Exact XML Type Checking in Polynomial Time

Sebastian Maneth[1], Thomas Perst[2], and Helmut Seidl[2]

[1] National ICT Australia Ltd., Sydney, Australia
[2] Technische Universität München, Garching, Germany

Abstract. Stay macro tree transducers (smtts) are an expressive formalism for reasoning about XSLT-like document transformations. Here, we consider the exact type checking problem for smtts. While the problem is decidable, the involved technique of inverse type inference is known to have exponential worst-case complexity (already for top-down transformations without parameters). We present a new *adaptive* type checking algorithm based on forward type inference through exact characterizations of output languages. The new algorithm correctly type-checks all call-by-value smtts. Given that the output type is specified by a deterministic automaton, the algorithm is *polynomial-time* whenever the transducer uses only few parameters and visits every input node only constantly often. Our new approach can also be generalized from smtts to stay macro forest transducers which additionally support concatenation as built-in output operation.

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 [20] 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 [19]. 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.

In previous work [15], we have considered the transformation language TL which provides all standard features of top-down transformation languages such as named functions, MSO expressible match and select patterns as well as accumulating parameters. For this transformation language, we have provided an exact type checking algorithm by decomposing every such transformation into at most three simple transformations, namely, *stay macro tree transducers* (smtts) — independently of the match and select patterns used by the transformation. This work together with the results of [6] (showing that pebble tree transducers can be simulated by compositions of smtts) have established (compositions of) stay macro tree transducers as an adequate model for formally reasoning about XML transformations.

Even though the class of translations for which exact type checking is decidable is surprisingly large [6, 18, 15], 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 [23]. 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 [16]. 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 one stay macro tree transducer with few parameters which translates each node of the input tree at most once ("linear" smtts), or more generally, which translates every node only constantly often (*b-bounded copying* smtts). Note that no restriction is put on the copying that the smtt 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 necessary in practical implementations of type checking, because it can be used to deal with conditionals depending, e.g., on data value comparisons. Opposed to the techniques in [15], our new approach is based on *forward type inference*. In detail, our contributions are the following. First, we generalize the well-known triple construction for context-free grammars to provide a general construction for smtts to produce only output trees from the language accepted by some deterministic finite automaton. Secondly, we use stay moves to cut down the numbers of function calls in right-hand sides which crucially affect the complexity of the construction. Also, we present a formulation through Datalog to obtain a practically efficient implementation. Then we exhibit subclasses for which our approach to type checking is provably efficient and present an adaptive algorithm which is correct for arbitrary smtts but automatically meets the improved time bounds on the provably efficient sub-classes. Finally, the new approach is generalized from smtts to stay macro forest transducers which additionally provide built-in support for concatenation of forests.

Related Work Approximative type checking for XML transformations is typically based on (subclasses of) recognizable tree languages. Using XPathas pattern language, XQuery [1] is a functional language for querying XML documents. It is strongly-typed and type checking is performed via forward type inference rules computing approximative types for each expression. Approximative type inference is also used in XDuce [13] and its follow-up version CDuce [10]; 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 [12] for parametric polymorphism for XML [11]. Type variables are interpreted as markings indicating the parameterized subparts. In [19] a sound type checking algorithm is proposed (originally developed for the Java-based language XACT [14]) 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. [18]. There, *inverse* type inference is proposed for translations expressible as k-pebble tree transducers. Inverse type inference was also studied by Tozawa [27] for a subset of XSLT which roughly corresponds to top-down forest transducers (without parameters) [23], and it was studied in [15] for a much more expressive transformation language.

2 Stay 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:

```
<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>
```

This example represents a mail file, where the elements `mbox` and `trash` collect the incoming and deleted mails, respectively. Besides `mail` elements, the `mbox` also contains mails inside a `spam` element indicating that these mails have been identified as spam, e.g., by some automated filter.

