T U M

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

Exact XML Type Checking in Polynomial Time

Sebastian Maneth, Thomas Perst, Helmut Seidl

TUM-I0521
Dezember 05

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

TUM-INFO-12-I0521-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Exact XML Type Checking in Polynomial Time

Sebastian Maneth
National ICT Australia Ltd.
Sydney, Australia
sebastian.maneth@nicta.com.au

Thomas Perst, Helmut Seidl
Technische Universität München
Garching, Germany
{perst,seidl}@in.tum.de

December 20, 2005

Abstract

Macro tree transducers (mtts) are an expressive formalism for reasoning about XSLT-like document transformations. Here we are interested in the exact type checking problem for mtts. While the problem is decidable, the involved technique of inverse type inference is, however, known to have exponential worst-case complexity (already for top-down transformations without parameters). We present new type checking algorithms based on forward type inference through exact characterizations of output languages. The algorithms show that exact type checking for call-by-value mtts with few parameters can be done in polynomial time, given that the output type is specified by a deterministic automaton and that the mtt visits every input node only constantly often. For general mtts, a fast approximative type checking algorithm is presented. The algorithms in this paper are based on results about context-free tree and graph grammars. Finally, the new approach is generalized from mtts to macro forest transducers which additionally support concatenation as built-in output operation.

1 Introduction

Currently, the extensible markup language XML is the 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 in the form of special purpose languages like XDuce, or in the form 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 XML type and a transformation f, can we statically check whether all outputs generated by f on inputs conforming to the input type conform to the output type? XML types are intrinsically more complex than the types found in conventional programming languages, and henceforth the type checking problem for XML poses new challenges on the design of type checking algorithms. The excellent survey [22] 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; for XSLT,

approximative solutions seem to work well for practical transformations [21]. Another approach is to restrict the types and transformations in such a way that the type checking problem becomes decidable; we then refer to the problem as *exact XML type checking*. In the exact setting it is common to use recognizable tree languages as type formalism. Recognizable tree languages capture the tree structure of all known type formalisms for XML, and compositions of macro tree transducers (mtts) capture the tree translation core of the known XML query and transformation languages [7, 20]. Even though the class of translations for which exact type checking is decidable is surprisingly large, the price to be paid for exactness is also extremely large: the complexity of the known algorithms for compositions of mtts is a tower of exponentials whose height grows with the number of transducers in the composition. In fact, 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 [26].

In previous work [17], we have provided an exact type checking algorithm for the very powerful transformation language TL by decomposing every such transformation into at most three mtts — independently of the used match and select patterns. This work together with the results of [7] (showing that pebble tree transducers can be simulated by compositions of mtts) have established macro tree transducers as an adequate model for formally reasoning about XML transformations. For practical considerations, however, one is interested in useful subclasses of transformations for which exact type checking is tractable. Such classes are investigated by Martens and Neven [18]. Their restrictions on transducers are, however, rather severe. Here we report on another successful escape from exponential wasteland into polynomial time: we show that exact type checking can be done in polynomial time for a large class of practically interesting transformations obtained by putting only mild restrictions onto the transducers. More precisely, we show that exact type checking can be solved in polynomial time for any transformation realized by a macro tree transducer with few parameters which translate each node of the input tree at most once ("linear" mtts), or more generally, which translate every node only constantly often (*b-bounded copying* mtts). Note that no restriction is put on the copying that the mtt applies to its accumulating parameters: parameters may freely be copied! Note further, that the above results are for *nondeterministic* transducers with call-by-value semantics. Nondeterminism of transducers is specially important for practical implementations of type checking, because there it can be used to simulate a conditional (like, e.g., a data value comparison) of the query or programming language. Our proofs are based on results from tree language and graph grammar theory. In particular, we use forward type inference and construct from the input type and the transducer a context-free tree or graph grammar, which generates the output set of the transformation. Since checking intersection emptiness of such grammars with deterministic tree automata is in PTIME, we obtain the desired result. Besides exact characterizations of output languages of transducers we also propose a simpler scheme for approximative type checking which is based on an over-approximation of the output language through context-free tree grammars. Finally, we extend our approach to transducers which directly operate on forests of unranked trees and also support concatenation as a built-in output operation.

Related Work

Approximative type checking for XML transformations is typically based on (subclasses of) recognizable tree languages. Using the pattern language XPath [3], XQuery [1] is a functional language for querying XML documents which is strongly-typed. Type checking here is performed via forward type inference rules computing approximative types for each expression. Approximative type inference is also used in the functional transformation language XDuce [15] and its follow-up version CDuce [12]; 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 [14] for parametric polymorphism for XML [13]. Type variables are interpreted as markings indicating the parameterized subparts. In [21] Møller et al. propose a sound type checking algorithm (originally developed for the Java-based language XACT [16]) 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.

The first technique for exact type checking has been proposed by Milo et al. [20]. There, inverse type inference is proposed for translations which can be expressed as k-pebble tree transducers. Inverse type inference was also studied by Tozawa [30] for a subset of XSLT which roughly corresponds to top-down forest transducers (without parameters) [26]. Inverse type inference for a much more expressive transformation language is considered in [17].

2 Macro Tree Transducers

An XML document can be seen as a sequential representation of sequences of unranked trees also called hedges or *forests*. Here is a small example document representing a mail file:

```
<doc>
 <mbox>
  <mail>
   <sender> Homer Simpson </sender>
   <address> homer@simpson.com </address>
   <subject> CONFIDENTIAL </subject>
   <body> ... </body> </mail>
  <spam><mail> ...
   <subject> V.I.A.G.R.A. </subject>
   ... </mail></spam>
 </mbox>
 <trash> ... </trash>
</doc>
```

In this example, the elements `mbox` and `trash` are meant to collect the incoming and deleted mails, respectively. Besides normal `mail` elements, the `mbox` also contains

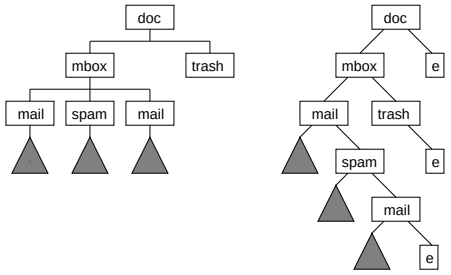


Figure 1: An unranked tree and its binary encoding.

mails inside a `spam` element indicating that these mails have been identified as spam, e.g., by some automated filter.

The transformation model which we consider first, though, does not operate on forests directly but on their representations as binary trees. The empty forest then is represented as a leaf `e` and the content of an element node `a` is coded as the left child of `a` while the forest of right siblings of the element is represented as the right child. Figure 1 illustrates the relationship between unranked trees and their representation as binary trees. In the following we use the term 'tree' as a synonym for *ranked tree*. For a finite (ranked) alphabet Σ the set $\mathcal{T}_\Sigma$ of trees over Σ is defined by:

$$t \;::=\; \mathsf{b} \mid \mathsf{a}(t_1, \ldots, t_k)$$

where $\mathsf{b}, \mathsf{a} \in \Sigma$ are symbols of rank zero and two, respectively; thus, we assume given a fixed rank mapping for the elements of Σ. Often, we consider leaf nodes together with constructor applications by allowing k to equal 0. If leaf nodes additionally can be labeled by elements of a set $Y = \{y_1, y_2, \ldots\}$ of variables, then $\mathcal{T}_\Sigma(Y)$ denotes the set of trees over Σ and Y.

Consider for example a transformation which cleans up the mail folder by moving all sub-documents marked as spam into the trash, while leaving all mail elements untouched. When executed on the example document above, the transformation generates as output:

```
<doc>
 <mbox>
  <mail>
   <sender> Homer Simpson </sender>
   <address> homer@simpson.com </address>
   <subject> CONFIDENTIAL </subject>
   <body> ... </body> </mail>
 </mbox>
 <trash>
  <spam><mail> ...
   <subject> V.I.A.G.R.A. </subject>
   ... </mail></spam>
   ... </trash>
</doc>
```

Using our representation of forests by binary trees (Fig. 1), this transformation is realized by the following macro tree transducer. It has three initial rules transforming the document root and its direct successors:

$$
\begin{aligned}
&1\quad \mathit{init}(\mathsf{doc}(x_1, x_2)) && \to \mathsf{doc}(\mathit{mbox}(x_1), \mathsf{e})\\
&2\quad \mathit{mbox}(\mathsf{mbox}(x_1, x_2)) && \to \mathsf{mbox}(\mathit{mail}(x_1),\\
&&& \qquad \mathit{trashinit}(x_2, \mathit{spam}(x_1)))\\
&3\quad \mathit{trashinit}(\mathsf{trash}(x_1, x_2), y_1) && \to \mathsf{trash}(\mathit{trash}(x_1, y_1), \mathsf{e}),
\end{aligned}
$$

a function *mail* for collecting all ordinary mails in mbox

$$
\begin{aligned}
&4\quad \mathit{mail}(\mathsf{mail}(x_1, x_2)) && \to \mathsf{mail}(\mathit{copy}(x_1), \mathit{mail}(x_2))\\
&5\quad \mathit{mail}(\mathsf{spam}(x_1, x_2)) && \to \mathit{mail}(x_2)\\
&6\quad \mathit{mail}(\mathsf{e}) && \to \mathsf{e},
\end{aligned}
$$

a function *spam* for collecting the spam mails in mbox

$$
\begin{aligned}
&7\quad \mathit{spam}(\mathsf{mail}(x_1, x_2)) && \to \mathit{spam}(x_2)\\
&8\quad \mathit{spam}(\mathsf{spam}(x_1, x_2)) && \to \mathsf{spam}(\mathit{copy}_0(x_1), \mathit{spam}(x_2))\\
&9\quad \mathit{spam}(\mathsf{e}) && \to \mathsf{e}\\
&10\quad \mathit{copy}_0(\mathsf{mail}(x_1, x_2)) && \to \mathsf{mail}(\mathit{copy}(x_1), \mathit{mail}(x_2))\\
&11\quad \mathit{copy}_0(\mathsf{e}) && \to \mathsf{e},
\end{aligned}
$$

and finally, a function *trash* for copying the content of trash and completing it with the spam mails from mbox

$$
\begin{aligned}
&12\quad \mathit{trash}(\mathsf{spam}(x_1, x_2), y_1) && \to \mathsf{spam}(\mathit{copy}_0(x_1), \mathit{trash}(x_2, y_1))\\
&13\quad \mathit{trash}(\mathsf{mail}(x_1, x_2), y_1) && \to \mathsf{mail}(\mathit{copy}(x_1), \mathit{trash}(x_2, y_1))\\
&14\quad \mathit{trash}(\mathsf{e}, y_1) && \to y_1
\end{aligned}
$$

The first line inspects the root node of the document, generates a fresh node doc in the output whose children are determined by the recursive call $\mathit{mbox}(x_1)$ which transforms the mbox node of the input document. The second line produces a mbox node in the output and splits the processing into two parts: the call to the function *mail* transforms the content of mbox, while the call to *trashinit* constructs the transformed trash folder. Since we want to move all mails marked as spam into trash we have to pass the content of mbox to the transformation of trash (because we cannot go back to a node and visit its content again).

