustrated in Figure 1.5. As the last condition requires, the graph G is acyclic, and we can read off the mapping μ_1.

Also we verify, that $\langle \mu_\epsilon, 0 \rangle$ is an accepting state. According to Lemma 1.5, the resulting run ρ' is an accepting run of $A_{M'_{\text{staff}}}$ on t. ◁

1.5.3. *Efficient Subcases*

In the previous section, we have provided an algorithm for deciding emptiness of atwas and, thus, also of 2tts which runs in exponential time. This algorithm is indeed worst-case optimal. Not withstanding that, this algorithm allows us to identify subclasses of transducers where emptiness can be decided in polynomial time.

We call a transducer M *b-bounded*, if every accepting run ρ of M has at most b subruns starting at node v. We call M *strictly b-bounded* if every accepting run ρ of M visits each node v in the input tree at most b times, i.e., has at most b subrun *occurrences* starting at node v. The same definitions are also employed for atwas.

Note that the definition of b-boundedness does not exclude that the same node v is traversed arbitrarily often: if so, however, these traversals will be copies of at most b distinct traversals.

Note further that for a given transducer M it is decidable whether or not there exists a b such that M is b-bounded; the same holds for strict b-boundedness. To see this, add for each input symbol a new marked symbol of the same rank. We then consider input trees in which exactly one node is labeled by a marked symbol (this is a regular input tree language). Finally, we change the transducer M in such a way that it produces a specific output tree, for each subrun that starts at the marked input node (resp. for each time the marked node is visited), and other than that does not produce any output. The output of the new transducer, when applied to input trees with exactly one node marked, is finite if and only if the transducer is b-bounded for some b (resp. strictly b-bounded for some b). The finiteness is decidable for a very large class of tree transformations [DE98], which contains the pebble tree transducers (and hence also the 2tts) by [EM03a].

Consider the construction from the last subsection of a bta A_M which accepts the same language as an atwa M. If the atwa M is b-bounded, then it suffices to consider sets T of atwa rules of size b. Also, this means that partial mappings μ need to be taken into account which are of the form: $B \to 2^{B'}$ for subsets B, B' of states of cardinalities at most b.

Note that the number of subsets of size at most b of a set with n elements is bounded by $\frac{1}{b!}(n+1)^b$. Thus, the number of the partial mappings μ can be bounded by: $(n+1)^{2b}\frac{2^{b^2}}{b!}$ if n is the number of states of M. We will not provide an explicit estimation of the number of transitions of the bta since it crucially also depends on other parameters such as the number of rules of M which agree in input symbol and direction (which is typically small). We just note that for b-bounded M, the size of the bta A_M is polynomial in the size of M. The occurring exponent, though, is bounded by $\mathcal{O}(kb^2)$ where k is the maximal arity of an input symbol. Summarizing, we find that emptiness for b-bounded 2tts can be done in polynomial time. Note, however, that neither b-boundedness or strict b-boundedness is preserved by our construction to reduce the number of state calls in right-hand sides. For an efficient method for type-checking, we also require that the bound on the number of visits to every node in the input tree is preserved under the intersection construction. In this respect, we observe:

Lemma 1.6. *If M is strictly b-bounded and A is a bta, then M_A is also strictly b-bounded. If M is just b-bounded, this need not be the case.*

We thus obtain a polynomial-time algorithm for the class of strictly b-bounded 2tts where the number of occurrences of state calls in right-hand sides is also bounded.

1.5.4. *Conclusion*

In this section we presented techniques to type check 2tts against regular tree languages. Our approach is forward type inference. For that purpose, for a given 2tt M we build a second 2tt which produces only output trees in the complement of the output type, and otherwise realizes the same transformation as the original 2tt. For a bta A describing the complement of the output type, the size of the new 2tt M_A is in $\mathcal{O}(|M| \cdot |A|^{d+1})$ where d is the maximal number of occurrences of states in right-hand sides in M; for binary trees and with Lemma 1.1 it is in $\mathcal{O}(|M| \cdot |A|^3)$.

For this intersection 2tt M_A we build an alternating tree walking automaton M', which imitates the behavior of M_A, but does not produce any output. This construction can be done in linear time. And then, in order to decide emptiness of the atwa M' and accordingly of M_A, we construct a nondeterministic bottom-up finite state tree automaton $A_{M'}$. In general, this construction is exponential in the size of M'. Hence emptiness of a 2tt can be decided in exponential time — a result which has already been known for a long time, see the notes at the end of Section 1.5.5. The general approach, however, allowed us to identify more efficient subclasses. These, we have discussed in Section 1.5.3. If M' is b-bounded, then emptiness can be decided in polynomial time where the exponent of the polynomial only depends on b^2 if the transducer is two-way, i.e., uses up-operations. A closer inspection of the construction of a bta from an atwa, though, reveals that the exponent can be reduced to b if the transducer is stay top-down, i.e., uses no up-operations (but possibly $stay$s). The construction for the intersection, on the other hand, is polynomial in the sizes both of the 2tt and the bta — but may be exponential in the number of occurrences

of state calls in right-hand sides. Also, if we start with a b-bounded 2tt M, the construction may not preserve b-boundedness. Instead, the bound on the number of visits to an input node may be increased as much as by a factor of the number of states of the bta. If the 2tt M is *strictly* b-bounded, this property will be retained and also the atwa M' is strictly b-bounded.

As a last step of verifying whether a 2tt type checks w.r.t. input and output types $\mathcal{L}(A_I)$ and $\mathcal{L}(A)$ (for a bta A_I), we construct a bta C with $\mathcal{L}(C) = L(A_{M'}) \cap \mathcal{L}(A_I)$ using the obvious product construction (see, e.g., Section 1.3 in [CDG+07]) such that $|C| \in \mathcal{O}(|A_{M'}| \cdot |A_I|)$. According to Theorem 1.2, we can test whether $\mathcal{L}(C) = \emptyset$ (which means that M type checks w.r.t. $\mathcal{L}(A_I)$ and $\mathcal{L}(A)$) in time linear in $|C|$.

Theorem 1.5. *Deciding whether a strictly b-bounded 2tt M type checks w.r.t. regular tree languages I and O, given by btas A_I and A_O, is polynomial in the size of M, A_I, and A_O, but exponential in $b^2 \cdot (d+1)$ where d is the maximal number of occurrences of state calls in right-hand sides. If M has no up-operations, the exponent can be improved to $b \cdot (d+1)$.*

1.5.5. *Notes and References*

Our definition of atwas is equivalent to the alternating two-way finite tree automaton of [Slu85]. Note that alternating tree automata have recently been used in the context of a practical implementation of type checking for tree transducers [FH07].

The intersection of a 2tt with a given output type (Theorem 1.3) can be seen as a sequential composition of the 2tt with a translation in FTA; the latter is the class of partial identity mappings for regular tree languages. With this in mind, we can, for instance, obtain that top-down tree transducers allow a similar result as the one in Theorem 1.3: by Corollary 2(1) of [Bak79], top-down tree transducers are closed under composition with linear and nondeleting top-down tree transducers; since FTA is included in the latter class, we obtain the desired result for top-down tree transducers. It is an interesting open problem whether a similar composition result holds for 2tts, i.e., whether 2tts are closed under composition with linear and nondeleting 2tts.

The notion of b-boundedness is similar to the notion of finite-copying in tree transducers, see, e.g., [ERS80; EM99]. Similar to the results of [EM03b], it probably holds that b-bounded transformations are of linear size increase. A more static version of b-boundedness is the single-use restriction known for attribute grammars [Gie88]. According to [EM99], it can probably be shown that total, deterministic, strictly b-bounded 2tts are equivalent to single-use restricted attribute grammars. In Section 5 of [MPS07] a similar result as Theorem 1.5 has been shown for stay-macro tree transducers (cf. also the discussion in Section 1.6.4)

Engelfriet et al. show in [EHS07] (Theorem 5), that for every 2tt M (TT in [EHS07]) a regular tree grammar G can be constructed in exponential time such that G generates the domain of τ_M. They refer to the relationship between 2tt and attributed tree transducers explained in [EM03a] and a result of [Bar82] — giving the Theorem 1.4 above. The result of Theorem 1.4 has also been stated in Theorem A.2 of [CGKV88] with a proof sketch that

uses a game theoretic interpretation of acceptance due to Muller and Schupp. The result also appears as Theorem 1 in [Eng08], where a finite state automaton for the domain of a tree-walking transducer (twt) is constructed in exponential time; Theorem 2 of that paper states that inverse type inference is in k-fold exponential time, for k-fold compositions of twts.

1.6. Macro Tree Walking Transducers

In our running example we have considered the 2tt M_{staff} which lists the members of staff of a department. Although in general, several employees may have the same boss, the transducer spawns for every employee a separate computation to determine the corresponding boss. Conceptually as well as technically, it would be more convenient to determine the boss first, store it in some accumulating parameter and then propagate it to each of his employees. For this reason, we enhance tree transducers with accumulating parameters. A tree transducer with accumulating parameters is also called *macro tree transducer*.

Example 1.7. We omit the state q_{boss} and store the data of the boss in the first parameter y_b. The transformation of the next employee which is not a subordinate of the current, is then stored in the second parameter (y_n). By this construction, we completely omit the states q_{sub}, q_{next}, and q_{up}. The transducer consists of the following rules, for all $\eta \in \{1, 2\}$:

$$
\begin{aligned}
&1\quad q_I(\texttt{department}, 0) && \to \texttt{staff}(q(down_1, \texttt{e}, \texttt{e}), \texttt{e})\\
&2\quad q(\texttt{employee}, \eta, y_b, y_n) && \to \texttt{employee}(\texttt{data}(q_{data}(down_1), y_b),\\
&3\quad && \qquad q(down_1, \texttt{boss}(q_{data}(down_1), \texttt{e}), q(down_2, y_b, y_n)))\\
&4\quad q(\texttt{e}, \eta, y_b, y_n) && \to y_n\\
&5\quad q(\texttt{data}, 1, y_b, y_n) && \to q(down_2, y_b, y_n),\\
&6\quad q(\texttt{subordinates}, 2, y_b, y_n) && \to q(down_1, y_b, y_n),\\
&7\quad q_{data}(\texttt{data}, 1) && \to copy(down_1).
\end{aligned}
$$

where state *copy* is meant to copy the content of `data` (i.e., the left child in the binary representation). ◁

For the formal definition of macro tree walking transducer we assume that every state $q \in Q$ has a fixed rank, i.e., $Q = \biguplus_{n\in\mathbb{N}} Q^{(n)}$ where $Q^{(n)}$ is the set of all states with rank n.

Definition 1.10 (2mtt). *A* macro tree walking transducer M *(*2mtt *for short) is a tuple* (Q, Σ, R, Q_0) *where* Q *is a set of ranked states,* Σ *is a ranked alphabet,* $Q_0 \subseteq Q^{(1)}$ *is a set of initial states, and* R *is a finite set of rules of the form* $q(\texttt{a}, \eta, y_1, \dots y_n) \to \zeta$*, where* $q \in Q^{(n+1)}$*,* $\texttt{a} \in \Sigma^{(m)}$*,* $n, m \geq 0$*,* $\eta \geq 0$ *is a direction and* $y_1, \dots, y_n$ *are the accumulating parameters of* q*. Possible right-hand sides are described by the grammar*

$$\zeta \ ::= \ \texttt{b}(\overbrace{\zeta, \dots, \zeta}^{m' \text{ times}}) \mid y_j \mid q'(op, \overbrace{\zeta, \dots, \zeta}^{n' \text{ times}}),$$

with $m', n' \geq 0$*,* $\texttt{b} \in \Sigma^{(m')}$*,* $j \in \{1, \dots, n\}$*,* $q' \in Q^{(n'+1)}$*, and* $op \in \{stay, up\} \cup \{down_\nu \mid 1 \leq \nu \leq m\}$.