Stay macro tree transducers do not operate on forests directly but on their representations as binary trees. The empty forest then is represented by 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: $t \quad ::= \quad \mathsf{b} \quad | \quad \mathsf{a}(t_1, \ldots, t_n)$ where $\mathsf{b}, \mathsf{a} \in \Sigma$ are symbols of rank zero and n, respectively; thus, we assume that we are given a fixed rank for every element of Σ. Often, we consider constructor applications together with

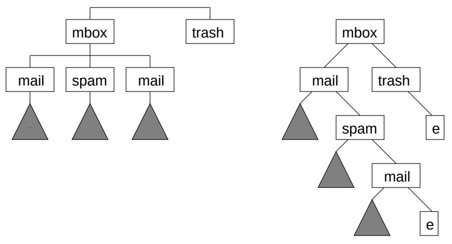

Fig. 1. An unranked forest and its binary encoding.

leaf nodes by allowing n to equal 0. For a set $Y = \{y_1, y_2, \dots\}$ of variables of rank 0, $\mathcal{T}_\Sigma(Y)$ 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 by spam into trash, while leaving all mail elements untouched. For our example document, the transformation produces:

```
<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>
```

Using our representation of forests by binary trees (Fig. 1), this transformation is realized by a tree transducer with the following rules:

$$\begin{array}{lll} 1 & q(\mathsf{mbox}(x_1, x_2)) & \rightarrow \mathsf{mbox}(q_1(x_1), p(x_2, q_2(x_1))) \\ 2 & p(\mathsf{trash}(x_1, x_2), y_1) & \rightarrow \mathsf{trash}(app(x_1, y_1), \mathsf{e}), \end{array}$$

together with a function q_1 for collecting all ordinary mails in mbox

$$\begin{array}{lll} 3 & q_1(\mathsf{mail}(x_1, x_2)) & \rightarrow \mathsf{mail}(cp(x_1), q_1(x_2)) \\ 4 & q_1(\mathsf{spam}(x_1, x_2)) & \rightarrow q_1(x_2) \\ 5 & q_1(\mathsf{e}) & \rightarrow \mathsf{e}, \end{array}$$

as well as a function q_2 for collecting the spam mails in mbox

$$\begin{array}{lll} 6 & q_2(\mathsf{mail}(x_1, x_2)) & \rightarrow q_2(x_2) \\ 7 & q_2(\mathsf{spam}(x_1, x_2)) & \rightarrow \mathsf{spam}(cp(x_1), q_2(x_2)) \\ 8 & q_2(\mathsf{e}) & \rightarrow \mathsf{e} \end{array}$$

where function *app* in line 2 is meant to copy the content of trash in front of the accumulating parameter, here containing the spam elements collected by the call q_2. Likewise, function *cp* is meant to produce an exact copy of its input.

Formally, a *stay macro tree transducer* M (smtt for short) is a tuple (Q, Σ, R, Q_0) where Q is the (ranked) set of function names or states, Σ is the (ranked) alphabet of input and output symbols, $Q_0 \subseteq Q$ is the set of initial functions, and R is a finite set of rules of the form

$$q(x_0, y_1, \ldots, y_k) \to t \qquad \text{or} \qquad q(\mathsf{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t'$$

where $q \in Q$ is of rank k, $\mathsf{a} \in \Sigma$ is of rank n, $x_0, x_1, \ldots, x_n$ are input variables, $y_1, \ldots, y_k, k \geq 0$ are the accumulating parameters of q, and t, t' are expressions describing the output actions of the corresponding rule. Possible actions are:

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

where b is a label of an output node, 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 of the left-hand side. 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. The rules which do not process input symbols are called *stay*-rules. Transducers without stay-rules are also called (ordinary) mtts. Note that in [15], smtts are defined in a slightly different way. The rules there are of the form $q(x_0 \text{ as } \mathsf{a}(x_1, \ldots, x_n), \ldots) \to t$ where variable x_0 is bound to the current node and x_0 as well as $x_1, \ldots, x_n$ can occur in the right-hand side t. In our nondeterministic setting both formats can be converted into each other by means of a polynomial algorithm.

Intuitively, the meaning of 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 or on the current input node itself. Thus, the transformation of an smtt M starts at the root node of the input with one of the initial functions. A function q with actual accumulating parameters $t_1, \ldots, t_k$ is applied to an input subtree $s = \mathsf{a}(s_1, \ldots, s_n)$ as follows. If a stay rule $q(x_0, y_1, \ldots, y_k) \to t$ is chosen for q, s and the t_j are substituted in t for x_0 and the variables y_j, respectively. If an ordinary rule $q(\mathsf{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t'$ for q is chosen, the subterms s_i and t_j are substituted in t' for the variables x_i and y_j, respectively. Since function calls may be nested, the order in which they are evaluated matters. In *outside-in* (OI) or *call-by-name* evaluation order, outermost calls are evaluated first. The parameters of a function call may themselves contain function calls which then are transferred to the body in an unevaluated form [8]. In this paper, however, we consider the *inside-out* (IO) evaluation order. This order corresponds to *call-by-value* parameter passing as provided by mainstream programming languages like C or OCaml. The inside-out strategy evaluates innermost calls first, meaning that fully evaluated output trees are passed in accumulating parameters.