The functions *mail* and *spam* define the complementary transformations of the content of mbox. Function *mail* returns the list of ordinary mails while *spam* returns the list of all spam mails. The method is straight-forward: reaching a node with label mail (spam for *spam*) a new node is generated in the output, followed by a copy of its content and the transformation result of the remainder of mbox' content (line 4). Reaching

a node labeled spam (mail), it is discarded and only the rest of the list is processed (line 5).

Function *trash* writes a copy of the content of trash into the output by means of the rules in lines 12 and 13. The last rule copies the already produced list of spam mails from mbox after the last element of trash, by writing the content of parameter y_1 into the output.

Formally, a *macro tree transducer* M (mtt for short) is a tuple

$$(Q, \Sigma, Q_0, R)$$

where Q is the (ranked) set of function names or states, Σ is the (ranked) alphabet of input symbols, $Q_0 \subseteq Q$ is the set of initial functions, and R is a finite set of rules of the form

$$q(\mathsf{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t,$$

where $q \in Q$ is of rank $k+1$, $\mathsf{a} \in \Sigma$ is of rank n, $x_1, \ldots, x_n$ are input variables, $y_1, \ldots, y_k, k \geq 0$ are the accumulating parameters of q, and t is an expression describing the output actions of the rule. Possible actions are described by the grammar:

$$t \quad ::= \quad \mathsf{b}(t_1, \ldots, t_m) \mid y_j \mid q'(x_i, t_1, \ldots, t_m),$$

where b is a label of an output node (we leave an alphabet Δ of output symbols unspecified because our results do not depend on it), y_j is one of the accumulating parameters ($1 \leq j \leq k$), $q' \in Q$ of rank m, and x_i is one of the input variables ($1 \leq i \leq n$). If we deal with binary encodings of forests, the ranks n of input symbols are either zero or two. Also, we assume that initial function symbols $q_0 \in Q_0$ have no accumulating parameters. Accordingly, right-hand sides of q_0-rules do not contain parameters y_j.

Intuitively, the meaning of the action expressions is as follows: The output can either be an element b whose content is recursively determined, the content of one of the accumulating parameters y_j, or a recursive call to some function q' on the i-th subtree of the current input node.

The evaluation of an mtt begins at the root node of the input. Given an input tree t, an mtt M starts processing by evaluating one of its initial functions q_0 for the root node of s. A function q with actual accumulating parameters $t_1, \ldots, t_k$ is applied to an input subtree $\mathsf{a}(s_1, \ldots, s_n)$ by carrying out the following steps. First, we (nondeterministically) choose one of the rules $q(\mathsf{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t$ for q. Then we substitute s_i and t_j for the variables x_i and y_j in the right-hand side t.

Since function calls may be nested, the order in which they are evaluated influences the value of the final output. There are two well-known evaluation orders: outside-in or inside-out. In *outside-in* call-by-name order (OI), outermost calls are evaluated first. The parameters of a function call may themselves contain function calls which are thus transferred to the body in an unevaluated form [9]. The OI order describes the same translations as if leaving the order completely unrestricted [9]. In this paper, however, we consider the *inside-out* evaluation order. This order corresponds to call-by-value parameter passing as provided by mainstream imperative programming languages like C or functional languages such as ML or OCaml. The inside-out evaluation strategy evaluates innermost calls first, meaning that fully evaluated output trees are passed in accumulating parameters when a function call is evaluated.

As in [26], we will not use an operational semantics of mtts based on rewriting, but prefer a denotational formulation which greatly simplifies proof arguments. In perspective, the meaning $[\![a]\!]$ of state q of M with k accumulating parameters is defined as a function from input trees to sets of trees with parameters in $Y = \{y_1, \ldots, y_k\}$:

$$[\![q]\!] : \mathcal{T}_\Sigma \to 2^{\mathcal{T}_\Sigma(Y)}$$

When, during a computation, we evaluate an innermost call $q(s, t_1, \ldots, t_k)$, it suffices to substitute actual parameters t_j for the formal parameters y_j of all terms from $[\![q]\!](s)$ to obtain the set of produced outputs. The values $[\![q]\!]$ for all q are jointly defined as the least functions satisfying:

$$[\![q]\!](\mathsf{a}(s_1, \ldots, s_d)) \supseteq [\![t]\!]\ \sigma$$

for every rule $q(\mathsf{a}(x_1, \ldots, x_d), y_1, \ldots, y_k) \to t$ of M, where

$$\begin{array}{rcl} [\![y_j]\!]\ \sigma & = & \{y_j\} \\ [\![\mathsf{b}(t_1, \ldots, t_m)]\!]\ \sigma & = & \{\mathsf{b}(t'_1, \ldots, t'_m) \mid t'_i \in [\![t_i]\!]\ \sigma\} \\ [\![q'(x_i, t_1, \ldots, t_l)]\!]\ \sigma & = & \{t'[t'_1/y_1, \ldots, t'_l/y_l] \mid \\ & & \quad t' \in [\![q']\!](\sigma(x_i)), t'_i \in [\![t_i]\!]\ \sigma\}, \end{array}$$

σ is a substitution with $\sigma(x_i) = s_i$ for $i = 1, \ldots, d$, and t'_j/y_j denotes the substitution of the tree t'_j for all occurrences of the parameter y_j. Note that the call-by-value semantics is reflected in the last equation: the same trees t'_i are used for all occurrences of a variable y_i in the tree t' corresponding to a potential evaluation of the function symbol q'. The transformation realized by the mtt M on a non-empty input tree s, is the function $\tau_M : \mathcal{T}_\Sigma \to 2^{\mathcal{T}_\Sigma}$ induced by the initial functions from Q_0 of M

$$\tau_M(s) = \bigcup \{[\![q_0]\!](s) \mid q_0 \in Q_0\}$$

For a given set $S \subseteq \mathcal{T}_\Sigma$ we denote by $\tau_M(S)$ the set of all outputs which are produced by M on input trees in S:

$$\tau_M(S) = \bigcup \{\tau_M(s) \mid s \in S\}.$$

Since we are concerned with techniques for type checking we need to define the type of the input and output language of a transformation. Usually, types for XML documents are given by a document type definition (DTD) [31] or a schema [10, 4].

A convenient abstraction of the existing XML type formalisms, in particular DTDs, are recognizable (or: regular) tree languages [23, 24]. In the context of this work we use bottom-tree automata to define recognizable tree languages. As usual, a *bottom-up finite state tree automaton* (fta) is a tuple $A = (P, \Sigma, \delta, F)$ where P is a finite set of states, $F \subseteq P$ is a set of accepting states, and $\delta \subseteq P \times \Sigma \times P^k$ is a set of transitions of the form $(p, \mathsf{a}, p_1 \ldots p_k)$ where a is a symbol of rank k from the alphabet Σ and $p, p_1, \ldots, p_k$ are states in P. Our finite automata will operate on binary representations of forests, i.e., there exists a distinguished symbol e (representing the empty forest) of rank 0 and all other elements of Σ have rank 2.

A transition $(p, \mathsf{a}, p_1 \dots p_k)$ denotes that if A arrives in state p_i after processing the tree t_i, then it can assign state p to the tree $\mathsf{a}(t_1, \dots, t_k)$. A run of A on a tree $t \in \mathcal{T}_\Sigma$ is a mapping which assigns to each node v of t a state $r(v) \in P$, following the transitions in δ.

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, or, equivalently, all trees having runs which map their roots to an accepting state. Coming back to our example, an fta describing (the binary representation of) valid mailbox documents before applying the transformation can have as set of states $P = \{p_{\text{doc}}, p_{\text{mbox}}, p_{\mathsf{e}}, p_{\text{mail}}, p_{\text{trash}}, p_{\text{spam}}, p_{\text{content}}, \dots\}$ and set of transitions:

$$\begin{aligned}\delta = \{&(p_{\text{doc}}, \mathsf{doc}, p_{\text{mbox}}p_{\mathsf{e}}), && (p_{\text{mbox}}, \mathsf{mbox}, p_{\text{spam}}p_{\text{trash}}),\\ &(p_{\text{spam}}, \mathsf{mail}, p_{\text{content}}p_{\text{spam}}), && (p_{\text{spam}}, \mathsf{mail}, p_{\text{content}}p_{\mathsf{e}}),\\ &(p_{\text{spam}}, \mathsf{spam}, p_{\text{mail}}p_{\text{spam}}), && (p_{\text{spam}}, \mathsf{spam}, p_{\text{mail}}p_{\mathsf{e}}),\\ &(p_{\text{mail}}, \mathsf{mail}, p_{\text{content}}p_{\text{mail}}), && (p_{\text{mail}}, \mathsf{mail}, p_{\text{content}}p_{\mathsf{e}}),\\ &(p_{\text{trash}}, \mathsf{trash}, p_{\text{spam}}p_{\mathsf{e}}), && (p_{\mathsf{e}}, \mathsf{e}), \quad \dots \quad \},\end{aligned}$$

where p_{content} is the state characterizing valid content of mails where we have omitted further states and transitions for checking its validity, e.g., of sender, address, subject and body etc. According to this automaton, the element mbox contains a possibly empty sequence of mail and spam elements where every spam element contains arbitrary *sequences* of mails.