In practice, states q may differ in their *rank*, i.e., the numbers of their accumulating parameters. Let $X = \{x_1, x_2, \ldots\}$ denote a countable set of variables of rank (not necessarily 0), and assume that Σ, X and Y are disjoint. For a right-hand side ζ, we write also $\zeta = s[q_1(op_1), \ldots, q_c(op_c)]$ to refer to all occurrences of (maybe nested) state calls in the right-hand side. Here, $s \in \mathcal{T}_{\Sigma \cup X}(Y)$ is a tree which contains each variable $x_1, \ldots, x_c$ exactly once with $\zeta = s[q_1(op_1)/x_1, \ldots, q_c(op_c)/x_c]$ where $s[q_i(op)/x_i]$ denotes the substitution of the state call $q_i(op, s_1, \ldots, s_n)$ for the subtree $x_i(s_1, \ldots, s_n)$ in s where n is the rank of x_i and $n+1$ is the rank of q_i. Note that in s no state call occurs anymore. For example the right-hand side of the rule in Lines 2 and 3 in the Example 1.7 can be written as $s[q_{data}(down_1), q(down_1), q_{data}(down_1), q(down_2)]$ where $s = \texttt{employee}(\texttt{data}(x_1, y_b), x_2(\texttt{boss}(x_3, \texttt{e}), x_4(y_b, y_n)))$.

Intuitively, the meaning of the expressions of a right-hand side is as follows: The produced output is defined analogously to the output of a 2tt up to the accumulating parameters. Here, we consider *call-by-value* parameter passing only. Thus, the expression ζ_j in parameter position j is evaluated first; then the result (which is a tree without state calls) may be copied to the various uses of the formal parameter y_j. This evaluation strategy is also called *inside-out* (IO for short). Note that we slightly abuse Definition 1.10 and use accumulating parameters with names other than $y_1, y_2, \ldots$ (e.g. in Example 1.7 where we use y_b and y_n). Clearly this is without loss of generality, as parameters can easily be renamed according to the definition.

Example 1.8. The rules in the beginning of this section with the initial state q_I form a 2mtt $M_{y,\text{staff}}$. To transform the tree t'_B (the binary representation of our common example tree, see the left tree in Figure 1.2), the 2mtt starts at the root and applies the initial state. Thus, for the first step we get $\texttt{staff}(q(down_1, \texttt{e}, \texttt{e}), \texttt{e})$ where $down_1$ refers to the node 1. Applying the state q to this `employee`-node, we get several state calls. These state calls are partially nested. Figure 1.6 illustrates the two steps. In the upper picture is the first output with one state call. The lower figure shows the tree after processing $q(1, \texttt{e}, \texttt{e})$. There, we get the state call $q(down_1, \texttt{boss}(q_{data}(down_1), \texttt{e}), q(down_2, y_b, y_n))$ with nested calls. The first parameter accumulates a tree with root `boss`, whereas the second parameter is a further state call.

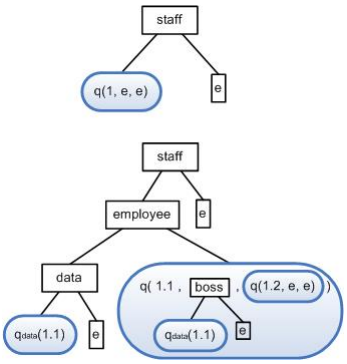


Fig. 1.6. Analyzing $q_I(\epsilon)$ and $q(1, e, e)$.

◁

The order in which nested state calls are evaluated indeed matters. Consider, e.g., a transducer with rules $q_I(\texttt{a}, 0) \to p(stay, q'(stay))$, $p(\texttt{a}, 0, y) \to \texttt{a}$, and $q'(\texttt{b}, 0) \to \texttt{b}$. If we evaluate the outermost calls first, the tree $t = \texttt{a}(t_1, \ldots, t_k)$ will be transformed into `a`. In this case, the accumulating parameter of p need not to be evaluated. If we start with the

innermost calls, there is no rule to evaluate the state call $q'(stay)$ in the right-hand side of the first line. Thus, the output is empty.

We specify the translation induced by a 2mtt using a denotational formulation. Later, we will also consider an operational semantics based on runs. In the denotational semantics, the meaning $[\![q]\!]_t$ of state q of M with n accumulating parameters (with respect to an input tree t) is defined as a mapping from nodes in the input tree to sets of trees with parameters in $Y_n = \{y_1, \ldots, y_n\}$, i.e., $[\![q]\!]_t : \mathcal{N}(t) \to 2^{\mathcal{T}_\Sigma(Y)}$. When we evaluate an innermost call $q(v, s_1, \ldots, s_n)$ during a computation, it suffices to substitute actual parameters s_j for the formal parameters y_j of all terms from $[\![q]\!]_t(v)$ to obtain the set of produced outputs. The values $[\![q]\!]_t$ for all q are jointly defined as the least mappings satisfying: $[\![q]\!]_t(v) \supseteq [\![\zeta]\!]_t$ for rule $q(\mathtt{a}, \eta, \bar{y}) \to \zeta$ where $\bar{y}$ denotes the sequence $y_1, \ldots, y_n$ and v is a node of t with $lab[v] = \mathtt{a}$, $\eta(v) = \eta$ and $[\![\zeta]\!]_t$ is defined by:

$$\begin{aligned} [\![y_j]\!]_t &= \{y_j\} \\ [\![\mathtt{b}(\zeta_1, \ldots, \zeta_m)]\!]_t &= \{\mathtt{b}(s_1, \ldots, s_m) \mid s_\nu \in [\![\zeta_\nu]\!]_t\} \\ [\![q'(op, \zeta_1, \ldots, \zeta_{n'})]\!]_t &= \{s[s_1/y_1, \ldots, s_{n'}/y_{n'}] \mid s \in [\![q']\!]_t([\![op]\!]_t(v)), s_\nu \in [\![\zeta_\nu]\!]_t\} \end{aligned}$$

Again, op stands for $down_\nu$, $stay$ or up. Recall that the meaning $[\![op]\!]$ is defined by $[\![stay]\!]_t(v) = v$, $[\![down_\nu]\!]_t(v) = v\nu$, and $[\![up]\!]_t(v\nu) = v$. Also, $s[s_1/y_1, \ldots, s_n/y_n]$ denotes the simultaneous substitution of the trees s_j for all occurrences of the variables y_j in the tree s. Note that the call-by-value semantics is reflected in the last equation: the same trees s_j are used for all occurrences of a variable y_j in the tree s corresponding to a potential evaluation of the state q'. The transformation τ_M realized by the 2mtt M on an input tree t and sets T of input trees, respectively, is, thus, defined by $\tau_M(t) = \bigcup\{[\![q_0]\!]_t(\epsilon) \mid q_0 \in Q_0\}$ and $\tau_M(T) = \bigcup\{\tau_M(t) \mid t \in T\}$.

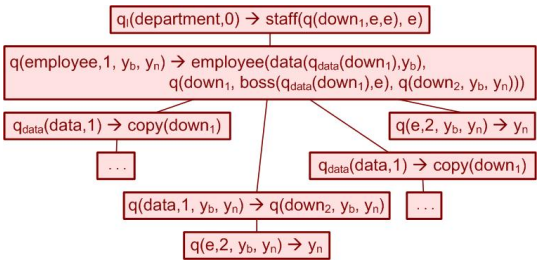


Fig. 1.7. A run of the 2mtt $M_{y,\text{staff}}$ on the tree t_e.

For the operational semantics runs of a 2mtt M on a tree t may be similarly defined as for a 2tt. It is a ranked tree over the set of rules. Here, the rank of a rule $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$ is given by the number of occurrences of recursive calls $q'(op)$ in ζ to states q' in Q. These calls may be nested. Figure 1.7 shows an accepting run ρ of the (deterministic) example 2mtt $M_{y,\text{staff}}$ on the tree t_e, which describes a department with one employee.

The denotational view on the semantics of a 2mtt allows us to use fixpoint arguments for proving the correctness of constructions, whereas the operational view is better suited for more combinatorial arguments. In particular, we can show that the number of occurrences of states in right-hand sides can be restricted to the maximum of the ranks of output symbols and states. We have:

Lemma 1.7. *For every 2mtt M there exists a 2mtt M' with (i) $\tau_{M'} = \tau_M$, (ii) the number of states occurring in each right-hand side is bounded by k, and (iii) $|M'| \in \mathcal{O}(|M| \cdot k^2)$,*

where k is the maximum of the ranks of output symbols and states of M.

Proof. The construction proceeds in two phases. In the first phase, we replace every *complicated* call $q'(op, \zeta_1, \ldots, \zeta_n)$ in the right-hand side of a rule $q(\mathtt{a}, \eta, y_1, \ldots, y_{n'}) \rightarrow \zeta$ by the simple call $\langle q, op, \zeta_1, \ldots, \zeta_n\rangle(stay, y_1, \ldots, y_{n'})$ for a new state $\langle q, op, \zeta_1, \ldots, \zeta_n\rangle$. Let $[\zeta]$ denote the resulting tree. For the new state $\langle q, op, \zeta_1, \ldots, \zeta_n\rangle$, we introduce the rule $\langle q, op, \zeta_1, \ldots, \zeta_n\rangle(\mathtt{a}, \eta, y_1, \ldots, y_{n'}) \rightarrow q(op, \langle\zeta_1, n'\rangle(stay, y_1, \ldots, y_{n'}), \ldots, \langle\zeta_n, n'\rangle(stay, y_1, \ldots, y_{n'}))$ for again fresh states $\langle\zeta_j, n'\rangle$ which are meant to produce the output of ζ_j using n' parameters. For these states, we introduce the rules: $\langle\zeta_j, n'\rangle(\mathtt{a}, \eta, y_1, \ldots, y_{n'}) \rightarrow [\zeta_j]$.

As a result of this first transformation phase, we achieve that all right-hand sides either are of the form $q(op, q_1(stay, y_1, \ldots, y_{n'}), \ldots, q_n(stay, y_1, \ldots, y_{n'}))$ or contain only non-nested calls, i.e., calls of the form $q(op, y_1, \ldots, y_{n'})$. In order to restrict the number of calls in right-hand sides of the second type, we essentially proceed as in the proof of Lemma 1.1, i.e., we introduce extra auxiliary states for every proper subtree of right-hand sides of the second kind which contain more than one call.

The resulting transducer has at most one fresh state for every node of a right-hand side while the total sum of sizes of right-hand sides may increase by a factor of k^2 in order to spell out all the auxiliary lists of parameters for the new states. □

1.6.1. *Type Checking Macro Tree Walking Transducers*

As for 2tts we now consider type checking for 2mtts. For a 2mtt M and a regular language $\mathcal{L}$, we again construct a transducer M' with $\tau_{M'}(t) = \tau_M(t) \cap \mathcal{L}$. As in Section 1.5.1, the language $\mathcal{L}$ consists of all erroneous outputs. In our application scenario of type checking, the language $\mathcal{L}$ is the complement of the output type which is either specified by a total dbta or by a dtta. Beyond the case of 2tts, we now additionally must deal with accumulating parameters. The transducer M' must keep track of the states of an automaton for $\mathcal{L}$ on the current values of the respective parameters. We start with a general construction for deterministic bottom-up automata.