As in [23], we will not use an operational semantics of smtts based on rewriting, but prefer a denotational formulation which greatly simplifies proof arguments. Thus, the meaning $[\![q]\!]$ of state q of M with k accumulating parameters is defined as a function from input trees to sets of trees with parameters in $Y = \{y_1, \ldots, y_k\}$, i.e., $[\![q]\!] : \mathcal{T}_\Sigma \to$

$2^{\mathcal{T}_\Sigma(Y)}$. 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:

$$\begin{array}{lll} [\![q]\!](s) & \supseteq [\![t[s/x_0]]\!] & \text{for rule} \quad q(x_0, \boldsymbol{y}) \to t \\ [\![q]\!](\mathsf{a}(s_1, \ldots, s_d)) & \supseteq [\![t'[s_1/x_1, \ldots, s_d/x_d]]\!] & \text{for rule} \quad q(\mathsf{a}(x_1, \ldots, x_d), \boldsymbol{y}) \to t' \end{array}$$

where $\boldsymbol{y}$ denotes the sequence $y_1, \ldots, y_k$ and

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

Here, t''/z denotes the substitution of the tree t'' for all occurrences of the variable z. 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 τ_M realized by the smtt M on a input tree s and sets S of input trees, respectively, is thus defined by:

$$\tau_M(s) = \bigcup\{[\![q_0]\!](s) \mid q_0 \in Q_0\} \quad \text{and} \quad \tau_M(S) = \bigcup\{\tau_M(s) \mid s \in S\}.$$

3 General Properties of SMTTs

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) [28] or a schema [9, 3].

A convenient abstraction of the existing XML type formalisms, in particular DTDs, are recognizable (or: regular) tree languages [21, 22]. 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 $(p, \mathsf{a}, p_1 \ldots p_k)$ denotes that if A arrives in state p_i after processing the tree t_i, then it can assign state p to the tree $\mathsf{a}(t_1, \ldots, t_k)$. 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 $r(v) \in P$ w.r.t. 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 $P = \{p_{\text{mbox}}, p_{\mathsf{e}}, p_{\text{mail}}, p_{\text{trash}}, p_{\text{spam}}, p_{\text{content}}, \ldots\}$ and set of transitions:

$$\begin{array}{llll} \delta = \{ & (p_{\text{mbox}}, \mathsf{mbox}, p_{\text{spam}} p_{\text{trash}}), & (p_{\mathsf{e}}, \mathsf{e}), & \\ & (p_{\text{spam}}, \mathsf{mail}, p_{\text{content}} p_{\text{spam}}), & (p_{\text{spam}}, \mathsf{mail}, p_{\text{content}} p_{\mathsf{e}}), & \\ & (p_{\text{spam}}, \mathsf{spam}, p_{\text{mail}} p_{\text{spam}}), & (p_{\text{spam}}, \mathsf{spam}, p_{\text{mail}} p_{\mathsf{e}}), & \\ & (p_{\text{mail}}, \mathsf{mail}, p_{\text{content}} p_{\text{mail}}), & (p_{\text{mail}}, \mathsf{mail}, p_{\text{content}} p_{\mathsf{e}}), & \\ & (p_{\text{trash}}, \mathsf{trash}, p_{\text{spam}} p_{\mathsf{e}}), & (p_{\text{mbox}}, \mathsf{mbox}, p_{\mathsf{e}} p_{\text{trash}}), \ldots & \}, \end{array}$$

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, mbox contains a possibly empty sequence of mail and spam elements where every spam element contains one mail element.

In the following, we will not mention explicitly given input types in our theorems. Instead, we implicitly assume that this type has been encoded into the smtt. This can be done as follows. Assume that the input type S is given by a (possibly nondeterministic) finite tree automaton A. From an smtt M, we then build a new smtt M_A whose function symbols are pairs consisting of a function of M and an automaton state of A. E.g., from a rule $q(a(x_1, x_2), y_1) \to b(q_1(x_1, y_1), q_2(x_2, y_1))$ we obtain the following new rule

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

if $(p, a, p_1 p_2)$ is a transition of A. Thus, 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. In order to deal with variables x_i not occurring in the right-hand side, we introduce extra functions $\mathsf{check}_{p'}$ for every state p' of A such that $\mathsf{check}_{p'}(s, y_1)$ produces y_1 iff there is a run of A on s resulting in state p'. 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 smtt M_A is of size $\mathcal{O}(|M| \cdot |A|)$. Since the construction of M_A does not add new function calls in rules of M, M_A is linear in $x_0, x_1, \ldots$ if M is, and M_A is syntactically b-bounded if M is, cf. Sections 4 and 5.