Convention. In the rest of this paper, we will not mention the input types in our theorems and proofs. Instead, we always implicitly assume that this type has been encoded into the mtt. This can be done as follows. Assume that the input type S is given by a (possibly nondeterministic) finite tree automaton A. From an mtt M, we then build a new mtt M_A whose function symbols are pairs consisting of a function of M and an automaton state of A. E.g., from a rule

$$q(\mathsf{a}(x_1, x_2), y_1) \to \mathsf{b}(q_1(x_1, y_1), q_2(x_2, y_1))$$

we obtain the following new rule

$$\langle q, p\rangle(\mathsf{a}(x_1, x_2), y_1) \to \mathsf{b}(\langle q_1, p_1\rangle(x_1, y_1), \langle q_2, p_2\rangle(x_2, y_1))$$

if (p, a, p_1p_2) is a transition of A. Note that the predecessor state p_i corresponds to the input variable x_i and therefore occurs in the right-hand side as the second component in recursive calls on x_i. The new set of initial states then is the set of all pairs $\langle q_0, f\rangle$ consisting of an initial state of M and an accepting state of A. In particular, the new mtt M_A is of size $\mathcal{O}(|M| \cdot |A|)$. As usual, the *size* $|M|$ of an mtt M is the sum of the sizes of all its rules where the size of a rule is defined as the sum of the sizes of the terms representing the left- and right-hand sides of the rule. The size $|A|$ of a finite automaton A is defined analogously.

3 Linear Mtts

In this section we want to prove that type checking is in PTIME for mtts that process every node of the input tree at most once. Syntactically, this can be guaranteed

by requiring that in every right-hand side, each input variable x_i occur at most once. Mtts satisfying this restriction are called *linear* [9]. As an example of linear mtt consider the mtt *app* which evaluates $L@$-symbols as concatenation in the binary tree representation of forests. The set or rules of *app* is:

$$\begin{array}{lcl} \mathit{init}(\mathsf{a}(x_1, x_2)) & \to & \mathsf{a}(\mathit{app}(x_1, \mathsf{e}), \mathit{app}(x_2, \mathsf{e})) \\ \mathit{app}(\mathsf{e}, y_1) & \to & y_1 \\ \mathit{app}(\mathsf{a}(x_1, x_2), y_1) & \to & \mathsf{a}(\mathit{app}(x_1, \mathsf{e}), \mathit{app}(x_2, y_1)) \\ \mathit{app}(@(x_1, x_2), y_1) & \to & \mathit{app}(x_1, \mathit{app}(x_2, y_1)) \end{array}$$

The first rule defines the action for the initial function symbol *init*. The label a should be considered as a generic representative of any symbol in the document besides e and @. Since mtts operate on ranked trees, they do not support concatenation as a base operation. We will lift this restriction in Section 5. The mtt *app*, however, shows that it is possible to evaluate symbolic occurrences of a concatenation operator @. Applications of this constructor are evaluated in the last rule: The evaluation of the right child is stored in the parameter while the left child is recursively traversed until reaching the leaf symbol e.

Note that linearity for an mtt in particular implies that the number of function calls in right-hand sides is bounded by the maximal rank of input symbols (in our case: 2). Here, we observe for linear mtts that their output languages can be described by means of rules where the input arguments of all occurring function symbols is simply deleted. Accordingly, the resulting rules no longer specify a transformation but generate output trees. A set of rules which we obtain in this way, constitutes a *context-free tree grammar* (cftg). The grammar characterizing the output language of the linear mtt *app*, for example, looks as follows:

$$\begin{array}{lcl} \mathit{init} & \to & \mathsf{a}(\mathit{app}(\mathsf{e}), \mathit{app}(\mathsf{e})) \\ \mathit{app}(y_1) & \to & y_1 \mid \mathsf{a}(\mathit{app}(\mathsf{e}), \mathit{app}(y_1)) \mid \mathit{app}(\mathit{app}(y_1)) \end{array}$$

where $\mathit{init}, \mathit{app}$ are nonterminals, and *app* has one parameter. Note that selection of rules depending on input symbols now is replaced with nondeterministic choice denoted by "|".

Context-free tree grammars were invented in the 70s [28]. See [8] for a comprehensive study of their basic properties. 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 $q(y_1, \ldots, y_k) \to t$ where $q \in E$ is a nonterminal of rank $k \geq 0$. The right-hand side t is a tree built up from variables $y_1, \ldots, y_k$ by means of application of nonterminal and terminal symbols. In the example, we have represented the cftg only by its set of rules. As for mtts, inside-out (IO) and outside-in (OI) evaluation order for nonterminal symbols must be distinguished [8]. 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 mtts — simply by removing the corresponding input components (and the substitution σ when evaluating right-hand sides). In particular, this semantics assigns to every nonterminal q of rank $k \geq 0$, a set:

$$[\![q]\!] \subseteq \mathcal{T}_\Sigma(Y)$$

for $Y = \{y_1, \ldots, y_k\}$. The language generated by G is:

$$\mathcal{L}(G) = \bigcup \{[\![q_0]\!] \mid q_0 \in E_0\}$$

Theorem 1 (Corollary 5.7 of [9]) The output language of a linear mtt M can be characterized by a cftg G_M. The cftg G_M can be constructed from M in linear time.

Proof. Given a linear mtt $M = (Q, \Sigma, Q_0, R)$ we construct $G_M = (E, \Sigma, P, E_0)$ where $E = Q$ and $E_0 = Q_0$. The new nonterminals differ, however, from the function symbols in M in that the input argument has been canceled. Thus, the symbol q now has rank k iff the function symbol q of M has k accumulating parameters. For every rule

$$q(\mathsf{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t,$$

of the mtt, there is precisely one production in P:

$$q(y_1, \ldots, y_k) \to t',$$

where the new right-hand side t' is obtained by replacing every call $q'(x_i, \ldots)$ in t by $q'(\ldots)$. A formal proof that G_M indeed characterizes the output language of M can be found, e.g., in [9]. ◇

The characterization of mtt output languages by cftgs is useful because (1) emptiness for (IO-)cftgs is decidable in linear time (using a similar algorithm as the one for ordinary context-free (word) grammars, see, e.g.,[5]), and (2) cftgs are closed under intersection with recognizable tree languages [8].

While for the specification of input types, we allowed *nondeterministic* finite tree automata, our further constructions require the output type to be specified by a *deterministic* automaton. As usual, we call a fta $A = (P, \Sigma, \delta, F)$ *deterministic* (dfta) if for each symbol $\mathsf{a} \in \Sigma$ of rank $k \geq 0$ and every tuple $p_1 \ldots p_k$ of states, there is exactly one state p with $(p, \mathsf{a}, p_1 \ldots p_k) \in \delta$, i.e., δ is a function $\delta : \Sigma \times P^k \to P$. In theory, deterministic ftas can be exponentially larger than nondeterministic ones. In practise, however, they are usually not much larger than a corresponding nondeterministic one.

In our example, the output type could, e.g., indicate that after transformation, the element mbox should contain only a list of mail elements. For this purpose, we can use a deterministic bottom-up tree automaton with set of states

$$\{p_{\mathsf{e}}, p_{\text{trash}}, p_{\text{doc}}, p_{\text{mail}}, p_{\text{spam}}, p_{\text{mbox}}, p_{\text{content}}, p_{\text{fail}}, \ldots\},$$

where state p_{content} codes that a mail has a correct content. The leaf e is accepted by the state p_{e}. For all other symbols, we only list the transitions not resulting in the error state p_{fail}. The state p_{doc} is obtained for a node labeled doc with left child mbox and right child e:

e	
	p_{e}

doc	p_{e}
p_{mbox}	p_{doc}

Each table represents δ for the label given in its upper left corner. States in the first row are possible states for the right child, and accordingly states in the first column are possible states for the left child. For mbox and trash we have the following transition tables:

mbox	p_{trash}
p_{mail}	p_{mbox}
p_{e}	p_{mbox}

trash	p_{e}
p_{spam}	p_{trash}
p_{mail}	p_{trash}
p_{e}	p_{trash}

Finally, these are the transitions for mail and spam:

mail	p_{e}	p_{mail}	p_{spam}
p_{content}	p_{mail}	p_{mail}	p_{spam}

spam	p_{e}	p_{mail}	p_{spam}
p_{content}	p_{spam}	p_{spam}	p_{spam}

Now assume the output type is given by a dfta. In order to obtain precise complexity estimations for type checking, we briefly recall the construction for intersecting cftgs with dftas.

Theorem 2 Let G be a cftg.

1. It can be decided in linear time whether or not $\mathcal{L}(G) = \emptyset$.
2. For every dfta A, a cftg G_A can be constructed such that $\mathcal{L}(G_A) = \mathcal{L}(G) \cap \mathcal{L}(A)$. The grammar G_A can be constructed in time $\mathcal{O}(N \cdot n^{k+1+d})$ where N is the size of G, k is the maximal rank of the nonterminals of G, d is the maximal number of occurrences of nonterminals in right-hand sides, and n is the number of states of the finite tree automaton.

Proof. Let $A = (P, \Sigma, \delta, F)$ and $G = (E, \Sigma, P, E_0)$. The set of nonterminals of G_A consists of all tuples $\langle q, p_0, p_1 \dots p_k \rangle$ where $q \in E$ of rank k and $p_0, \dots, p_k \in P$. The new nonterminal $\langle q, p_0 \dots p_k \rangle$ is meant to generate all trees t of the nonterminal q of G for which there is a run of A, starting in states p_i for leaves y_i, and ending in state p_0.