Theorem 1.6. *For every 2mtt M and every dbta A, a 2mtt M_A can be constructed with*

$$\tau_{M_A}(t) = \tau_M(t) \cap \mathcal{L}(A)$$

for all $t \in \mathcal{T}_\Sigma$. The 2mtt M_A is of size $\mathcal{O}(|M| \cdot |A|^{l\cdot(d+1)})$ where l is the maximal rank of a state in M and d is the maximal number of occurrences of states in right-hand sides of M.

Proof. Let $M = (Q, \Sigma, R, Q_0)$ and $A = (P, \Sigma, \delta, F)$. For each state q in $Q^{(n)}$ and all states $p_0, \ldots, p_n \in P$, the 2mtt M_A has a state $\langle q, p_0 p_1 \ldots p_n\rangle$ which is meant to generate all trees s (possibly with variables from $\{y_1, \ldots, y_n\}$) which could be produced by M and for which additionally there is a run of A starting at the leaves y_j with state p_j and reaching the root of s in state p_0. The rules of M_A are: $\langle q, p_0 p_1 \ldots p_n\rangle(\mathtt{a}, \eta, y_1, \ldots, y_n) \rightarrow \zeta'$ for every rule $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \rightarrow \zeta$ of M and $\zeta' \in \tau^{p_0, p_1, \ldots, p_n}[\zeta]$ where the sets $\tau^{p_0, p_1, \ldots, p_n}[.]$ are inductively defined by:

$\tau^{p_j,p_1,\ldots,p_n}[y_j] = \{y_j\}$
$\tau^{p_0,p_1,\ldots,p_n}[\mathtt{b}(\zeta_1,\ldots,\zeta_m)] = \{\mathtt{b}(\zeta'_1,\ldots,\zeta'_m) \mid \exists p'_1,\ldots,p'_m \in P : (p_0,\mathtt{b},p'_1\ldots p'_m) \in \delta \wedge \forall j : \zeta'_j \in \tau^{p'_j p_1\ldots p_n}[\zeta_j]\}$
$\tau^{p_0,p_1,\ldots,p_n}[q'(op,\zeta_1,\ldots,\zeta_{n'})] = \{\langle q', p_0 p'_1\ldots p'_{n'}\rangle(op,\zeta'_1,\ldots,\zeta'_{n'}) \mid \exists \zeta'_1,\ldots,\zeta'_m \in Z : \forall j : \zeta'_j \in \tau^{p'_j p_1\ldots p_n}[\zeta_j]\}$

where Z denotes the set of all subterms of possible right-hand sides of rules of M_A. The set of initial states of M_A is given by $Q'_0 = Q_0 \times F$. By fixpoint induction, we verify for every state q of rank $n+1$, every input tree $t \in \mathcal{T}_\Sigma$, every node $v \in \mathcal{N}(t)$ and states $p_0,\ldots,p_n$ of A that:

$$[\![\langle q,p_0,\ldots,p_n\rangle]\!]_t(v) = [\![q]\!]_t(v) \cap \{s \in \mathcal{T}_\Sigma(Y) \mid \delta^*(s,p_1\ldots p_n) = p_0\} \qquad (*)$$

where $Y = \{y_1,\ldots,y_n\}$ and δ^* is the extension of the transition function of A to trees containing variables from Y, namely, for $\underline{p} = p_1\ldots p_n$, $\delta^*(y_j,\underline{p}) = p_j$, and $\delta^*(\mathtt{a}(t_1,\ldots,t_m),\underline{p}) = \delta(\mathtt{a},\delta^*(t_1,\underline{p})\ldots\delta^*(t_m,\underline{p}))$. The correctness of the construction follows from $(*)$. For each state in M we have at most $|A|^{l+1}$ new states in M_A where l is the maximal rank of states in M. Assume that d is the maximal number of occurrences of states in right-hand sides of rules of M. Then each rule of M gives rise to at most $|A|^{(l+1)\cdot(d+1)}$ new rules in M_A. Therefore, the new 2mtt is of size $\mathcal{O}(|M|\cdot|A|^{(l+1)\cdot(d+1)})$. □

Lemma 1.2 provides a bta describing the complement of a dtta — which, however, is not necessarily deterministic. Theorem 1.6, on the other hand, only holds for deterministic btas. A similar construction is also possible if the bta A is nondeterministic — but then only for transducers which are output-*linear*. Here, we call a 2mtt output-linear if every accumulating parameter occurs at most once in a right-hand side. Nonetheless, we are able to handle complements of output types described by dttas directly. For that, however, we introduce a dedicated construction of an 2mtt M_A.

Theorem 1.7. *For every 2mtt M and every dtta A, a 2mtt $M_{\bar{A}}$ can be constructed with*

$$\tau_{M_{\bar{A}}}(t) = \tau_M(t) \cap \overline{\mathcal{L}(A)}$$

for all $t \in \mathcal{T}_\Sigma$. The 2mtt $M_{\bar{A}}$ is of size $\mathcal{O}(|M|\cdot(h\cdot|A|)^{d+2})$ where $h+1$ is the maximum of the maximal rank of a state in M and the maximal rank of an output symbol, and d is the maximal number of occurrences of state calls in right-hand sides of M.

Proof. Let $M = (Q,\Sigma,R,Q_0)$ and $A = (P,\Sigma,\delta,p_0)$. Our goal is to construct a 2mtt $M_{\bar{A}}$ which simulates the behavior of M while at the same time guessing a path in the output tree which proves non-containment in the set $\mathcal{L}(A)$. For that, the set Q' is defined as: $Q' = Q \cup \{\langle q,p\rangle \mid q \in Q, p \in P\} \cup \{\langle q,p,j,p'\rangle \mid q \in Q^{(n)}, p,p' \in P, j \in \{1,\ldots,n\}\}$. Here, a state $q \in Q$ of $M_{\bar{A}}$ behaves like the state q of M. States $\langle q,p\rangle$ or $\langle q,p,j,p'\rangle$ behave like q in M but additionally make sure that there is a path in the generated output starting from a state p of A at the root which verifies that there is no p-run of A on the output. Thereby, a state $\langle q,p\rangle$ will directly generate the end point of such a path whereas state $\langle q,p,j,p'\rangle$ will only generate a path with p at the root reaching a parameter y_j with state p'. Accordingly, the 2mtt $M_{\bar{A}}$ has the following rules:

$$q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$$
$$\langle q, p\rangle(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta' \text{ with } \zeta' \in [\zeta]^p$$
$$\langle q, p, j, p'\rangle(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta' \text{ with } \zeta' \in [\zeta]^{p,j,p'}$$

for every rule $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$ of M. The sets $[.]^x$ are inductively defined by:

$$\begin{aligned}
[y_i]^p &= \emptyset\\
[\mathtt{b}(\zeta_1, \ldots, \zeta_m)]^p &= \{\mathtt{b}(\zeta_1, \ldots, \zeta_m) \mid \forall \bar{p} \in P^m : \ (p, \mathtt{b}, \bar{p}) \notin \delta\}\\
&\quad \cup \{\mathtt{b}(\zeta_1, \ldots, \zeta_{\nu-1}, \zeta'_\nu, \zeta_{\nu+1}, \ldots, \zeta_m) \mid \nu \geq 1, (p, \mathtt{b}, p_1 \ldots p_m) \in \delta,\ \zeta'_\nu \in [\zeta_\nu]^{p_\nu}\}\\
[q(op, \zeta_1, \ldots, \zeta_n)]^p &= \{\langle q, p, \nu, p_\nu\rangle(op, \zeta_1, \ldots, \zeta_{\nu-1}, \zeta'_\nu, \zeta_{\nu+1}, \ldots, \zeta_n) \mid \zeta'_\nu \in [\zeta_\nu]^{p_\nu}\}\\
&\quad \cup \ \{\langle q, p\rangle(op, \zeta_1, \ldots, \zeta_n)\}
\end{aligned}$$

$$\begin{aligned}
[y_j]^{p,j,p'} &= \{y_j \mid p = p'\}\\
[\mathtt{b}(\zeta_1, \ldots, \zeta_m)]^{p,j,p'} &= \{\mathtt{b}(\zeta_1, \ldots, \zeta_{\nu-1}, \zeta'_\nu, \zeta_{\nu+1}, \ldots, \zeta_m) \mid \nu \geq 1, (p, \mathtt{b}, p_1 \ldots p_m) \in \delta,\\
&\qquad\qquad \zeta'_\nu \in [\zeta_\nu]^{p_\nu, j, p'}\}\\
[q(op, \zeta_1, \ldots, \zeta_n)]^{p,j,p'} &= \{\langle q, p, \nu, p_\nu\rangle(op, \zeta_1, \ldots, \zeta_{\nu-1}, \zeta'_\nu, \zeta_{\nu+1}, \ldots, \zeta_n) \mid \zeta'_\nu \in [\zeta_\nu]^{p_\nu, j, p'}\}
\end{aligned}$$

First, we verify that for every tree $s \in \mathcal{T}_\Sigma(Y)$ the sets $[s]^p$ and $[s]^{p,j',p'}$ either are empty or equal $\{s\}$ where following holds:

(1) $[s]^p = \{s\}$ iff s contains a node $v = i_1 \ldots i_r$ such that there are transitions $(p_0^{(j)}, \mathtt{a}_j, p_1^{(j)}, \ldots, p_{m_j}^{(j)}) \in \delta$ for $j = 1, \ldots, r-1$, such that
   (a) The label of the node $i_1 \ldots i_j$ equals $\mathtt{a}_j$;
   (b) $p_0^{(1)} = p$ and for $j = 1, \ldots, r-2$, $p_0^{(j+1)} = p_{i_j}^{(j)}$;
   (c) there is no $p_{i_r}^{(r-1)}$-transition of A for $\mathtt{a}_r$.

(2) $s \in [s]^{p,j',p'}$ iff s contains a node $v = i_1 \ldots i_{r+1}$ which is labeled with $y_{j'}$ and there are transitions $(p_0^{(j)}, \mathtt{a}_j, p_1^{(j)}, \ldots, p_{m_j}^{(j)}) \in \delta$ for $j = 1, \ldots, r$, such that:
   (a) For $j = 1, \ldots, r$, the label of the node $i_1 \ldots i_j$ equals $\mathtt{a}_j$;
   (b) $p_0^{(1)} = p$ and $p_{i_{r+1}}^{(r)} = p'$; and for $j = 1, \ldots, r-1$, $p_0^{(j+1)} = p_{i_j}^{(j)}$.