As usual, the *size* $|M|$ of an smtt 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. The most basic problem for a given smtt M is to decide whether or not the translation of M is non-empty. For this problem we recall:

Theorem 1. *Deciding whether $\tau_M \neq \emptyset$ for an smtt M is DEXPTIME-complete.*

Proof. The lower bound follows since translation non-emptiness is DEXPTIME-hard already in absence of parameters, i.e., for top-down tree transducers [26].

Since we will heavily rely on this algorithm, we briefly sketch the construction for the upper bound. Assume that $M = (Q, \Sigma, R, Q_0)$ and, w.l.o.g., that for every state q of M with k accumulating parameters there is a rule $q(x_0, y_1, \ldots, y_k) \to q(x_0, y_1, \ldots, y_k)$. These rules will be used when checking the nonemptiness of several states simultaneously where for some states stay moves are selected.

For every subset $B \subseteq Q$, we introduce a propositional variable $[B]$ where $[B] = \mathsf{true}$ denotes the fact that $\exists\, t \in \mathcal{T}_\Sigma\ \forall\, q \in B :\ [\![q]\!](t) \neq \emptyset$. In particular, for the empty set we have the fact $[\emptyset] \Leftarrow \mathsf{true}$. We then consider the set of all propositional implications

$$[B] \ \Leftarrow\ [B_1]$$

for all selections of rules $q(x_0, y_1, \ldots, y_k) \to t_q\ \in\ R$, $q \in B$, with $B_1 = \{p \in Q \mid \exists\, q \in B :\ p(x_0, \ldots)$ occurs in $t_q\}$ as well as all implications

$$[B] \ \Leftarrow\ [B_1] \ \wedge\ \ldots\ \wedge\ [B_n]$$

for all selections of rules $q(\mathsf{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t'_q \;\in\; R$, $q \in B$, for the same a where $B_i = \{p \in Q \mid \exists\, q \in B : \; p(x_i, \ldots)$ occurs in $t'_q\}$.

Let $\mathcal{C}_M$ denote this system of implications. By construction, the size of $\mathcal{C}_M$ is exponential in the size of M. Moreover, the translation of M is nonempty iff, for some $q \in Q_0$, $[\{q\}] = \mathsf{true}$ follows from $\mathcal{C}_M$. Since systems of propositional Horn clauses can be solved in linear time, the assertion follows. $\square$

Next, we show how to effectively restrict a given smtt so that it only produces output trees in a given recognizable output tree language. In fact, the corresponding construction is a straightforward generalization of the triple construction known for context-free grammars. In case of smtts, the construction is simpler if we additionally assume that the recognizable tree language, is given by a *deterministic* finite tree automaton.

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 an 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 practice, however, they are usually not much larger than corresponding nondeterministic ones.

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

e	
	p_{e}

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}

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. The transitions for mail and spam are defined as:

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}

Our key theorem is:

Theorem 2. *Assume M is an smtt and A is a dfta. Then there is an smtt M_A with*

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

for all $t \in \mathcal{T}_\Sigma$. The smtt M_A can be constructed in time $\mathcal{O}(N \cdot n^{k+1+d})$ where N is the size of M, k is the maximal number of accumulating parameters of a function symbol of M, d is the maximal number of function occurrences in any right-hand side, and n is the size of the finite tree automaton A.