For every rule $q(y_1, \dots, y_k) \to g$ of G the intersection grammar G_A has the rules:

$$[q, p_0 \dots p_k](y_1, \dots, y_k) \;\to\; g'$$

where $g' \in \mathcal{T}^{p_0 \dots p_k}[g]$ and the sets $\mathcal{T}^{p_0 \dots p_k}[g]$ are inductively defined as:

$$\begin{array}{lcl}
\mathcal{T}^{p_i p_1 \dots p_k}[y_i] & = & \{y_i\} \\
\mathcal{T}^{p_0 p_1 \dots p_k}[\mathsf{a}(g_1, \dots, g_m)] & = & \{\mathsf{a}(g'_1, \dots, g'_m) \mid \\
\qquad \delta(\mathsf{a}, p'_1, \dots, p'_m) = p_0 \wedge \forall\, i :\; g'_i \in \mathcal{T}^{p'_i p_1 \dots p_k}[g_i]\} & & \\
\mathcal{T}^{p_0 p_1 \dots p_k}[q'(g_1, \dots, g_m)] & = & \\
\{[q', p_0 p'_1 \dots p'_m](g'_1, \dots, g'_m) \mid \forall\, i :\; g'_i \in \mathcal{T}^{p'_i p_1 \dots p_k}[g_i]\} & &
\end{array}$$

By fixpoint induction, we verify for every $q \in E$ of rank $k \geq 0$ and states $p_0, \ldots, p_k \in P$ that:

$$[\![q, p_0 \ldots p_k]\!] = [\![q]\!] \cap \{t \in \mathcal{T}_\Sigma(Y) \mid \delta^*(t, p_1 \ldots p_k) = p_0\} \quad (*)$$

where $Y = \{y_1, \ldots, y_k\}$ and δ^* is the extension of the transition function of A to trees containing variables from Y, namely, for $\underline{p} = p_1 \ldots p_k$:

$$\begin{array}{lcl} \delta^*(y_i, \underline{p}) & = & p_i \\ \delta^*(\mathsf{a}(t_1, \ldots, t_m), \underline{p}) & = & \delta(\mathsf{a}, \delta^*(t_1, \underline{p}), \ldots, \delta^*(t_m, \underline{p})) \end{array}$$

The set of new initial nonterminals consists of all $[q_0, f]$ where $q_0 \in E_0$ and f is an accepting state of A. The correctness of the construction follows from equation $(*)$.

For a cftg of size N with at most k parameters and at most d occurrences of nonterminals in right-hand sides, and a tree automaton with n states, the intersection grammar is of size $\mathcal{O}(N \cdot n^{k+1+d})$: since there can be in the worst case n^{k+1} copies of a rule of the cftg G and for every non-terminal occurring in the right-hand side we may choose arbitrary output states. This completes the proof. $\diamond$

Note that the complexity bound provided for the construction of Theorem 2 is a worst-case estimation. Practically, one can organize the construction such that only nonterminals of the intersection grammar are constructed which generate nonempty languages. Also, trap states can be excluded which may not show up in any accepting run to the automaton A. In this way, the number of newly constructed nonterminals will generally be much smaller than the bounds stated in the theorem. Consider, e.g., the cftg characterizing the output language of the linear mtt *app*. Assume we are interested in an output type as given by a dfta with the following transitions:

e	
	p_e

a	p_e	p_a
p_e	p_a	p_a
p_a	p_a	p_a

where p_a is the final state and p_e is the state assigned to leaf nodes e. In order to obtain a characterization of erroneous outputs, we construct the complement automaton A by inverting final and non-final states. Then we construct the intersection grammar with A:

$$\begin{array}{lcl}
[\mathit{init}, p_\mathsf{a}] & \to & \mathsf{a}([\mathit{app}, p_\mathsf{e}p_\mathsf{e}](\mathsf{e}), [\mathit{app}, p_\mathsf{e}p_\mathsf{e}](\mathsf{e})) \mid \\
& & \mathsf{a}([\mathit{app}, p_\mathsf{a}p_\mathsf{e}](\mathsf{e}), [\mathit{app}, p_\mathsf{a}p_\mathsf{e}](\mathsf{e})) \\
[\mathit{app}, p_\mathsf{a}p_\mathsf{e}](y_1) & \to & \mathsf{a}([\mathit{app}, p_\mathsf{e}p_\mathsf{e}](e), [\mathit{app}, p_\mathsf{e}p_\mathsf{e}](y_1)) \mid \\
& & \mathsf{a}([\mathit{app}, p_\mathsf{a}p_\mathsf{e}](\mathsf{e}), [\mathit{app}, p_\mathsf{e}p_\mathsf{e}](y_1)) \mid \\
& & \mathsf{a}([\mathit{app}, p_\mathsf{a}p_\mathsf{e}](\mathsf{e}), [\mathit{app}, p_\mathsf{a}p_\mathsf{e}](y_1)) \mid \\
& & [\mathit{app}, p_\mathsf{a}p_\mathsf{a}]([\mathit{app}, p_\mathsf{a}p_\mathsf{e}](y_1)) \\
[\mathit{app}, p_\mathsf{a}p_\mathsf{a}](y_1) & \to & \mathsf{a}([\mathit{app}, p_\mathsf{a}p_\mathsf{e}](\mathsf{e}), [\mathit{app}, p_\mathsf{a}p_\mathsf{a}](y_1)) \mid \\
& & [\mathit{app}, p_\mathsf{a}p_\mathsf{a}]([\mathit{app}, p_\mathsf{a}p_\mathsf{a}](y_1)) \mid y_1 \\
[\mathit{app}, p_\mathsf{e}p_\mathsf{e}](y_1) & \to & [\mathit{app}, p_\mathsf{e}p_\mathsf{e}]([\mathit{app}, p_\mathsf{e}p_\mathsf{e}](y_1)) \mid y_1
\end{array}$$

In our example, the initial nonterminal *init* gives rise only to a tuple containing the state p_a. We conclude therefore that the intersection is indeed empty.

In general, we are interested in type checking transformations implemented through mtts. Since we have already coded the input type specification into the mtt M, type checking amounts to verifying, for a given output type T_{out}, whether or not $\tau_M(\mathcal{T}_\Sigma) \subseteq T_{\text{out}}$. Applying our above constructions we obtain our first type checking result.

Theorem 3 Type checking for a linear mtt M can be done in time $\mathcal{O}(N \cdot n^{k+1+d})$ where N is the size of the mtt, k is the maximal number of accumulating parameters, d is the maximal rank of an input symbol and n is the size of a dfta for the output type.

Proof. Following Theorem 1, we first construct for M the cftg G_M characterizing the output language of M. Then, following Theorem 2, we construct the intersection grammar between G_M and the complement automaton of the dfta. Since M is linear, there are at most d nonterminals in right-hand sides of G_M. Finally we check non-emptiness of the intersection grammar, which can be done in linear time. $\diamond$

The algorithm in the proof of Theorem 3 can also be applied to *NON-linear* mtts. Then, the constructed cftg does no longer precisely characterize the output language of the transformation, because dependencies on input subtrees (viz. several function calls on the same input variable x_i) have been lost in the grammar. Rather, the cftg generates a superset and hence provides a conservative over-approximation.

Theorem 4 Assume G_M is the cftg constructed for an arbitrary mtt 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 it accepts only correct programs.

Consider, e.g., our example of Section 2. We construct for each mtt rule exactly one production of the cftg G_M and obtain:

$$
\begin{array}{lcl}
\mathit{init} & \to & \mathsf{doc}(\mathit{mbox}, \mathsf{e}) \\
\mathit{mbox} & \to & \mathsf{mbox}(\mathit{mail}, \mathit{trashinit}(\mathit{spam})) \\
\mathit{trashinit}(y_1) & \to & \mathsf{trash}(\mathit{trash}(y_1), \mathsf{e}) \\
\mathit{trash}(y_1) & \to & \mathsf{spam}(\mathit{copy}_0, \mathit{trash}(y_1)) \mid \\
 & & \mathsf{mail}(\mathit{copy}, \mathit{trash}(y_1)) \mid y_1 \\
\mathit{copy}_0 & \to & \mathsf{mail}(\mathit{copy}, \mathit{copy}_0) \mid \mathsf{e} \\
\mathit{mail} & \to & \mathsf{mail}(\mathit{copy}, \mathit{mail}) \mid \mathit{mail} \mid \mathsf{e} \\
\mathit{spam} & \to & \mathsf{spam}(\mathit{copy}_0, \mathit{spam}) \mid \mathit{spam} \mid \mathsf{e}
\end{array}
$$

where the two function calls *mail* and *spam* on the same input variable x_1 are simply represented by the two nonterminals *mail* and *spam*, respectively. The intersection grammar contains the following rules for *init*:

$$
\begin{array}{lcl}
[\mathit{init}, p_{\text{doc}}] & \to & \mathsf{doc}([\mathit{mbox}, p_{\text{mbox}}], \mathsf{e}) \\
[\mathit{mbox}, p_{\text{mbox}}] & \to & \mathsf{mbox}([\mathit{mail}, p_{\text{mail}}], \\
 & & \qquad [\mathit{trashinit}, p_{\text{trash}} p_{\mathsf{e}}]([\mathit{spam}, p_{\mathsf{e}}])) \mid \\
 & & \mathsf{mbox}([\mathit{mail}, p_{\text{mail}}], \\
 & & \qquad [\mathit{trashinit}, p_{\text{trash}} p_{\text{spam}}]([\mathit{spam}, p_{\text{spam}}]))
\end{array}
$$

where we only have listed rules using nonterminals with nonempty semantics. We omit the remaining rules of the intersection grammar but note that no nonterminal $[\mathit{init}, p]$ is found to represent a nonempty set for $p \neq p_{\text{doc}}$. We thus conclude that type checking succeeds, and indeed, all mail sequences marked spam have been removed by the mtt from mbox.

Note that when approximating the output languages of general mtts with cftgs, then we no longer may 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 the well-known trick of constructing an equivalent cftg G' where the maximal depth of right-hand sides is bounded by 2 [11]. This will increase the size of the grammar by a factor at most $\mathcal{O}(k^2)$ and reduce the maximal number occurring nonterminals in right-hand sides to at most $k + 1$.

4 MTTs with bounded copying

In this section we investigate in how far the exact techniques from the last section can be extended to more general classes of mtts. The goal again is to find precise and tractable characterizations of the output language. If the mtt is no longer linear, we must take into account that distinct function calls could refer to the same input node and therefore must be "glued together", i.e., be jointly evaluated.