Note in particular that by this definition, $s \notin \mathcal{L}(A)$ iff $[s]^{p_0} = \{s\}$ for the initial state p_0 of A. Let us extend the operators $[\,.\,]^p$ and $[\,.\,]^{p,j',p'}$ by:

$$[S]^p = \bigcup\{[s]^p \mid s \in S\} \qquad [S]^{p,j',p'} = \bigcup\{[s]^{p,j',p'} \mid s \in S\}$$

for $S \subseteq \mathcal{T}_\Sigma(Y)$ with $Y = \{y_1, \ldots, y_n\}$. By fixpoint induction, we verify for every state q of rank $n+1$, every $j \in \{1, \ldots, n\}$, every input tree $t \in \mathcal{T}_\Sigma$ and states $p, p' \in P$ that:

$$[\![\langle q, p, j, p'\rangle]\!](t) = [\,[\![q]\!](t)\,]^{p,j,p'}$$
$$[\![\langle q, p\rangle]\!](t) = [\,[\![q]\!](t)\,]^p$$

For each state p and right-hand side ζ of a rule, we assign states of the dtta to the nodes of the tree. Either we immediately hit a node certifying the non-existence of a p-run of the dtta on the output generated from ζ, or we hit an occurrence of a state call $q(op, \ldots)$. If $n \geq 1$ is the rank of q, we have n choices here: either we expect a certificate for the failure of the dtta A inside the evaluation of q or in one of the parameters of q. Overall, we find that every

rule of M, thus, gives rise to $(h \cdot |A|)^{d+2}$ rules. Thus, we have $|M_{\bar{A}}| \in \mathcal{O}(|M| \cdot (h \cdot |A|)^{d+2})$. □

Assume that we have a binary ranked alphabet and that at least one state has an accumulating parameter, i.e., the maximal rank l of states is at least 2. By Lemma 1.7 it is then possible to restrict the number of occurrences of state calls in right-hand sides of the 2mtt to l. This implies that the size of the intersection 2mtt M_A in Theorem 1.6 for a dbta describing the output language is in $\mathcal{O}(|M| \cdot |A|^{l \cdot (l+1)})$. Furthermore, the size of the 2mtt M_A in Theorem 1.7 for a deterministic top-down tree automaton describing the output language is in $\mathcal{O}(|M| \cdot (l \cdot |A|)^{l+1})$.

1.6.2. *Deciding Emptiness of 2MTTs*

To decide emptiness of a 2mtt M, we follow the approach taken for 2tts: we construct an alternating tree walking automaton A_M which is then tested for emptiness. The atwa A_M has the same set of states as M (but they are not ranked anymore now), where the initial states of M and A_M coincide. For every rule $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$ of M, the atwa A_M has a rule $q(\mathtt{a}, \eta) \to q_1(op_1) \wedge \ldots \wedge q_c(op_c)$ if $q_1(op_1, \ldots), \ldots, q_c(op_c, \ldots)$ is the sequence of calls to states of M (possibly nested inside each other), in any order. Since we use 2mtts with call-by-value semantics, M has an accepting run on some input tree t iff A_M has an accepting run on t. Note that this construction is wrong for call-by-need semantics, because M could have an accepting run on a tree $t \notin \mathcal{L}(A_M)$; for instance the 2mtt with the rules $q_I(\mathtt{a}, 0) \to q(stay, q'(stay)), \quad q(\mathtt{a}, 0, y) \to \mathtt{a}$ on the tree $\mathtt{a}$.

Theorem 1.8. *For every 2mtt M an atwa A_M can be constructed in polynomial time such that $\mathcal{L}(A_M) = dom(M)$. Thus, it can be decided in deterministic exponential time whether the translation of a 2mtt is empty or not.*

1.6.3. *Input-Linear 2MTTs*

The notions of b-boundedness and strict b-boundedness which we have defined for 2tts stay meaningful also in presence of accumulating parameters. Analogously, we find that emptiness for b-bounded transducers is decidable in polynomial time — independent on the number of accumulating parameters of states.

In order to identify classes of 2mtts where full type-checking is tractable, we therefore take a closer look at the construction for the intersection of 2mtts with (complements of) output types. For simplicity, we first consider 2mtts which are *strictly 1-bounded*. This notion is only meaningful for *top-down* mtts, i.e., 2mtts without operations up or $stay$. A top-down transducer M is guaranteed to visit each node of the input tree at most once, if for the same i, the operation $down_i$ does not occur twice in the same right-hand side of M. This property can easily be checked syntactically. Tree transducers satisfying this restriction are called *input-linear*.

Note that input-linearity for a tree transducer implies that the number of state calls in right-hand sides is bounded by the maximal rank of input symbols. Moreover, the output

language can be described by rules that are obtained by simply deleting all directives from the transducer's rules. The resulting rules no longer specify a transformation but constitute a *context-free tree grammar* (short: cftg) for generating output trees. As an example of an input-linear mtt consider the following mtt which produces the same output as the common example $M_{y,\text{staff}}$ without the `boss`-subtrees. The transducer only needs one parameter (for the next employee) and has a new state q_{empl}. For $\eta \in \{1, 2\}$, it has the rules:

$$
\begin{array}{ll}
1 & q_I(\texttt{department}, 0) \quad \to \texttt{staff}(q(down_1, \texttt{e}), \texttt{e}) \\
2 & q(\texttt{employee}, \eta, y_n) \quad \to q_{empl}(down_1, q(down_2, y_n)) \\
3 & q_{empl}(\texttt{data}, 1, y_n) \quad \to \texttt{employee}(\texttt{data}(q_{data}(down_1), \texttt{e}), q(down_2, y_n)) \\
4 & q(\texttt{subordinates}, 2, y_n) \to q(down_1, y_n) \\
5 & q(\texttt{e}, \eta, y_n) \quad \to y_n \\
6 & q_{data}(\texttt{data}, 1) \quad \to \mathit{copy}(down_1).
\end{array}
$$

The grammar characterizing its output language looks as follows:

$$
\begin{array}{ll}
1 & q_I \quad \to \texttt{staff}(q(\texttt{e}), \texttt{e}) \\
2 & q(y_n) \quad \to q_{empl}(q(y_n)) \mid q(y_n) \mid y_n \\
3 & q_{empl}(y_n) \to \texttt{employee}(\texttt{data}(q_{data}, \texttt{e}), q(y_n)) \\
4 & q_{data} \quad \to \mathit{copy}
\end{array}
$$

where $q_I, q, q_{empl}, q_{data}, \mathit{copy}$ are nonterminals. Selection of rules depending on input symbols and directions now has been replaced with nondeterministic choice.

Context-free tree grammars generalize context-free grammars to trees. Formally, a cftg G can be represented by a tuple (E, Σ, P, E_0) where E is a finite ranked set of function symbols or nonterminals, $E_0 \subseteq E$ is a set of initial symbols of rank 0, Σ is the ranked alphabet of terminal nodes and P is a set of rules of the form $q(y_1, \ldots, y_n) \to \zeta$ where $q \in E$ is a nonterminal of rank $n \geq 0$. The right-hand side ζ is a tree built up from variables $y_1, \ldots, y_n$ by means of application of nonterminal and terminal symbols. As for 2mtts, inside-out (IO) and outside-in evaluation order for nonterminal symbols must be distinguished. Here, we use the IO or call-by-value evaluation order. The least fixpoint semantics for the cftg G is obtained straightforwardly along the lines for 2mtts — simply by removing the corresponding directive components, i.e., by removing in the last line of the definition of $[\![\zeta]\!]$ for 2mtts the op and $[\![op]\!]_t(v)$. In particular, this semantics assigns to every nonterminal q of rank $n \geq 0$, a set $[\![q]\!] \subseteq \mathcal{T}_\Sigma(Y)$ for $Y = \{y_1, \ldots, y_n\}$. The language generated by G is $\mathcal{L}(G) = \bigcup\{[\![q_0]\!] \mid q_0 \in E_0\}$.

It is easy to see that the output language of an input-linear mtt M can be characterized by a cftg G_M which can be constructed from M in linear time. During this construction every rule $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$ is rewritten as a production $q(y_1, \ldots, y_n) \to \zeta'$, where ζ' is obtained from ζ by deleting all occurrences of navigation operators.

The characterization of output languages for input-linear mtts by cftgs is useful because emptiness for (IO-)cftgs is decidable using a similar algorithm as the one for ordinary context-free (word) grammars, and hence can be done in linear time.

Theorem 1.9. *It can be decided in linear time for a cftg G whether or not $\mathcal{L}(G) = \emptyset$.*

Here, we are interested in testing whether a given input-linear mtt M type checks w.r.t. input and output types I and O. Assume that I is given by a (possibly non-deterministic) bta B with only productive states, i.e., for every state p of B there exists a p-run of B on a tree. As a first step, we construct a new input-linear mtt M_B such that M_B's range, i.e., the set $\tau_{M_B}(\mathcal{T}_\Sigma)$ is equal to $\tau_M(I)$. This is done by a straightforward product construction of the bta B and the input-linear mtt M. Note that it may happen that M does not visit a certain subtree t of the input tree. In such a case the checking of t w.r.t. B cannot be done by the new transducer M_B. This does not affect the corresponding output language though. We now construct the intersection 2mtt for M_B and the complement of the output type O. In case, O is given by a dbta, this can be done along the lines of the proof of Theorem 1.6. If O is given by a dtta, we rely on the construction from Theorem 1.7. Since M is input-linear, the intersection 2mtt is again input-linear — meaning that its range can be described by a cftg (thus generating all "illegal outputs" of M w.r.t. I and O). Therefore, Theorem 1.9 gives us:

Theorem 1.10. *Assume M is an input-linear mtt where the ranks of input symbols are bounded, and let I and O denote input and output types for M where I is given by a bta.*

(1) Assume that the output type O is specified with a dbta and the maximal rank of states of M is bounded. Then M can be type checked relative to I and O in polynomial time.
(2) Assume that the output type O is specified with a dtta. Then M can be type checked relative to I and O in polynomial time — even in presence of unbounded ranks of states.

The worst-case complexity bounds for the construction of Theorem 1.10 are exponential in $l{\cdot}(k{+}1)$ (for output types given through dbtas) or $k{+}2$ (for output types given through dttas) where l is the maximal rank of states and k is the maximal rank of an input symbol of M. In practical applications, both k and l may be moderately small. Still, we want to point out that in case of input-linear mtts, the intersection construction can be organized in such a way that only "useful" states are constructed. In order to see this, consider again an input-linear mtt M and a dbta A (representing the incorrect output trees). The idea is to introduce for every q of M of rank $n+1$, a *Datalog* predicate $q/n+1$. Every rule $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$ of M then gives rise to the *Datalog* implication: $q(Y_0, \ldots, Y_n) \Leftarrow \mathcal{D}[\zeta]_{Y_0}$ where $\mathcal{D}[\zeta]_X$ (X is a variable) is defined by

$$\begin{array}{ll} \mathcal{D}[y_j]_X & = X = Y_j \\ \mathcal{D}[\mathtt{b}(\zeta_1, \ldots, \zeta_m)]_X & = \delta(X, \mathtt{b}, X_1 \ldots X_m) \ \wedge \ \mathcal{D}[\zeta_1]_{X_1} \wedge \ldots \wedge \mathcal{D}[\zeta_m]_{X_m} \\ \mathcal{D}[q'(\zeta_1, \ldots, \zeta_m)]_X & = q'(X, X_1, \ldots, X_m) \ \wedge \ \mathcal{D}[\zeta_1]_{X_1} \wedge \ldots \wedge \mathcal{D}[\zeta_m]_{X_m} \end{array}$$

and the variables $X_1, \ldots, X_m$ in the last two rows are fresh. For subsets $X, X_1, \ldots, X_{m'}$ of the set of states of A, $\delta(X, \mathtt{a}, X_1 \ldots X_{m'})$ denotes the fact that $(x, \mathtt{a}, x_1 \ldots x_{m'}) \in \delta_A$ for all $x \in X$ and $x_j \in X_j$, $j = 1, \ldots, m'$. A bottom-up evaluation of the resulting *Datalog* program computes for every $q/(n+1)$, the set of all tuples $(p_0, \ldots, p_n)$ such that the translation of $\langle q, p_0 \ldots p_n \rangle$ is non-empty. If we additionally want to restrict these predicates only to tuples which may contribute to a terminal derivation of initial nonterminals $\langle q_0, p_f \rangle$,

we may top-down query the program with queries $\Leftarrow q_0(p_f)$. Practically, top-down solving organizes the construction such that only useful nonterminals of the intersection grammar are considered. Using this approach, the number of newly constructed nonterminals often will be much smaller than the bounds stated in the theorem. A similar construction is also possible for the intersection of mtts with the complements of dtta languages.