In general, an arbitrary number of function calls may be applied to the same subdocument of the input. In many practical transformations, though, this is not the case. Instead, typical transformations consult every part of the input only a small number of times. In our running example with mail and spam, every subtree of the input is processed at most twice. Therefore, we consider the subclass of mtts which are *b-bounded copying in the input*, $b \geq 1$. Every such mtt is allowed to process every subtree of the input at most b times. Thus in principle, b-bounded copying is a semantic property. In particular, it implies that every variable x_i occurs at most b times in corresponding right-hand sides (as, e.g., in the syntactic definition in [19]), but it also rules out that increasing numbers of copies can be produced, say, by repeated application of copying rules in a loop. This (dynamic) property has been defined and investigated in [6] under the name "finite-copying in the input"; intuitively, the property says that the state sequence at any given node of the input tree (i.e., the sequence of states that process the node) may not be longer than b; note, however, that [6] only deals with total deterministic mtts. Instead of dealing with semantic b-bounded copying, we find it more convenient to consider syntactic b-bounded copying only. In order to introduce this notion precisely, let us assume w.l.o.g. that every state of M is useful, i.e., can produce an output for at least one input tree.

For all states q of M, we define the maximal copy numbers $b[q]$ as the least fixpoint of a constraint system over:

$$\mathcal{N} = \{1 < 2 < \ldots < \infty\},$$

the complete lattice of naturals extended with ∞. The constraint system consists of all constraints:

$$b[q] \;\geq\; b[q_1] + \ldots + b[q_m]$$

where $q(\mathsf{a}(x_1,\ldots,x_l),y_1,\ldots,y_k) \to t$ is a rule of M and, for some i, and $q_1,\ldots,q_m$ is the sequence of occurrences of calls $q_j(x_i,\ldots)$ for the same variable x_i in the right-hand side t. The mtt M then is *syntactically b-bounded copying* (or, a b-mtt for short) iff for all states, $[q] \leq b$.

In our mailbox example this constraint system looks like:

$$\begin{array}{lcllcl}
b[\textit{init}] & \geq & b[\textit{mbox}] & b[\textit{mbox}] & \geq & b[\textit{trashinit}] \\
b[\textit{mbox}] & \geq & b[\textit{mail}] + b[\textit{spam}] & b[\textit{mail}] & \geq & b[\textit{copy}] \\
b[\textit{spam}] & \geq & b[\textit{copy}_0] & b[\textit{trashinit}] & \geq & b[\textit{trash}] \\
b[\textit{trash}] & \geq & b[\textit{copy}] & b[\textit{trash}] & \geq & b[\textit{copy}_0] \\
b[\textit{copy}_0] & \geq & b[\textit{mail}] & & &
\end{array}$$

where we have removed trivial constraints such as $b[\textit{mail}] \geq b[\textit{mail}]$. Thus, the copy number of *init* and *mbox* equal 2 while all other copy numbers equal 1 (the least element of $\mathcal{N}$).

The least solutions of such constraint systems over the naturals can be determined in linear time [29]. In fact, the latter paper also provides a simple criterion which precisely characterizes whether or not all copy numbers are finite. It amounts to checking that for every constraint $b[q] \geq b[q_1] + \ldots + b[q_m]$, whenever q and q_j are in the same strong component of the variable dependence graph of the constraint system then the constraint is of the simple form: $b[q] \geq b[q_j]$ only. The next theorem thus follows from the definitions and [29].

Theorem 5 Assume that M is an mtt without useless states.

1. It can be decided in linear time whether the mtt M is syntactically b-bounded for some b.
2. If M is syntactically b-bounded-copying then $b \leq 2^{|M|}$.
3. The syntactic copy numbers of every state of M can be determined in linear time.

Still, we may worry how syntactic bounded copying is related to semantic bounded copying. An alternative syntactic restriction, which implies our restriction of b-bounded copying, is the notion of "single use restriction (sur)", originally invented in the context of attribute coupled grammars, but later generalized to mtts [6]. In that paper it is shown for a restricted case of mtts, that finite-copying (=semantic bounded copying) implies sur, and hence syntactic bounded copying. We conjecture that, also for our (nondeterministic, IO) mtts, a similar result can be proved which shows that the semantic b-bounded copying restriction implies the syntactic one.

Depart from fundamental considerations, we here are interested in mtts where every input node is visited only a *small* number of times. Since an input node can be visited more than once, we need a language model which allows us to glue together multiple computations. For this purpose, we propose *coupled replacement grammars* (crgs). This concept is a restricted form of *context-free graph grammars* (cf., e.g., [5]) and seems similar to the grammar formalism proposed in [27]. We deliberately have refrained from introducing the latter (notationally heavy) concept in order to present an abridged version which is streamlined for our application. This abridged version

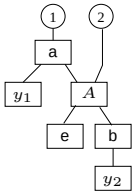


Figure 2: An example hypergraph.

can be viewed as a mild generalization of context-free tree grammars where right-hand sides are no longer trees. In particular, we allow nonterminals to have multiple roots ("outputs"). Thus, right-hand sides of rules now are (restricted forms of) hypergraphs such as the one shown in Figure 2. The idea is that the new right-hand sides conceptually consist of a tuple of trees which are glued together at nonterminals. Therefore, we can assume that every argument of a nonterminal is dedicated to a specific output. Thus, the functionality of a nonterminal A with r outputs can be described by a tuple $\langle k_1, \ldots, k_r \rangle$ meaning that the first k_1 arguments are dedicated to the first output and so forth. This tuple is called the *sort* of A. Semantically, the nonterminal A is meant to return r-tuples of trees. In order to obtain a linear representation which is reminiscent of terms, we allow us to introduce unique auxiliary names for the components of an occurrence of the nonterminal A which then are used similar to nonterminals in cftgs. The hypergraph from Figure 2 then would be represented as:

$$\begin{array}{ll} \textbf{let} & (A_1, A_2) = A \\ \textbf{in} & (\mathsf{a}(y_1, A_1(\mathsf{e})), A_2(\mathsf{b}(y_2))) \end{array}$$

In general, we define a coupled context-free tree grammar (ccftg) to consist of rules

$$A(y_1, \ldots, y_m) \rightarrow g$$

where A is a nonterminal of sort $\langle k_1, \ldots, k_r \rangle$ with $m = k_1 + \ldots + k_r$ and g is a hyper-graph of the form:

$$\textbf{let } \mathit{defs} \textbf{ in } (t_1, \ldots, t_r)$$

where *defs* is a sequence of component declarations

$$(B_1, \ldots, B_s) = B$$

with B a nonterminal of some sort $\langle k'_1, \ldots, k'_s \rangle$ such that the components B_j have ranks k'_j. Furthermore, $(t_1, \ldots, t_r)$ is a tuple of trees which are built up from variables, terminals and components according to the following restrictions:

1. Every declared component B_j occurs at most once;
2. The tree t_i may only contain variables $y_{m_i+1}, \ldots, y_{m_i+k_i}$ for $m_i = k_1 + \ldots + k_{i-1}$.

The semantics of rewriting in a hyperedge replacement grammar in a "standard" language theoretic way (by means of derivation steps) can be found in [5], but here we are only interested in hypergraphs as compact representations of tree tuples. Therefore, we only consider a least fixpoint semantics based on the interpretation of nonterminals as sets of tree tuples. More precisely, we interpret a nonterminal A of sort $\langle k_1, \ldots, k_r \rangle$, by a subset:

$$[\![A]\!] \subseteq \mathcal{T}_\Sigma(Y_1) \times \ldots \times \mathcal{T}_\Sigma(Y_r)$$

where $Y_j = \{y_1, \ldots, y_{k_j}\}$.

The semantics is based on an evaluation function $[\![g]\!]\ \alpha$ of a right-hand side g for A w.r.t. an appropriate assignment α of the components B_j to trees. If $g \equiv$ **let** *defs* **in** $(t_1, \ldots, t_r)$ then

$$[\![g]\!]\ \alpha \ = \ ([\![t_1]\!]\ \alpha, \ldots, [\![t_r]\!]\ \alpha)$$

where the values $[\![t_i]\!]\ \alpha$ are inductively defined by:

$$\begin{array}{lcl} [\![\mathsf{a}(e_1, \ldots, e_n)]\!]\ \alpha & = & \mathsf{a}([\![e_1]\!]\ \alpha, \ldots, [\![e_n]\!]\ \alpha) \\ [\![y_{m_i+j}]\!]\ \alpha & = & y_j \qquad (m_i = k_1 + \ldots + k_{i-1}) \\ [\![B_j(e_1, \ldots, e_n)]\!]\ \alpha & = & \alpha(B_j)[[\![e_1]\!]\ \alpha/y_1, \ldots, [\![e_{m'_j}]\!]\ \alpha/y_n] \end{array}$$

Note that during evaluation, we have re-scaled the indices of the variables $y_{\ldots}$ in t_i to be from the set $y_1, \ldots, y_{k_i}$.

The sets $[\![A]\!]$, A a nonterminal, then are obtained as the least assignment of such sets for which

$$[\![A]\!] \ \ni \ [\![g]\!]\ \alpha$$

for every rule $A(y_1, \ldots, y_m) \ \rightarrow \ g$ of G, and assignments α for which

$$(\alpha(B_1), \ldots, \alpha(B_s)) \in [\![B]\!]$$

whenever $(B_1, \ldots, B_s) = B$ is a definition in g.

The *language* $\mathcal{L}(G)$ generated by the grammar G then is given by the union of the sets $[\![S]\!]$, S a start symbol of G.

Now consider again a b-bounded-copying mtt M. The goal is to glue together inside a right-hand side of M, all occurring function symbols which are called with the same x_i. Assume for example, that the following two rules are part of a 2-mtt.

$$\begin{array}{lcl} q_1(\mathsf{a}(x_1, x_2), y_1, y_2) & \rightarrow & \mathsf{b}(q_2(x_2, y_2), q_3(x_1, q_3(x_1, y_1))) \\ q_2(\mathsf{a}(x_1, x_2), y_1) & \rightarrow & q_3(x_2, c(y_1)) \end{array}$$

The right-hand sides can be represented by the two graphs shown in Figure 3. Note that both calls to q_3 in the first rule are on the same variable x_1 as well as q_2 in the first rule and q_3 in the second rule on x_2. The idea must therefore be to jointly produce the outputs for these pairs of calls by gluing them together into nonterminals $[q_3, q_3]$ and $[q_2, q_3]$, respectively. The graphical representation of the resulting structure is displayed in Figure 4. It seems as if the gluing had introduced a *cycle* into the structure. The present cycle, though is "harmless" since it only introduces a flow from "one component" of the new box into the other and not vice versa. The right-hand sides of

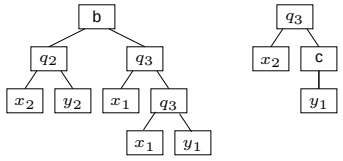


Figure 3: Right-hand sides as trees.

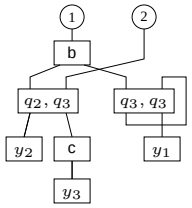


Figure 4: The merged right-hand sides.

the new function symbol $[q_2, q_3]$, e.g., then are obtained by pairing together all possible pairs of right-hand sides of rules for the same input symbol and again gluing together function symbols which are called for the same inputs.

Technically, we define the gluing operation for a given sort $\underline{k} = \langle k_1, \ldots, k_r \rangle$ and sequence $\underline{t} = (t_1, \ldots, t_r)$ of trees as follows. For every x_i occurring in the sequence, we introduce a definition $(\langle i, 1\rangle, \ldots, \langle i, s\rangle) = [q_1, \ldots, q_s]$ if $q_1, \ldots, q_s$ is the sequence of occurrences of function symbols applied to x_i. Let $defs$ denote the sequence of these definitions. Then the transformation $\mathsf{glue}(\underline{k}, \underline{t})$ should yield the graph:

$$\mathbf{let}\ defs\ \mathbf{in}\ (t'_1, \ldots, t'_r)$$

where the primed term t'_i is obtained from t_i by replacing variables y_j with y_{m_i+j} for $m_i = k_1 + \ldots + k_{i-1}$ and by replacing the l-th occurrence of a call $q'(x_i, \ldots)$ with $\langle l, i\rangle(\ldots)$. The gluing operation allows us to construct a ccftg which characterizes the output language of a b-mtt:

Theorem 6 For every b-mtt M, there is a ccftg G_M such that $\tau_M(\mathcal{T}_\Sigma) = \mathcal{L}(G_M)$. The ccftg G_M can be constructed in time $\mathcal{O}(|M|^b)$.

Proof. The construction of G_M is the immediate generalization of the corresponding construction of a cftg for linear mtts. Now, however, the nonterminals consist of tuples $[q_1, \ldots, q_r]$ of states of the b-mtt. The initial nonterminals of G_M consist of all $[q_0]$, q_0 an initial state of M. Every rule $q(\mathsf{a}(x_1, \ldots, x_d), y_1, \ldots, y_k) \rightarrow t$ of M gives rise to the grammar rule:

$$[q](y_1, \ldots, y_k) \rightarrow \mathsf{glue}(\langle k\rangle, (t))$$

Furthermore, whenever a fresh nonterminal $[q_1, \ldots, q_r]$ occurs in a right-hand side, q_j of rank k_j, we consider all tuples of rules:

$$q_i(\mathsf{a}(x_1, \ldots, x_d), y_1, \ldots, y_k) \rightarrow t_i \quad , \quad i = 1, \ldots, r$$

which agree in the input symbol a and construct:

$$[q_1, \ldots, q_r](y_1, \ldots, y_m) \;\rightarrow\; \mathsf{glue}(\underline{k}, \underline{t})$$

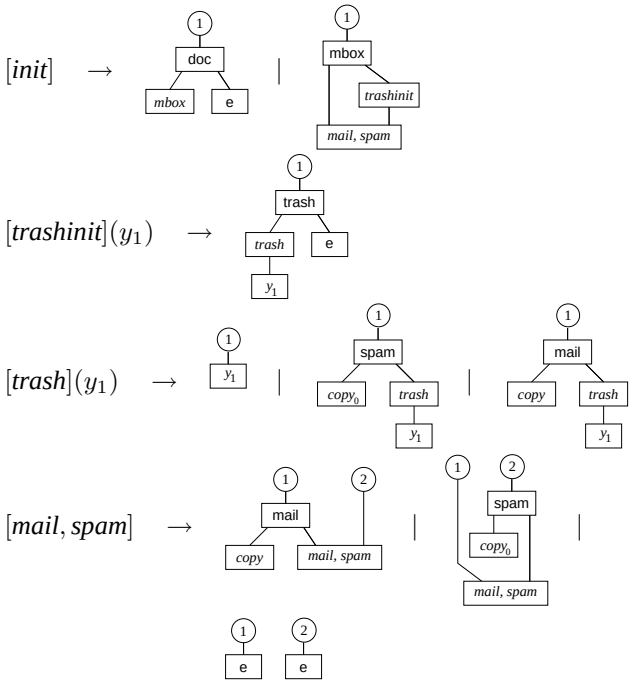
for $m = k_1 + \ldots + k_r$, $\underline{k} = \langle k_1, \ldots, k_r \rangle$ and $\underline{t} = (t_1, \ldots, t_r)$. During this construction, we maintain the invariant that the sum of the copy numbers $b[q_i]$ of the nonterminal $[q_1, \ldots, q_r]$ on the left-hand side of the rule is an upper bound to the corresponding sums of copy numbers for the nonterminals occurring in the newly constructed right-hand side. Since M is structurally b-bounded-copying, this invariant ensures that only tuples of states of length at most b are constructed. This gives the complexity estimation.

In order to prove that the resulting grammar G_M characterizes the output language of M, we verify by fixpoint induction:

$$[\![q_1, \ldots, q_r]\!] = \bigcup \{ [\![q_1]\!](s) \times \ldots \times [\![q_r]\!](s) \mid s \in \mathcal{T}_\Sigma \}$$

where $[\![\ldots]\!]$ on the left- and right hand sides are meant w.r.t. to the grammar G_M and to the mtt M, respectively. ◇

Coming back to our example (cf. Section 2), we construct from the mtt rules the following grammar:



where the productions for the nonterminal $[\textit{copy}]$ can be constructed straight-forwardly from the mtt rules. The rules for the new nonterminal $[\textit{mail}, \textit{spam}]$ indicate how pairs of output trees are generated when the mtt functions *mail* and *spam* are applied to the same input.

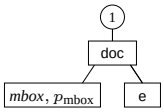
As for linear mtts, the characterization of output languages by means of grammars is only useful if the grammar formalism is effectively closed under intersection with recognizable languages and each grammar can be tested for emptiness. The key observation is that these two properties hold for ccftgs.

Theorem 7 Assume G is a ccftg.

1. It can be decided in linear time whether or not $\mathcal{L}(G) = \emptyset$.
2. For every dfta A, a ccftg G_A can be constructed such that $\mathcal{L}(G_A) = \mathcal{L}(G) \cap \mathcal{L}(A)$. The grammar G_A can be constructed in time $\mathcal{O}(N \cdot n^{m+(d+1)\cdot b})$ where N is the size of G, m is the maximal number of inputs of a nonterminal of G, d is the maximal number of occurrences of a nonterminal in right-hand sides, b is the maximal number of outputs of a nonterminal and n is the size of the dfta.

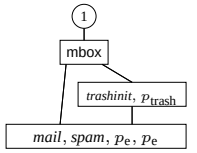
The proof is based on a straight-forward generalization of the corresponding construction in Theorem 2. Note that now, however, nonterminals may return up to b results. The number of nonterminals therefore may increase by a factor n^{m+b}. Given one of these nonterminals, we may have to choose a different state for each output of a nonterminal occurring in a right-hand side. This explains the additional factor $n^{b\cdot d}$.

Let us return to our running mailbox example together with the output type from Section 3. We present here only a manageable part of all productions of the intersection, but all omitted rules can be constructed in the same way. First we present the rules constructed from the initial nonterminal *init*:

$[\mathit{init}, p_{\text{doc}}] \quad \rightarrow$ 

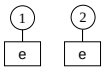
They are obtained from the ccftg rules by building all possible and consistent new right-hand sides in which every nonterminal is equipped with dfta states for the inputs as well as for the outputs.

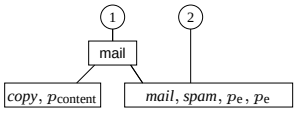
The new nonterminals consist of a nonterminal from the ccftg characterizing the output language and a tuple of sequences of states from the bottom-up tree automaton: one sequence for every component of the nonterminal. The nonterminal *init*, e.g., is annotated with the state p_{doc} alone, since *init* has only one component with no inputs. In the following rules, the nonterminal *mbox* is annotated with p_{mbox} where the annotations of the right-hand sides differ in the states for the outputs of the nonterminal $[\mathit{mail}, \mathit{spam}]$.

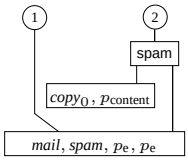
$[\mathit{mbox}, p_{\text{mbox}}] \quad \rightarrow$  $\mid g_2 \mid g_3 \mid g_4$

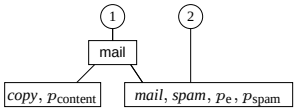
The graphs g_2, g_3, g_4 are as the first one, but the bottom edge has as last two entries of its label the pairs p_{e}, p_{spam} for g_2, p_{mail}, p_{spam} for g_3, and p_{mail}, p_{e} for g_4. We

will not list all rules of the intersection grammar but consider in detail the nonterminal $[\mathit{mail}, \mathit{spam}]$. This nonterminal has been obtained through merging of two mtt function symbols without accumulating parameters and therefore is annotated with a pair of states. We thus obtain the rules:

$[\mathit{mail}, \mathit{spam}, p_{\mathrm{e}}, p_{\mathrm{e}}] \rightarrow$ 

$[\mathit{mail}, \mathit{spam}, p_{\mathrm{mail}}, p_{\mathrm{e}}] \rightarrow$ 

$[\mathit{mail}, \mathit{spam}, p_{\mathrm{e}}, p_{\mathrm{spam}}] \rightarrow$ 

$[\mathit{mail}, \mathit{spam}, p_{\mathrm{mail}}, p_{\mathrm{spam}}] \rightarrow$ 

Theorem 7 provides us with the technical background to prove our main theorem:

Theorem 8 Type checking for a b-mtt M can be done in time $\mathcal{O}(N^b \cdot n^{b\cdot(k+1+d)})$ where N is the size of the mtt, k is the maximal number of accumulating parameters, d is the maximal rank of an input symbol and n is the size of a dfta for the output type.