The algorithm for *input-linear* mtts can also be applied to *non-input-linear* 2mtts. Then, the constructed *Datalog* program does no longer precisely characterize the non-empty functions of the intersection 2mtt because dependencies on input subtrees (viz. several transformations of the same input node) have been lost. Accordingly, a *superset* is returned. By means of cftgs, we can express this observation as follows:

Theorem 1.11. *Let G_M be the cftg constructed for a 2mtt M. Then $\tau_M(\mathcal{T}_\Sigma) \subseteq \mathcal{L}(G_M)$.*

Since the cftg still provides a safe *superset* of produced outputs, type checking based on cftgs is sound in the sense that if it does not flag an error, the transformation also will not go wrong. On the other hand, a flagged error may be possibly spurious, i.e., due to the over-approximation of the output language through the cftg.

Consider a top-down transducer M with rule $q_0(\mathtt{a}, 0) \to \mathtt{c}(p(down_1), p(down_1))$, where p realizes the identity using the rules $p(\mathtt{a}, \eta) \to \mathtt{a}(p(down_1))$, $p(\mathtt{b}, \eta) \to \mathtt{b}(p(down_1))$, $p(\mathtt{e}, \eta) \to \mathtt{e}$. In this case, the corresponding approximating cftg G_M is rather coarse: it generates $\mathtt{c}(u, v)$ with $u, v \in \{\mathtt{a}, \mathtt{b}\}^*\mathtt{e}$ (seen as monadic trees). Exact tree copying, however, can be realized through the use of parameters: the transducer with rules $q_0(\mathtt{a}, 0) \to q(down_1, p(down_1))$ and $q(\sigma, \eta, y_1) \to \mathtt{c}(y_1, y_1)$ (for all $\sigma \in \Sigma$) and the same p-rules as M will realize the same translation as M. The cftg for the resulting transducer now does not provide an over-approximation but precisely captures the output language of M.

When approximating the output languages of general 2mtts with cftgs, we no longer can assume that the maximal number d of occurrences of nonterminals in a right-hand side of this grammar is bounded by a small constant. If d turns out to be unacceptably large, we still can apply Lemma 1.7 to limit the maximal number of occurrences of nonterminals in each right-hand side to a number k which is the maximal rank of output symbols and states. This construction, however, introduces $stay$-moves and thus destroys input-linearity.

1.6.4. *Notes and References*

Macro tree transducers [EV85] are a combination of top-down tree transducers and macro grammars [Fis68]. Macro grammars are just like context-free tree grammars (cftgs), but produce strings (a cftg can be seen as a special macro grammar, because terms are particular strings). Fischer already distinguishes IO and OI for macro grammars, and proves that the corresponding classes of languages are incomparable (which also holds in the tree case). Our normal form of Lemma 1.7 can be seen as a variant of Fischer's IO standard form, which in fact is very similar to Chomsky normal form of context-free grammars: there are exactly 2 or 0 nonterminals in every right-hand side of a grammar in IO standard form. A similar normal form might be possible for 2mtts too, but will cause a larger size increase of the transducer (note that Fischer does not report on grammar sizes, in his constructions).

Fischer remarks, just after Corollary 3.1.6, that emptiness of IO macro languages can be reduced to emptiness of context-free languages, by simply dropping all parentheses, commas, argument, and terminal symbols. The resulting context-free (word) grammar generates the empty word if and only if the original language is empty. Since emptiness for a context-free grammar can be decided in linear time (see, e.g., [HMU01]), we obtain a linear time procedure for checking emptiness of IO context-free tree languages, as stated in Theorem 1.9. Context-free tree grammars were considered in [Rou70] and extensively studied in [ES77].

The fact that output languages of input-linear mtts are IO context-free tree languages is mentioned in Corollary 5.7 of [EV85] (the class of input-linear mtts is called LMT_{IO} there). A 2mtt without up-moves is called "stay-mtt". Results similar as the ones obtained in this section for 2mtts, have been obtained already in [MPS07] for the restricted case of stay-mtts. For instance, Proposition 3 of that paper is similar to our Theorem 1.7, Theorem 2 of [MPS07] corresponds to our Theorem 1.6, and Theorem 5 of that paper corresponds to our Theorem 1.10. Just before Theorem 1.10, we describe how to incorporate an input type into an input-linear transducer, so that the corresponding output language is preserved. The technical details are exactly as in the proof of Theorem 3.2.1 of [ERS80], where this results was proved for top-down tree transducers. 2mtts are essentially the same as the k-pebble macro tree transducers (k-pmtt) of [EM03a] for the case $k = 0$. For k-pmtts, a normal form similar to our Lemma 1.7 was shown in Theorem 16 of [EM03a].

1.7. Macro Forest Walking Transducers

Conceptually, XML documents are not trees, but forests. Therefore, we extend the concept of tree walking transducers (without or with parameters) to a transformation formalisms of forests. Forests are introduced in Definition 1.1.

Example 1.9. We consider again a transformation from company structures (cf. Section 1.2) to collections of employees which are listed under a new root node labeled `staff` (cf. Section 1.3). In contrast to tree walking transducers, forest transducers do not depend on a ranked alphabet. They can deal with arbitrary many subtrees of nodes. Here, we define a transformation which returns trees of the form $\texttt{staff}\langle f\rangle$ where f is a forest composed of trees of the form $\texttt{employee}(\texttt{data}(\ldots), \texttt{boss}(\ldots))$. The input trees are described by the DTD in Section 1.4. Additionally to the operations up, $stay$ as in tree walking transducers, forest transducers may use a directive $down$ for proceeding to the first child as well as directives $left$ and $right$ for proceeding to the left or right sibling, respectively.

$$
\begin{array}{lll}
1 & q_I(\texttt{department}, 0) & \to \texttt{staff}\langle q(down, \epsilon)\rangle \\
2 & q(\texttt{employee}, \eta, y_b) & \to \texttt{employee}\langle \texttt{data}\langle q_{data}(down)\rangle\ y_b\rangle \\
3 & & \quad q(down, \texttt{boss}\langle q_{data}(down)\rangle) \\
4 & & \quad q(right, y_b) \\
5 & q(\texttt{data}, 2, y_b) & \to q(right, y_b) \\
6 & q(\texttt{subordinates}, 3, y_b) & \to q(down, y_b) \\
7 & q(\epsilon, \eta, y_b) & \to \epsilon \\
8 & q_{data}(\texttt{data}, 2) & \to copy(down)
\end{array}
$$

where state *copy* in line 8 is meant to copy the forest f of a subtree $\mathtt{data}\langle f\rangle$. The right-hand side of the second rule is a composition of three forests (line 2-4). The initial state is q_I, which means that we start with state q_I at the root of the first tree of a forest. Here, the transducer walks only the first tree of an input forest. Note also that now rules may be selected depending on the current label of a node in the forest together with its forest direction. Thus, it can check whether the current node is the leftmost node on the top-level (value 0), is on the top-level, but not leftmost (value 1), is leftmost but not on the top-level (value 2) or is neither leftmost nor on the top-level (value 3). ◁

Definition 1.11 (2mft). *A* macro forest walking transducer *(*2mft *for short) is a tuple* $M = (Q, \Sigma, Q_0, R)$*, where* Q *is a finite ranked set of states,* Σ *is a finite alphabet with* $Q \cap \Sigma = \emptyset$*,* $Q_0 \subseteq Q^{(1)}$ *is the set of initial states and* R *is a finite set of rules of the form:* $q(\epsilon, \eta, y_1, \ldots, y_n) \to \zeta$ *or* $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$ *with* $\mathtt{a} \in \Sigma$*, direction* $\eta \in \{0, \ldots, 3\}$ *and* $q \in Q^{(n+1)}$ *where the right-hand sides are forests* ζ *of the following form:* $\zeta ::= \epsilon \mid y_j \mid q(op, \zeta_1, \ldots, \zeta_{n'}) \mid \mathtt{b}\langle\zeta_1\rangle \mid \zeta_1\,\zeta_2$ *where* $q \in Q^{(n'+1)}$*,* $\mathtt{b} \in \Sigma$*,* $op \in \{up, stay, down, left, right\}$ *and* $j = 1, \ldots, n$*. Moreover, the right-hand sides for empty input forests* ϵ *must not contain occurrences of the operations* $down, right$ *or* $left$*.*

In case of several rules for the same q, the same direction η and the same symbol a (or ϵ), we also write: $\zeta_1 \mid \ldots \mid \zeta_k$ to list all occurring right-hand sides. In case, that no operation up is used, the 2mft is also called *top-down* (short: *1mft* or *mft*). Likewise, if all states are of rank 1, i.e., have no accumulating parameters, the 2mft is an (ordinary) *forest walking* transducer (short: *2ft*). Finally, a 1mft without parameters is also called *forest transducer* (short: *1ft* or *ft*). As for macro tree walking transducers, in practice, states q may differ in their *ranks*, i.e., the numbers of their accumulating parameters plus 1. The set R of rules in Example 1.9 constitute the transducer $M_{y,\text{staff},f} = (Q, \Sigma, Q_0, R)$ with $Q = \{q_I, q, q_{data}, copy\}$ and $Q_0 = \{q_I\}$, which happens to be a 1mft.

A forest transducer behaves similar to a corresponding tree transducer: while walking over the input forest, the transducer chooses rules corresponding to the current states, input symbols and directions at the respective current nodes in the input, and then evaluates the right-hand sides of the rules. Again, we just consider the *inside-out* (IO or call-by-value) strategy for evaluating parameters. There are two significant differences between forest walking and tree walking transducers: First, a forest walking transducer produces output *forests* and therefore, as an extra operation, also supports *concatenation* of output forests. Secondly, the forest transducer has a different set of directions as well as a different set of navigational directives: up now means that the transducer moves to its ancestor in the input forest. $left$ and $right$ now means that the transducer moves to its left or right sibling, whereas $down$ means that the transducer moves to its first child. Formally, the semantics of these operations is defined by $[\![up]\!](vi) = v$, $[\![left]\!](vi) = v(i-1)$ if $i > 0$, $[\![right]\!](vi) = v(i+1)$, and $[\![down]\!](v) = v0$. Note that only the operations $down$ and $right$ have immediate equivalents in commands of a transducer on the *first-child next-sibling* encoding of forests as binary trees where they correspond to the commands $down_1$ and $down_2$, respectively. The up-command on the tree, on the other hand, may correspond to

the forest commands *left* or *up* — depending on whether the current node is a right or left child. The class of all (forest-walking) macro forest transformations is denoted by (2FMAC) FMAC. Analogously to Lemma 1.7, we find:

Lemma 1.8. *For every 2mft M there exists a 2mft M' with (i) $\tau_{M'} = \tau_M$ (ii) there are at most l occurrences of states on a right-hand side of rules in M' and (iii) $|M'| \in \mathcal{O}(|M| \cdot l^2)$, where l is the maximum of 2 and the maximal rank of states of M.*

Note that it is unfortunately *not* possible to simulate 2mfts by 2mtts which work on (possibly enriched) first-child next-sibling encodings of input and output trees. To see this, consider first the 1mft case. As shown in [PS04], one can easily construct a 1mft which takes as input a binary tree with m nodes, and outputs a forest consisting of 2^m leaves, i.e., a string of length 2^m. Consider the height increase of the corresponding translation on encoded trees: it is *double*-exponential. However, the height-increase of mtts is at most exponential (see [EV85]). Now consider the 2mft case. Clearly, a 2mtt can translate a binary tree with m nodes into a monadic tree with 2^m nodes, by doing a depth-first left-to-right traversal, and at each step generating a duplicated state call in a parameter position. As intermediate sentential form the transducer generates $q(\varepsilon, q(\varepsilon, \dots, q(\varepsilon, \mathtt{e})) \dots)$ which has 2^m-many occurrences of q; it then replaces $q(\varepsilon, t)$ by $\mathtt{g}(t)$, where g is an output symbol of rank 1. A 2mft can generate the same sentential form, but can replace $q(\varepsilon, t)$ by tt, i.e., the forest of concatenating two copies of t. In this way, a forest consisting of 2^{2^m}-many e's is generated. On first-child next-sibling encodings, this corresponds to a tree of height 2^{2^m}. Thus, the translation on encodings has *double*-exponential size-to-height increase. However, it is not difficult to see that the size-to-height increase of 2mtts is at most exponential.