5 Macro Forest Transducers

Macro tree transducers have the disadvantage that they do not operate on forests directly but refer to representations of forests through ranked trees. This limitation, though, can be lifted. In [25], we have proposed macro forest transducers (mfts) which operate on forests directly. Mfts generalize mtts by providing concatenation as additional operation on output forests. This extra feature implies that some mft translations cannot be realized by a single mtt alone but only by the composition of a mtt with the transformation *app* from Section 3 [25]. Our transformation of the mailbox, for example, can be represented by a forest transducer as follows:

$$
\begin{array}{lll}
1 & \mathit{init}(\mathsf{doc}\langle x_1\rangle x_2) & \to \mathsf{doc}\langle \mathit{mbox}(x_1)\rangle \\
2 & \mathit{mbox}(\mathsf{mbox}\langle x_1\rangle x_2) & \to \mathsf{mbox}\langle \mathit{mail}(x_1)\rangle \\
 & & \quad \mathit{trashinit}(x_2, \mathit{spam}(x_1)) \\
3 & \mathit{trashinit}(\mathsf{trash}\langle x_1\rangle x_2, y_1) & \to \mathsf{trash}\langle \mathit{copy}_1(x_1)\ y_1\rangle \\
4 & \mathit{mail}(\mathsf{mail}\langle x_1\rangle x_2) & \to \mathsf{mail}\langle \mathit{copy}(x_1)\rangle \mathit{mail}(x_2) \\
5 & \mathit{mail}(\mathsf{spam}\langle x_1\rangle x_2) & \to \mathit{mail}(x_2) \\
6 & \mathit{spam}(\mathsf{mail}\langle x_1\rangle x_2) & \to \mathit{spam}(x_2) \\
7 & \mathit{spam}(\mathsf{spam}\langle x_1\rangle x_2) & \to \mathsf{spam}\langle \mathit{copy}_0(x_1)\rangle \mathit{spam}(x_2)
\end{array}
$$

$$
\begin{array}{lll}
8 & \mathit{copy}_1(\mathsf{mail}\langle x_1\rangle x_2) & \to \mathsf{mail}\langle \mathit{copy}(x_1)\rangle\ \mathit{copy}_1(x_2) \\
9 & \mathit{copy}_1(\mathsf{spam}\langle x_1\rangle x_2) & \to \mathsf{spam}\langle \mathit{copy}_0(x_1)\rangle\ \mathit{copy}_1(x_2) \\
10 & \mathit{copy}_0(\mathsf{mail}\langle x_1\rangle x_2) & \to \mathsf{mail}\langle \mathit{copy}(x_1)\rangle\ \mathit{copy}_0(x_2)
\end{array}
$$

where again the function *copy* is responsible for correctly copying the content of mail elements into the output. The functions *mail*, *spam*, copy_1, and copy_0 additionally have a rule translating e to e. Note that in order to have a more compact notation, we have represented elements by the label of their root, followed with the content in brackets. Due to concatenation of output forests in lines 2, 3, 4 and 7, an accumulating parameter is only used by the function *trashinit* for transporting the spam mails extracted from the mbox element, into the content of the right sibling of mbox.

Formally, a *macro forest transducer* M (mft for short) is similarly defined as an mtt. Now, however, rules are of the form:

$$
\begin{array}{r}
q(\mathsf{a}\langle x_1\rangle x_2, y_1, \ldots, y_k) \to f \quad \text{or} \\
q(\mathsf{e}, y_1, \ldots, y_k) \to f
\end{array}
$$

where right-hand sides f now are expression forests which can be constructed according to the following grammar:

$$
f \quad ::= \quad \mathsf{e} \mid \mathsf{b}\langle f\rangle \mid q'(x_i, f_1, \ldots, f_m) \mid y_j \mid f_1 f_2
$$

Here, b is a label of a node in the output, y_j is one of the accumulating parameters from the left-hand side, q' is a function name, and x_i is one of the input variables of the left-hand side ($i = 1, 2$ if available at all).

Again, initial function symbols q_0 may not have accumulating parameters.

The evaluation of an mft begins at the root node of the leftmost tree in the input forest. Then it traverses the input forest as if it were a binary tree meaning that in every step, it may proceed either to the content of the current node or to its right context. It is only when producing output that it may refer to the new ability of concatenation. As for mtts, we adopt an *inside-out* evaluation strategy, i.e., call-by-value passing of parameters. The only difference, thus, compared to mtts is that we now additionally may concatenate forests in accumulating parameters as well as in the outputs of the transducer. Let $\mathcal{F}_\Sigma$ denote the set of all forests with node labels from Σ. Accordingly,

let $\mathcal{F}_\Sigma(Y)$ denote the set of all forests with node labels from Σ which additionally may contain (forest) variables from the set Y. The least fixpoint semantics for the mft M then assigns to every state q with $k \geq 0$ accumulating parameters a function:

$$[\![q]\!] \ : \ \mathcal{F}_\Sigma \to 2^{\mathcal{F}_\Sigma(Y)}$$

where $Y = \{y_1, \ldots, y_k\}$. The definition of the translation τ_M realized by the mft M then is analogous to the definition for mtts in Section 2.

In the following, we will not mention explicitly given input types in our theorems and proofs. Instead, we always implicitly assume that this type has been encoded into the mft. For that, we assume that the input type is specified by a (possibly nondeterministic) finite forest automaton (fta for short) which is essentially the same as a finite tree automaton running on the binary representation of the forest. This automaton then is simulated during transformation along the same lines as in Section 2. In the following, we therefore concentrate on the output languages of mfts.

6 Linear Mfts

The constructions which we have provided for describing or approximating the output languages of mtts naturally can be extended to mfts as well. The only property which we have to take care of is that the grammar notions are appropriately generalized to deal with concatenation of forests.

Thus, we introduce the concept of a *context-free forest grammar* G (cffg for short) as 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 alphabet of terminal nodes and P is a set of rules of the form $q(y_1, \ldots, y_k) \to f$ where $q \in E$ is a nonterminal of rank $k \geq 0$. The right-hand side t is built up from the empty forest and variables $y_1, \ldots, y_k$ by means of concatenation, application of nonterminal and terminal symbols. Note that this new grammar formalism can be considered as a generalization of Fischer's macro grammars [11] from strings to forests. As in the rest of the paper, we refer to the inside-out mode of evaluation of nested nonterminal occurrences.

Again, the cffg for our example mft is obtained by canceling the input:

$$\begin{array}{lcl}
\mathit{init} & \to & \mathsf{doc}\langle \mathit{mbox} \rangle \\
\mathit{mbox} & \to & \mathsf{mbox}\langle \mathit{mail} \rangle \mathit{trashinit}(\mathit{spam}) \\
\mathit{trashinit}(y_1) & \to & \mathsf{trash}\langle \mathit{copy}\ y_1 \rangle \\
\mathit{mail} & \to & \mathsf{mail}\langle \mathit{copy} \rangle \mathit{mail} \mid \mathit{mail} \mid \mathsf{e} \\
\mathit{spam} & \to & \mathit{spam} \mid \mathsf{spam}\langle \mathit{copy} \rangle \mathit{spam} \mid \mathsf{e}
\end{array}$$

The notion of linearity for mfts is completely analogously defined as linearity for mtt. We obtain:

Theorem 9 Consider an mft M. Then a cffg G_M can be constructed in linear time with the following properties:

1. $\tau_M(\mathcal{F}_\Sigma) \subseteq \mathcal{L}(G_M)$.

2. If M is linear, then $\tau_M(\mathcal{F}_\Sigma) = \mathcal{L}(G_M)$.

For Theorem 9 to be useful, we additionally verify that emptiness for cffgs is efficiently decidable and also that cffgs are closed under intersection with recognizable forest languages. While emptiness still can be decided with very much the same algorithm as for cftgs, also the construction of a grammar for the intersection works pretty much along the same lines. We have to take care, however, that our deterministic finite-state representation of the output type is compatible with concatenations.

Therefore we propose to use finite forest monoids (compare, e.g., the discussion in [2]). A *finite forest monoid* (ffm) consists of a finite monoid M with a neutral element e, a finite subset $F \subseteq M$ of accepting elements, and finally, a function $up : \Sigma \times M \to M$ mapping a symbol of Σ together with a monoid element for its content to a monoid element representing a forest of length 1.

Given a deterministic bottom-up tree automaton $A = (P, \Sigma, \delta, F_A)$, we can construct a finite forest monoid as follows. Let $M = P \to P$ be the monoid of functions from the set of automata states into itself 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:

$$up(a, f_1)(p) = \delta(a, f_1(\delta(\mathsf{e})), p)$$

Finally, the set of accepting elements is given by:

$$F = \{f \in M \mid f(\delta(\mathsf{e})) \in F_A\}$$

This construction shows that every recognizable forest language can be recognized by a finite forest monoid.

Although the ffm for a bottom-up tree automaton generally can be exponentially larger, this need not always be the case.

For our running example the monoid consists of the neutral element e together with the following elements:

doc, mbox, trash, mboxtrash, mail, spam, fail

Besides the compositions with e and those resulting in fail, we have mbox $\cdot$ trash $=$ mboxtrash, and moreover:

	mail	spam
mail	mail	spam
spam	spam	spam

We summarize our observations for cffgs in the next theorem:

Theorem 10 Assume G is a cffg.

1. It can be decided in linear time whether or not $\mathcal{L}(G) = \emptyset$.
2. For every ffm M, a cffg G_M can be constructed such that $\mathcal{L}(G_M) = \mathcal{L}(G) \cap \mathcal{L}(M)$. The grammar G_M can be constructed in time $\mathcal{O}(N \cdot n^{k+1+d})$ where N is the size of G, k is the maximal rank of a nonterminal of G, d is the maximal number of occurrences of a nonterminal in right-hand sides and n is the size of the finite forest monoid.

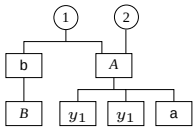


Figure 5: An example right-hand side of a ccffg.