1.7.1. *Intersecting Forest Walking Transducers with Output Types*

In this section, we consider general techniques for intersecting forest walking transducers with output types. Assume that we are given a regular forest language $\mathcal{L}$. Our goal is to construct for a given 2mft M, another 2mft M' which behaves similar to M but produces only outputs in $\mathcal{L}$. If $\mathcal{L}$ describes the set of all invalid outputs, type-checking for M, thus, reduces to checking emptiness of the transformation M'.

In order to provide a general construction for regular $\mathcal{L}$, let us first assume that $\mathcal{L}$ is given as the language defined by a finite forest monoid A, i.e., $\mathcal{L} = \mathcal{L}(A)$. A finite forest monoid (short: ffm) can be considered as a deterministic bottom-up automaton which combines the individual states for the trees t_i in a forest $f = t_1 \dots t_m$ by means of a monoid operation $\circ$ (compare, e.g., the discussion in [BW05]). Formally, a *finite forest monoid* consists of a finite monoid G with a neutral element e, a finite subset $F \subseteq G$ of accepting elements, together with a function $up : \Sigma \times G \to G$ mapping a symbol of Σ together with a monoid element for its content to a monoid element representing a forest of length 1. A finite forest monoid accepts a forest f if $up^*(f) \in F$ where $up^*(f_1 f_2) = up^*(f_1) \circ up^*(f_2)$, $up^*(\mathtt{a}\langle f' \rangle) = up(\mathtt{a}, up^*(f'))$, and $up^*(\epsilon) = e$. Given a total deterministic bottom-up forest automaton $A = (P, \Sigma, \delta, F_A)$, i.e., a dbta operating on the first-child next-sibling representation of forests, we construct a finite forest monoid as follows. Let G

be the monoid of functions $P \to P$ where the monoid operation is function composition. In particular, the neutral element of this monoid is the identity function. Moreover, the function up is defined by $up(\mathtt{a}, g) = p \mapsto \delta(\mathtt{a}, g(\delta(\mathtt{e}))\, p)$. Finally, the set of accepting elements is given by $F = \{g \in G \mid g(\delta(\mathtt{e})) \in F_A\}$.

On the other hand every forest monoid G gives rise to a finite tree automaton A_G (running on first-child next-sibling representations) whose set of states is given by the elements of M. The transition function δ of A_G is defined by: $\delta(\mathtt{e}) = e$ and $\delta(\mathtt{a}, g_1\ g_2) = up(\mathtt{a}, g_1) \circ g_2$. Then the set of accepting states simply is given by the accepting elements of G. These constructions show that every recognizable forest language can be recognized by a finite forest monoid and vice versa. Although the ffm for a bottom-up tree automaton generally can be exponentially larger, this need not always be the case.

Example 1.10. For our running example the bta in Example 1.3 is not a total deterministic bottom-up forest automaton. We get a total dbfa by adding an extra error state $\bullet$. The new transition function δ' then is defined by: $\delta'(\mathtt{staff}, r_{\text{empl}}\ r_{\mathtt{e}}) = \delta'(\mathtt{staff}, r_{\mathtt{e}}\ r_{\mathtt{e}}) = r_{\text{staff}}$, $\delta'(\mathtt{employee}, r_{\text{data}}\ \ r_{\text{empl}}) = \delta'(\mathtt{employee}, r_{\text{data}}\ \ r_{\mathtt{e}}) = r_{\text{empl}}$, $\delta'(\mathtt{data}, r_{\text{name}}\ \ r_{\text{boss}}) = \delta'(\mathtt{data}, r_{\text{name}}\ r_{\mathtt{e}}) = r_{\text{data}}$, $\delta'(\mathtt{boss}, r_{\text{name}}\ r_{\mathtt{e}}) = r_{\text{boss}}$, $\delta'(\mathtt{name}, r_{\text{content}}\ r_{\mathtt{e}}) = r_{\text{name}}$, $\delta'(\mathtt{e}, \epsilon) = r_{\mathtt{e}}$, and $\delta'(\mathtt{a}, r_1\ r_2) = \bullet$ otherwise. In the corresponding finite forest monoid $A = (G, \Sigma, up, F)$ the monoid G contains the following functions: $g_{\text{empl}} = \{r_{\mathtt{e}} \mapsto r_{\text{empl}}, r_{\text{empl}} \mapsto r_{\text{empl}}\}$, $g_{\text{data}} = \{r_{\mathtt{e}} \mapsto r_{\text{data}}, r_{\text{boss}} \mapsto r_{\text{data}}\}$, $r_{\text{staff}} = \{r_{\mathtt{e}} \mapsto r_{\text{staff}}\}$, $g_{\text{boss}} = \{r_{\mathtt{e}} \mapsto r_{\text{boss}}\}$, $g_{\text{name}} = \{r_{\mathtt{e}} \mapsto r_{\text{name}}\}$, $g_{\text{dataBoss}} = \{r_{\mathtt{e}} \mapsto r_{\text{data}}\}$, $g_{\text{content}} = \{r_{\mathtt{e}} \mapsto r_{\text{content}}\}$, $g_\bullet = \emptyset$, $Id = \{r \mapsto r \mid r \in P\}$, where we have omitted all entries $r \mapsto \bullet$. Note that in this example, the forest monoid has only one element more than the underlying finite automaton. Also, the composition table of these functions is given by $Id \circ g = g \circ Id = g$ for all g and furthermore: $g_{\text{data}} \circ g_{\text{boss}} = g_{\text{dataBoss}}$, $g_{\text{empl}} \circ g_{\text{empl}} = g_{\text{empl}}$, and otherwise $g \circ g' = g_\bullet$. For the function up, we find: $up(\mathtt{staff}, Id) = g_{\text{staff}}$, $up(\mathtt{staff}, g_{\text{empl}}) = g_{\text{staff}}$, $up(\mathtt{employee}, g_{\text{data}}) = g_{\text{empl}}$, $up(\mathtt{employee}, g_{\text{dataBoss}}) = g_{\text{empl}}$, $up(\mathtt{data}, g_{\text{name}}) = g_{\text{data}}$, $up(\mathtt{boss}, g_{\text{name}}) = g_{\text{boss}}$, $up(\mathtt{name}, g_{\text{content}}) = g_{\text{name}}$, and $up(\mathtt{a}, g) = g_\bullet$ otherwise. where the set of accepting functions is given by $F = \{g_{\text{staff}}\}$. ◁

Theorem 1.12. *For every 2mft M and every finite forest monoid A, a 2mft M_A can be constructed such that for all $f \in \mathcal{F}_\Sigma$,*

$$\tau_{M_A}(f) = \tau_M(f) \cap \mathcal{L}(A)$$

The size of M_A is in $\mathcal{O}(|M| \cdot |A|^{l \cdot (d+1)})$ where l is the maximal rank of a state q of M and d is the maximal number of occurrences of states in right-hand sides in M.

Proof. Let $M = (Q, \Sigma, R, Q_0)$ and $A = (G, \Sigma, up, F)$. For each state q in Q with rank $n + 1$ and all monoid elements $g_0, \ldots, g_n \in G$, we generate new states for the intersection 2mtt M_A of the form $\langle q, g_0 g_1 \ldots g_n \rangle$. Such a state is meant to generate all forests $f \in \mathcal{F}_\Sigma(\{y_1, \ldots, y_n\})$ for which there is a run of A starting at the leaves y_i with monoid element g_i and reaching the root of f in g_0. The rules of the new 2mft M_A are: $\langle q, g_0 g_1 \ldots g_n \rangle(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta'$ for every rule $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$ of M and

$\zeta' \in \tau^{g_0 g_1 \cdots g_n}[\zeta]$, where the sets $\tau^{g_0 g_1 \cdots g_n}[.]$ are inductively defined by:

$$\begin{array}{ll}
\tau^{g_0 g_1 \cdots g_n}[y_j] & = \{y_j \mid g_0 = g_j\} \\
\tau^{g_0 g_1 \cdots g_n}[\mathtt{b}\langle\zeta\rangle] & = \{\mathtt{b}\langle\zeta'\rangle \mid up(\mathtt{b}, g') = g_0 \ \wedge \ \zeta' \in \tau^{g' g_1 \cdots g_n}[\zeta]\} \\
\tau^{g_0 g_1 \cdots g_n}[\epsilon] & = \{\epsilon \mid g_0 = e\} \\
\tau^{g_0 g_1 \cdots g_n}[\zeta_1 \zeta_2] & = \{\zeta_1' \zeta_2' \mid g_0 = g_1' \circ g_2' \ \wedge \ \forall \nu : \zeta_\nu' \in \tau^{g_\nu' g_1 \cdots g_n}[\zeta_\nu]\} \\
\tau^{g_0 g_1 \cdots g_n}[q'(op, \zeta_1, \ldots, \zeta_{n'})] & = \{\langle q', g_0 g_1' \ldots g_{n'}'\rangle(op, \zeta_1', \ldots, \zeta_{n'}') \mid \forall \nu : \zeta_\nu' \in \tau^{g_\nu' g_1 \cdots g_n}[\zeta_\nu]\}
\end{array}$$

The set of initial states of M_A is $Q_0' = Q_0 \times F$. By fixpoint induction, we verify for every state q of rank $n \geq 1$, every input forest $f \in \mathcal{F}_\Sigma$, every node $v \in \mathcal{N}(f)$ and monoid elements $g_0, \ldots, g_n$ that:

$$[\![\langle q, g_0, \ldots, g_n\rangle]\!]_f(v) = [\![q]\!]_f(v) \cap \{f' \in \mathcal{F}_\Sigma(Y) \mid up^*(f', g_1 \ldots g_n) = g_0\} \qquad (*)$$

where $Y = \{y_1, \ldots, y_n\}$ and up^* is the extension of up to forests containing variables from Y, namely, for $\underline{g} = g_1 \ldots g_n$ we have $up^*(y_i, \underline{g}) = g_i$, $up^*(\epsilon, \underline{g}) = e$, $up^*(\mathtt{a}\langle f'\rangle, \underline{g}) = up(\mathtt{a}, up^*(f', \underline{g}))$, $up^*(f_1 f_2, \underline{g}) = up^*(f_1, \underline{g}) \circ up^*(f_2, \underline{g})$. The correctness of the construction follows from $(*)$.