Theorem 10 immediately gives us a first precise type checking result for linear mfts and an approximative type checking method for general mfts. Here, we only state the exact result:

Theorem 11 Type checking for a linear mft M can be done in time $\mathcal{O}(N \cdot n^{k+3})$ where N is the size of the mft, k is the maximal number of accumulating parameters, and n is the size of a ffm for the output type.

7 MFTs with bounded copying

We are now going to extend the methods from the last section to syntactically b-bounded copying mfts (b-mfts) for short. We deliberately omit the formal definition but appeal to the reader that it is completely analogous to the ranked tree case of mtts. In oder to describe the output languages of such transducers, we again glue together function calls which refer to the same node in the input forest.

For this purpose, we propose coupled context-free forest grammars (ccffgs). This concept is not so well-established in the literature. It is meant to be a generalization of context-free forest grammars where we allow nonterminals to have multiple roots. Graphically, an example of a right-hand side is shown in Figure 5. The idea is that every new right-hand side now conceptually consists of a tuple of forests which are glued together at nonterminals. In the example, the first output forest corresponds to the concatenation of the tree $\mathsf{b}\langle B\rangle$ and the forest returned by the first component of the nonterminal A while the second component of the nonterminal A provides the second output forest.

As before, we assume that every argument of a nonterminal is dedicated to a specific output of this nonterminal. Thus, the functionality of a nonterminal A is described by a sort $\langle k_1, \ldots, k_r\rangle$ meaning that the first k_1 arguments are dedicated to the first output and so forth. The linear representation of right-hand sides which we choose here is identical to the representation in the tree case:

$$\mathbf{let}\ \mathit{defs}\ \mathbf{in}\ (f_1, \ldots, f_r)$$

— with the only exception that the body of the **let** construct now consists of a tuple of forests which are built up from the empty forest and variables by application of components of nonterminals, terminal symbols and concatenation. The example from Figure 5 thus is represented by:

$$\begin{array}{ll}\mathbf{let} & (A_1, A_2) = A \\ \mathbf{in} & (\mathsf{b}\langle B\rangle A_1, A_2\langle y_1 y_1 \mathsf{a}\rangle)\end{array}$$

The least fixpoint semantics of a ccffg interprets every nonterminal q of sort $\langle k_1, \ldots, k_r \rangle$ as a set of forest tuples:

$$[\![A]\!] \subseteq \mathcal{F}_\Sigma(Y_1) \times \ldots \times \mathcal{F}_\Sigma(Y_r)$$

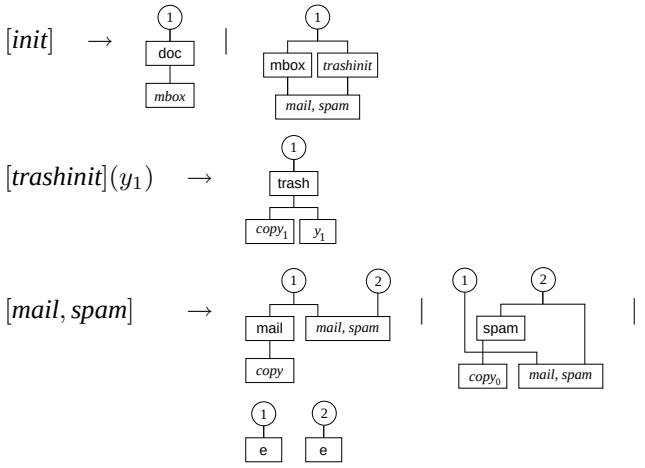
where $Y_j = \{y_1, \ldots, y_{k_j}\}$. Now consider again a b-bounded-copying mft M. Using the same gluing technique as for mtts, we obtain:

Theorem 12 For every b-mft M, there is a ccffg G_M with

$$\tau_M(\mathcal{F}_\Sigma) = \mathcal{L}(G_M)$$

The ccffg G_M can be constructed in time $\mathcal{O}(|M|^b)$.

Coming back to our example (cf. Section 2), we construct from the mft rules the following grammar:

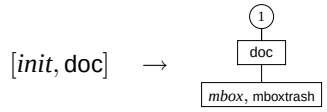


where the rules for the nonterminals $[\textit{copy}], [\textit{copy}_0]$ and $[copy_1]$ can be constructed straight-forwardly from the mft rules. The rules of the last line are for the new nonterminal $[\textit{mail}, \textit{spam}]$ indicating that the mft functions *mail* and *spam*, respectively, are applied on the same input variable.

As for linear mfts, the characterization of output languages by means of grammars is only useful if the grammar formalism is effectively closed under intersection with recognizable languages and each grammar can be tested for emptiness. Now in the same way how ccftgs cooperate nicely with deterministic ftas, our new grammar formalism works nicely together with finite forest monoids. Therefore, Theorem 7 literally holds if we replace "ccftg" with "ccffg" and "dfta" with "ffm", respectively.

Let us return to our running mailbox example together with the output type from Section 2. We present here only a manageable part of all productions of the intersection, but all omitted rules can be constructed in the same way. First we present the rules

constructed from the initial nonterminal *init*:



They are obtained from the ccffg rules by building all possible and consistent new right-hand sides in which every nonterminal is equipped with ffm states for the inputs as well as for the outputs. The new nonterminals consist of a nonterminal from the ccffg characterizing the output language and a tuple of sequences of monoid elements. Here, the nonterminal *init* is annotated with the state doc because this is the only consistent annotation of the hypergraph representation of the right-hand side.

Applying this technical background on ccffgs we arrive at our main theorem for macro forest transducers:

Theorem 13 Type checking for a b-mft M can be done in time $\mathcal{O}(N^b \cdot n^{b\cdot(k+3)})$ where N is the size of the mft, k is the maximal number of accumulating parameters, and n is the size of a finite forest monoid for the output type.

8 Conclusion

We have exhibited exact type-checking algorithms for useful classes of XML transformations based on a precise characterization of output languages. For our approach, the input type could always be described by a nondeterministic finite automaton. In order to obtain tractable algorithms, we assumed for macro tree transducers, that output types are given as *deterministic* finite automata, whereas for macro forest transducers, we even assumed legal outputs to be represented by finite forest monoids. The latter was necessary to elegantly cope with the extra ability of concatenating separately produced output forests. Besides exact but partial methods, we also provided approximate type-checking based on context-free tree grammars which is exponential only in the number of accumulating parameters. Note that this approach goes far beyond what is possible with approximations of outputs through recognizable sets.

All our techniques rely on a *inside-out* or call-by-value evaluation strategy for parameters. One may wonder in how far similar techniques may work a for *outside-in* or call-by-name evaluation corresponding to transformations expressed in (fragments of) lazy functional languages such as Haskell. While the approximate technique based on context-free tree grammars can readily be extended, it remains unclear whether similar constructions could provide exact techniques for useful subclasses of transducers as well.

References

[1] S. Boag and D. Chamberlin et.al., editors. XQuery 1.0: An XML Query Language. W3C Working Draft, World Wide Web Consortium, November 2003. Available online http://www.w3.org/TR/xquery/.

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

[3] J. Clark and S. DeRose, editors. XML Path Language (XPath) 1.0. W3C Recommendation, World Wide Web Consortium, November 1999. Available online http://www.w3.org/TR/xpath.

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

[5] J. Engelfriet. Context-Free Graph Grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 125–213. Springer-Verlag, Berlin, 1997.

[6] J. Engelfriet and S. Maneth. Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. *Inform. and Comput.*, 154(1):34–91, 1999.

[7] J. Engelfriet and S. Maneth. A Comparison of Pebble Tree Transducers with Macro Tree Transducers. *Acta Informatica*, 39:613–698, 2003.

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

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

[10] D.C. Fallside, editor. XML Schema. W3C Recommendation, World Wide Web Consortium, 2 May 2001. Available online http://www.w3.org/TR/xmlschema-0/.

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

[12] A. Frisch. Regular Tree Language Recognition with Static Information, 2004. PLAN-X 2004.

[13] H. Hosoya, A. Frisch, and G. Castagna. Parametric Polymorphism for XML. In *32nd ACM Symp. on Principles of Programming Langugaes (POPL)*, pages 50–62, 2005.

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

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

[16] C. Kirkegaard, A. Møller, and M.I. Schwartzbach. Static Analysis of XML Transformations in Java. *IEEE Transactions on Software Engineering*, 30:181–192, 2004.

[17] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML Type Checking with Macro Tree Transducers. In *24th Symp. on Principles of Database Systems (PODS)*, pages 283–294, 2005.

[18] W. Martens and F. Neven. Typechecking Top-Down Uniform Unranked Tree Transducers. In *9th Inter. Conference on Database Theory (ICDT)*, pages 64–78. LNCS 2572, 2002.

[19] W. Martens and F. Neven. Frontiers of Tractability for Typechecking Simple XML Transformations. In *23rd ACM Symp. on Principles of Database Systems (PODS)*, pages 23–34. ACM Press, 2004.

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

[21] A. Møller, M. Olesen, and M. Schwartzbach. Static Validation of XSL Transformations. Technical Report RS-05-32, BRICS, October 2005.

[22] A. Møller and M. I. Schwartzbach. The Design Space of Type Checkers for XML Transformation Languages. In *10th International Conference on Database Theory (ICDT)*, pages 17–36. LNCS 3363, 2005.

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

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

[25] T. Perst and H. Seidl. A Type-Safe Macro System for XML. In *Extreme Markup Languages*, Montréal, Quebec, 2002. Available online http://www.idealliance.org/papers/extreme02/.

[26] T. Perst and H. Seidl. Macro Forest Transducers. *Information Processing Letters*, 89:141–149, 2004.

[27] P. Réty, J. Chabin, and J. Chen. R-Unification thanks to Synchronized-Contextfree Languages. In *19th Workshop on Unification (UNIF)*, 2005.

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

[29] H. Seidl. Least Solutions of Equations over $\mathcal{N}$. In *Int. Coll. on Automata, Languages and Programming (ICALP)*, pages 400–411. Springer, LNCS 820, 1994.

[30] A. Tozawa. Towards Static Type Inference for XSLT. In *ACM Symp. on Document Engineering*, pages 18–27, 2001.

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