For each state in M we have at most $|A|^l$ new states in M_A, if l is the maximal rank of states in M. If we have d occurrences of states in the right-hand side of a rule r of M, we obtain $|A|^{l\cdot(d+1)}$ new rules for r in M_A. Therefore, the new 2mft is of size $\mathcal{O}(|M| \cdot |A|^{l\cdot(d+1)})$ where l is the maximal rank of a state in M and d bounds the number of occurrences of states in right-hand sides in M. □

Note that this construction differs from the corresponding construction for 2mtts in that we now additionally have to take concatenations of forests into account. It is precisely for this operation, that we rely on the monoid structure of the set G.

Example 1.11. Consider the 2mft M of Example 1.9 and the ffm A in Example 1.10. We get an intersection 2mft with the following rules, for $\eta \in \{2, 3\}$.

$$\begin{array}{lll}
1 & \langle q_I, g_{\text{staff}}\rangle(\mathtt{department}, 0) & \to \mathtt{staff}\langle\langle q, Id\ Id\rangle(down, \epsilon)\rangle \quad | \quad \mathtt{staff}\langle\langle q, g_{\text{empl}} Id\rangle(down, \epsilon)\rangle \\
2 & \langle q, g_{\text{empl}} Id\rangle(\mathtt{employee}, \eta, y_b) & \to \mathtt{employee}\langle \mathtt{data}\langle\langle q_{data}, g_{\text{name}}\rangle(down)\rangle\ y_b\rangle \\
3 & & \quad \langle q, g_2 g_{\text{boss}}\rangle(down, \mathtt{boss}\langle\langle q_{data}, g_{\text{name}}\rangle(down)\rangle) \quad \langle q, g_3 Id\rangle(right, y_b) \\
4 & \langle q, g_{\text{empl}} g_{\text{boss}}\rangle(\mathtt{employee}, \eta, y_b) & \to \mathtt{employee}\langle \mathtt{data}\langle\langle q_{data}, g_{\text{name}}\rangle(down)\rangle\ y_b\rangle \\
5 & & \quad \langle q, g_2 g_{\text{boss}}\rangle(down, \mathtt{boss}\langle\langle q_{data}, g_{\text{name}}\rangle(down)\rangle) \quad \langle q, g_3 g_{\text{boss}}\rangle(right, y_b) \\
6 & \langle q, g_0 g_b\rangle(\mathtt{data}, 2, y_b) & \to \langle q, g_0 g_b\rangle(right, y_b) \\
7 & \langle q, g_0 g_b\rangle(\mathtt{subordinates}, 3, y_b) & \to \langle q, g_0 g_b\rangle(down, y_b) \\
8 & \langle q, Id\ g_b\rangle(\epsilon, \eta, y_b) & \to \epsilon \\
9 & \langle q_{data}, g_{\text{name}}\rangle(\mathtt{data}, 2) & \to \langle copy, g_{\text{name}}\rangle(down)
\end{array}$$

where for the monoid elements g_2 and g_3 in the third and the forth rules $g_{\text{empl}} = g_{\text{empl}} \circ g_2 \circ g_3$ holds. Thus, g_2 and g_3 are in $\{g_{\text{empl}}, Id\}$. The element g_0 in lines 6 and 7 is either g_{empl} or Id, whereas g_b in lines 6-9 is in $\{g_{\text{boss}}, Id\}$. Additionally there are rules resulting in a state $\langle q, g_\bullet g_b\rangle$ or $\langle q, g_\bullet\rangle$ for a state $q \in Q$. ◁

The draw-back of this general construction, though, is that the (complement of the) output type with which we aim to intersect, first must be represented as a finite forest monoid.

In general, this alone may incur an exponential blow-up. If, however, the 2mft is *output-linear*, i.e., uses each parameter at most once, then a much cheaper direct construction is possible. In particular, this cheaper construction applies to 2fts since these transducers have no parameters at all.

Theorem 1.13. *Assume that M is an output-linear 2mft. Then for every (possibly nondeterministic) bfa A, a 2mft M_A can be constructed with*

$$\tau_{M_A}(f) = \tau_M(f) \cap \mathcal{L}(A)$$

for all $f \in \mathcal{F}_\Sigma$. The size of the 2mft $|M_A|$ is in $\mathcal{O}(|M| \cdot |A|^{2l\cdot(d+1)})$ where l is the maximal rank of a state q of M and d is the maximal number of occurrences of states in right-hand sides in M.

Proof. Let $M = (Q, \Sigma, R, Q_0)$ and $A = (P, \Sigma, \delta, \{p_0\})$. The idea for the new 2mft M_A for the intersection is to maintain for every possibly produced output forest f a *pair* of states $\langle p, p'\rangle$ so that the automaton A, when starting in p' to the right of f, possibly may arrive in state p to the left. Accordingly, the set Q' of M_A consists of all states $\langle q, p_0p'_0 \dots p_np'_n\rangle$ where $q \in Q$ is of rank $n+1$, i.e., has n accumulating parameters and $p_i, p'_i \in P$ for all i. Accordingly, the rules of the new 2mft are of the form: $\langle q, p_0p'_0 \dots p_np'_n\rangle(\mathtt{a}, \eta, y_1, \dots, y_n) \;\to\; f'$ with $f' \in \tau^{p_0p'_0\cdots p_np'_n}[f]$ for every rule $q(\mathtt{a}, \eta, y_1, \dots, y_n) \to f$ of M. The sets $\tau^{p_0p'_0\cdots p_np'_n}[.]$ are defined by:

$$\begin{aligned}
\tau^{p_jp'_j\ p_1p'_1\cdots p_np'_n}[y_j] &= \{y_j\}\\
\tau^{p_0p'_0\ p_1p'_1\cdots p_np'_n}[\mathtt{b}\langle\zeta\rangle] &= \{\mathtt{b}\langle\zeta'\rangle \mid (p_0, \mathtt{b}, p''_1\ p'_0) \in \delta \wedge (p''_2, \mathtt{e}) \in \delta \wedge \zeta' \in \tau^{p''_1p''_2\ p_1p'_1\cdots p_np'_n}[\zeta]\}\\
\tau^{p_0p'_0\ p_1p'_1\cdots p_np'_n}[\epsilon] &= \{\epsilon \quad\ \mid p_0 = p'_0\}\\
\tau^{p_0p'_0\ p_1p'_1\cdots p_np'_n}[\zeta_1\zeta_2] &= \{\zeta'_1\zeta'_2 \mid \exists\, p:\ \zeta'_1 \in \tau^{p_0p\ p_1p'_1\cdots p_np'_n}[\zeta_1] \wedge \zeta'_2 \in \tau^{pp'_0\ p_1p'_1\cdots p_np'_n}[\zeta_2]\}\\
\tau^{p_0p'_0\ p_1p'_1\cdots p_np'_n}&[q'(op, \zeta_1, \dots, \zeta_m)]\\
&= \{\langle q', p_0p'_0\ p''_1p'''_1 \dots p''_mp'''_m\rangle(op, \zeta'_1, \dots, \zeta'_m) \mid \forall\nu : \zeta'_\nu \in \tau^{p''_\nu p'''_\nu\ p_1p'_1\cdots p_np'_n}[\zeta_\nu]\}.
\end{aligned}$$

The set of initial states of M_A then consists of all states $\langle q, p_0p'\rangle$ where $q \in Q_0$ and $p_0 \in P$ are accepting states of M and A, respectively, and $(p', \mathtt{e}) \in \delta$. The estimation of the size of the resulting transducer is similar to the case of forest monoids — only that we have to replace the number of monoid elements with the number of *pairs* of states. Thus, the new intersection transducer is of size $\mathcal{O}(|M| \cdot |A|^{2l\cdot(d+1)})$. □

1.7.2. *Deciding Emptiness of 2MFTs*

For deciding emptiness of a forest transducer M, we conceptually follow the approach taken for tree transducers. There, we first constructed an alternating tree walking automaton accepting the domain of M for which in the second step, a nondeterministic tree automaton is constructed. In our case, this would mean that we first formally introduce the concept of alternating *forest walking* automata for which in a separate construction, a nondeterministic forest automaton is constructed. In order to simplify this, we will not intermediately rely on forest walking automata. Instead, we consider for each forest f, an *enriched* first-child next-sibling encoding through binary trees. This means that inside each node of the

encoding we additionally record whether or not the current tree node represents a node on the top-level of the forest. Let $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$ denote a set of new symbols of rank 2. Then the ranked alphabet used by the encoding will be $\Sigma_2 = \Sigma \cup \bar{\Sigma} \cup \{\mathtt{e}, \bar{\mathtt{e}}\}$ where the barred symbols will only occur on the rightmost spine in the tree. A dtta with two states $\{\mathtt{t}, \mathtt{n}\}$ can check whether a tree in $\mathcal{T}_{\Sigma_2}$ is the enriched encoding of a forest or not.

Since the encoding is injective, it suffices for a forest transducer M to construct an atwa M' which defines the set of encodings of the domain of M. Then the set of states of the atwa M' is given by $Q' = \{q'_0, \mathtt{t}, \mathtt{n}\} \cup Q \cup Q^{up}$ where $Q^{up} = \{q^{up} \mid q \in Q\}$ is a set of fresh copies of the states in Q and q'_0 serves as fresh initial state of M'. Assume that $q(\mathtt{a}, \eta, y_1, \ldots, y_n) \to \zeta$ is a rule of M and $q_1(op_1, \ldots), \ldots, q_c(op_c, \ldots)$ is the sequence of recursive calls in ζ. If $\eta \in \{2, 3\}$, i.e., if the rule is not applicable to nodes at the top-level of the input forest, then atwa M' has the rules: $q(\mathtt{a}, \eta - 1) \to q'_1(op'_1) \wedge \ldots \wedge q'_c(op'_c)$ where $q'_j(op'_j) = q_j(stay)$ if $op_j = stay$, $q'_j(op'_j) = q_j(down_1)$ if $op_j = down$, $q'_j(op'_j) = q_j(down_2)$ if $op_j = right$, $q'_j(op'_j) = q_j(up)$ if $op_j = left$ and $\eta = 3$, $q'_j(op'_j) = q^{up}_j(stay)$ if $op_j = up$. For states q^{up}, atwa M' has the rules: $q^{up}(\mathtt{a}, 2) \to q(up)$ and $q^{up}(\mathtt{a}, 3) \to q^{up}(up)$.

If on the other hand $\eta \in \{0, 1\}$, i.e., the rule of the 2mft refers to nodes at the top-level of the input forest, then the atwa M' has the rules: $q(\bar{\mathtt{a}}, 2 \cdot \eta) \to q'_1(op'_1) \wedge \ldots \wedge q'_c(op'_c)$ where $q'_j(op'_j) = q_j(stay)$ if $op_j = stay$, $q'_j(op'_j) = q_j(down_1)$ if $op_j = down$, $q'_j(op'_j) = q_j(down_2)$ if $op_j = right$, $q'_j(op'_j) = q_j(up)$ if $op_j = left$. For every rule $q(\mathtt{a}, 0) \to \zeta$ of M with $q \in Q_0$, we add the rule $q'_0(\bar{\mathtt{a}}, 0) \to \mathtt{t}(stay) \wedge \mathtt{q}(stay)$ where the rules for $\mathtt{t}$ and $\mathtt{n}$ simulate the computation of a top-down automaton to verify that the input tree is the enriched encoding of a forest. Thus, the rules of the atwa for q'_0 are meant to spawn a subrun which verifies the encoding and to spawn another subrun which simulates an accepting run of the forest transducer on the binary encoding. In particular, the states q^{up} are auxiliary states to implement the operation up on the binary representation of the forest. More precisely, the rules for the state q^{up} performs the operation up as long as the current node is a right child. If the current node is a left child, a final up-operation is executed to arrive at the tree representation of the father node in state q. Overall, we find:

Theorem 1.14. *For every 2mft M, an atwa M' can be constructed in polynomial time such that $\mathcal{L}(M')$ is the set of enriched binary encodings of the set $\{f \mid \tau_M(f) \neq \emptyset\}$. In particular, $\mathcal{L}(M') \neq \emptyset$ iff $\tau_M \neq \emptyset$.*

We thus obtain an exponential algorithm for deciding emptiness of 2mfts which is optimal. Together with our intersection constructions, this algorithm then can be applied also for type-checking 2mft transducers w.r.t. regular input and output types.

In order to arrive at more tractable algorithms or sub-classes, we can apply the same ideas as for 2mtts: in the first place, we can again approximate the set of output forests by means of a context-free forest grammar. A context-free forest grammar for the intersection with a regular forest language specified through a finite forest monoid is polynomial in the size of the grammar and the number of elements in the monoid and exponential only in $l \cdot (d+1)$ where l is the maximal rank of nonterminals and d is the number of occurrences of nonterminals in right-hand sides. Emptiness for this forest grammar again can be checked

in time linear in the size of the grammar. A practical implementation again may construct a *Datalog* program for the sets of useful nonterminals of the grammar. We can also generalize the notions of b-boundedness and strict b-boundedness for 2mfts. While emptiness for b-bounded 2mfts is decidable in polynomial time (where the exponent again depends on b^2), it is only strict b-boundedness which is preserved by our intersection constructions. Here we only state the corresponding result for output types specified through finite forest monoids.

Theorem 1.15. *Assume M is a strictly b-bounded 2mft and I and O are regular forest languages where I and O are given by a finite forest automaton and a finite forest monoid, respectively. Assume further that l is the maximal rank of a state of M and d is the maximal number of occurrences of state calls in right-hand sides. Then M can be type-checked w.r.t. I and O in time polynomial in the sizes of M, the automaton for I and the automaton for O where the exponent linearly depends on $(b+1)^2 \cdot l \cdot (d+1)$.*

1.7.3. *Notes and References*

Top-down macro forest transducers have been introduced by Perst and Seidl in [PS04]. They are closely related to the top-down transducers of Maneth and Neven [MN99] (but slightly more general). It was shown in [PS04] that, even though mfts are more powerful than mtts, they can be type checked with the same complexity bounds, as macro tree transducers. This idea was extended to two-fold compositions of deterministic mtts, in [MN08]. In [MBPS05], a general forest transformation language TL is introduced which captures most features of XML transformation languages such as XSLT. The language TL supports full MSO pattern matching both for the selection of rules applicable at a node in the input tree and for navigation inside the input tree. Thus, 2mfts can be considered as a sub-language of TL where rules are selected depending on the current state and input label only and where navigation is restricted to immediate neighbors in the input forest. The main contribution of that paper is to show how such transformations can be decomposed into three stay macro tree transducers running on the first-child next-sibling encoding of the XML documents in question. The semantics considered there was OI evaluation of nested calls, but similar results can also be proven for IO evaluation, i.e., call-by-value parameter passing as considered here [Per07].

1.8. Conclusion

In this chapter, we have reviewed basic constructions for tree walking transducers which allow to obtain algorithms for type-checking the transducers w.r.t. regular input and output types. There are three orthogonal variations in which the basic concept of a finite state machine can be made more expressive:

- top-down versus walking
- without parameters versus with parameters;
- on ranked trees versus on unranked forests.

At the very heart of our algorithms for type-checking is to check whether or not a transducer realizes an empty translation. Already for the weakest, i.e., one-way top-down transducers emptiness turns out to be complete for deterministic exponential time. Still, however, we were able to pin-point one major source for the complexity, namely, the number of visits to the same input node. If the transducer visits the same node only constantly often, i.e., is b-bounded for some constant b, then emptiness becomes decidable in polynomial time.

The second ingredient of our algorithm are constructions for computing intersection transducers, i.e., transducers which only produce outputs outside a specified regular set. Here, we considered regular sets as specified by bottom-up deterministic automata (or monoids, in case of forests) or by deterministic top-down automata. The latter construction for forest transducers was at least applicable to *output-linear* transducers, i.e., transducers which use each of their accumulating parameters at most once. Two separate constructions are crucial, since translating top-down deterministic automata into bottom-up automata may incur an extra exponentiation in the number of states. Since these constructions preserve strict b-boundedness we thus overall arrive at a general class of transducers for which type-checking is polynomial.

References

AU71. A. V. Aho and J. D. Ullman. Translations on a Context-Free Grammar. *Inform. and Control*, 19:439–475, 1971.

Bak79. B. S. Baker. Composition of top-down and bottom-up tree transductions. *Inform. and Control*, 41(2):186–213, 1979.

Bar82. M. Bartha. An algebraic definition of attributed transformations. *Acta Cybern.*, 5:409–421, 1982.

BC03. S. Boag and D. Chamberlin et.al., editors. XQuery 1.0: An XML Query Language. W3C Working Draft. Available at http://www.w3.org/TR/xquery/, 2003.

BW05. M. Bojańczyk and I. Walukiewicz. Unranked Tree Algebra. Technical report, University of Warsaw, 2005.

CDG+07. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata, 2007.

CGKV88. S. S. Cosmadakis, H. Gaifman, P. C. Kanellakis, and M. Y. Vardi. Decidable optimization problems for database logic programs (preliminary report). In *STOC*, pages 477–490. ACM Press, New York, 1988.

CM. J. Clark and M. Murata et al. *RelaxNG Specification*. OASIS. Available at http://www.oasis-open.org/committees/relax-ng.

Cou78. B. Courcelle. A representation of trees by languages II. *Theoret. Comput. Sci.*, 7:25–55, 1978.

DE98. F. Drewes and J. Engelfriet. Decidability of finiteness of ranges of tree transductions. *Inform. and Comput.*, 145:1–50, 1998.

EHS07. J. Engelfriet, H. J. Hoogeboom, and B. Samwel. XML transformation by tree-walking transducers with invisible pebbles. In *PODS*, pages 63–72. ACM Press, New York, 2007.

EM99. J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inform. and Comput.*, 154:34–91, 1999.

EM03a. J. Engelfriet and S. Maneth. A Comparison of Pebble Tree Transducers with Macro Tree Transducers. *Acta Inf.*, 39:613–698, 2003.

EM03b. J. Engelfriet and S. Maneth. Macro tree translations of linear size increase are MSO definable. *SIAM J. Comput.*, 32:950–1006, 2003.

Eng08. J. Engelfriet. The time complexity of typechecking tree-walking tree transducers. Technical report, Leiden Institute of Advanced Computer Science, Leiden University, 2008.

ERS80. J. Engelfriet, G. Rozenberg, and G. Slutzki. Tree transducers, L systems, and two-way machines. *J. Comp. Syst. Sci.*, 20:150–202, 1980.

ES77. J. Engelfriet and E.M. Schmidt. IO and OI. (I&II). *J. Comp. Syst. Sci.*, 15:328–353, 1977. and 16:67–99, 1978.

EV85. J. Engelfriet and H. Vogler. Macro Tree Transducers. *J. Comp. Syst. Sci.*, 31:71–146, 1985.

Fal01. D.C. Fallside, editor. XML Schema. W3C Recommendation, W3C, 2 May 2001. Available at http://www.w3.org/TR/xmlschema-0/.

FH07. A. Frisch and H. Hosoya. Towards practical typechecking for macro tree transducers. In *DBPL*, pages 246–260. Springer, Heidelberg, 2007.

Fis68. M. J. Fischer. *Grammars with Macro-like Productions*. PhD thesis, Harvard University, Massachusetts, 1968.

Fri04. A. Frisch. Regular Tree Language Recognition with Static Information. In *PLAN-X*, 2004.

Fül81. Z. Fülöp. On attributed tree transducers. *Acta Cybern.*, 5:261–279, 1981.

FV98. Z. Fülöp and H. Vogler. *Syntax-Directed Semantics; Formal Models Based on Tree Transducers*. Springer, Heidelberg, 1998.

FW04. D. C. Fallside and P. Walmsley. XML Schema part 0: Primer second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/xmlschema-0/.

Gie88. R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Inf.*, 25:355–423, 1988.

Gre78. Sheila A. Greibach. Hierarchy theorems for two-way finite state transducers. *Acta Inf.*, 11:80–101, 1978.

GS84. F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

GS97. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3*, chapter 1. Springer, Heidelberg, 1997.

HFC05. H. Hosoya, A. Frisch, and G. Castagna. Parametric Polymorphism for XML. In *POPL*, pages 50–62. ACM Press, New York, 2005.

HMU01. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, New York, second edition, 2001.

HP02. H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6):961–1004, 2002.

HP03. H. Hosoya and B.C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.

KMS04. C. Kirkegaard, A. Möller, and M.I. Schwartzbach. Static Analysis of XML Transformations in Java. *IEEE Trans. Soft. Eng.*, 30:181–192, 2004.

Knu68. D. E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2(2):127–145, June 1968.

KS81. T. Kamimura and G. Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Inform. and Control*, 49:10–51, 1981.

MBPS05. S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML Type Checking with Macro Tree Transducers. In *PODS*, pages 283–294. ACM Press, New York, 2005.

MLM00. M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, 2000.

MN99. S. Maneth and F. Neven. Structured Document Transformations Based on XSL. In *DBPL*, pages 80–98. Springer, Heidelberg, 1999.

MN04. W. Martens and F. Neven. Frontiers of Tractability for Typechecking Simple XML Trans-

formations. In *PODS*, pages 23–34. ACM Press, New York, 2004.

MN05. W. Martens and F. Neven. On the complexity of typechecking top-down xml transformations. *Theoret. Comput. Sci.*, 336:153–180, 2005.

MN08. S. Maneth and K. Nakano. XML type checking for macro tree transducers with holes. In *PLAN-X*, 2008.

MOS05. A. Möller, M. Olesen, and M. Schwartzbach. Static Validation of XSL Transformations. Technical Report RS-05-32, BRICS, October 2005.

MPS07. S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *ICDT*, pages 254–268. Springer, Heidelberg, 2007.

MS05. A. Möller and M. I. Schwartzbach. The Design Space of Type Checkers for XML Transformation Languages. In *ICDT*, pages 17–36. Springer, Heidelberg, 2005.

MSV03. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. *J. Comp. Syst. Sci.*, 66:66–97, 2003.

Nev02. F. Neven. Automata Theory for XML Researchers. *SIGMOD Record*, 31(3):39–46, 2002.

Per07. T. Perst. *Type Checking XML Transformations*. Dissertation, Technische Universität München, München, 2007.

PS04. T. Perst and H. Seidl. Macro Forest Transducers. *Inf. Proc. Letters*, 89:141–149, 2004.

Rou70. W.C. Rounds. Mappings and Grammars on Trees. *Math. Systems Theory*, 4:257–287, 1970.

Slu85. G. Slutzki. Alternating tree automata. *Theor. Comput. Sci.*, 41(2-3):305–318, 1985.

Tha69. J. W. Thatcher. Transformations and translations from the point of view of generalized finite automata theory. In *STOC*, pages 129–142. ACM Press, New York, 1969.

Vir81. J. Virágh. Deterministic ascending tree automata I. *Acta Cybern.*, 5:33–42, 1981.

W3C00. W3C. *Extensible Markup Language (XML) 1.0*, second edition, 6 October 2000. Available at http://www.w3.org/TR/2000/REC-xml-20001006.