Proof. Let $M = (Q, \Sigma, R, Q_0)$ and $A = (P, \Sigma, \delta, P_F)$. The set of function symbols of the new smtt M_A consists of all tuples $\langle q, p_0 p_1 \dots p_k \rangle$ where $q \in Q$ is a function symbol of the original smtt of rank k and $p_0, \dots, p_k \in P$ are states of the dfta A. The new function symbol $\langle q, p_0 \dots p_k \rangle$ is meant to generate all trees t with variables from $y_1, \dots, y_k$ for which there is a run of A starting at the leaves y_i with states p_i and reaching the root of t in state p_0. Therefore, the intersection smtt M_A has the rules:

$$\langle q, p_0 \dots p_k \rangle (r, y_1, \dots, y_k) \ \rightarrow \ t'$$

for every rule $q(r, y_1, \dots, y_k) \rightarrow t$ of M with either $r = x_1$ or $r = \mathsf{a}(x_1, \dots, x_d)$, and $t' \in \mathcal{T}^{p_0 \cdots p_k}[t]$; the sets $\mathcal{T}^{p_0 \cdots p_k}[t]$ are inductively defined by:

$$\begin{array}{ll}
\mathcal{T}^{p_i p_1 \dots p_k}[y_i] & = \{y_i\} \\
\mathcal{T}^{p_0 p_1 \dots p_k}[a(t_1, \dots, t_m)] & = \{a(t'_1, \dots, t'_m) \mid \delta_a(p'_1, \dots, p'_m) = p_0 \wedge \\
& \qquad\qquad\qquad \forall\, i :\ t'_i \in \mathcal{T}^{p'_i p_1 \dots p_k}[t_i] \ \} \\
\mathcal{T}^{p_0 p_1 \dots p_k}[q'(x_j, t_1, \dots, t_m)] & = \{ \langle q', p_0 p'_1 \dots p'_m \rangle (x_j, t'_1, \dots, t'_m) \mid \\
& \qquad\qquad\qquad \forall\, i :\ t'_i \in \mathcal{T}^{p'_i p_1 \dots p_k}[t_i] \}
\end{array}$$

By fixpoint induction, we verify for every state q of rank $k \geq 0$, every input tree $s \in \mathcal{T}_\Sigma$ and states $p_0, \dots, p_k$ that:

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

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

$$\begin{array}{ll}
\delta^*(y_i, \underline{p}) & = p_i \\
\delta^*(a(t_1, \dots, t_m), \underline{p}) & = \delta_a(\delta^*(t_1, \underline{p}), \dots, \delta^*(t_m, \underline{p}))
\end{array}$$

The set of new initial function symbols then consists of all $\langle q_0, p_f \rangle$ where $q_0 \in Q_0$ and p_f is an accepting state of A. Then the correctness of the construction follows from $(*)$.

For an smtt of size N with at most k parameters and at most d occurrences of states in right-hand sides, and a tree automaton with n states, the intersection grammar is of size $\mathcal{O}(N \cdot n^{k+1+d})$: there can be in the worst case n^{k+1} copies of a rule of the smtt M, and for each function occurrence in the right-hand side we may choose an arbitrary output states. This completes the proof. $\square$

Note that in general, the number d of occurrences of states in a right-hand side can be arbitrarily large. SMtts, however, allow a construction which cuts down the depth of right-hand sides to at most 2. We have:

Proposition 3. *For every smtt M, an smtt M' can be constructed with:*

1. *The translations of M and M' agree;*
2. *Whenever a right-hand side t of M' is not contained in $T_\Sigma(Y)$, then the depth of t is bounded by 2;*
3. *The maximal number of states in a right-hand side of M' is at most $k + 1$;*
4. *The size of M' is bounded by $\mathcal{O}(|M| \cdot k^2)$*

where k is the maximum of the maximal rank of output symbols and the maximal number of accumulating parameters of a state of M.

The idea of Proposition 3 is to split the right-hand sides into their subterms and to organize the execution by stay-rules. In this way, for every internal (i.e., non-root and non-leaf) node (of rank r) in the right-hand side of a rule of M, the transducer M' has a new state of rank r. Clearly, r is bounded by the maximum rank of states and output symbols of M. Moreover, if the corresponding left-hand side of M is a state with m parameters, then each new state also has rank m. This means that m parameters are passed in each of the new rules, which explains the size increase of at most k^2. Note that, given some input tree s, if there is a computation of M using n sequential rule applications (in the conventional term rewriting sense), then there is a corresponding computation of M' with at most $c \cdot n$ rule applications, where c is the size of the largest right-hand side of the rules of M.

4 Linear SMTTs

In this section we prove that type checking is in PTIME for smtts with a bounded number of parameters which process every node of the input tree at most once. Syntactically, the latter can be guaranteed by requiring that in every right-hand side, each input variable x_i occurs at most once. Mtts satisfying this restriction are called *linear* [8].

Note that linearity for an smtt 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 smtts 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). As an example of a linear smtt consider the smtt q_1 (lines 3-5 in our mail transformation). The grammar characterizing q_1's output language looks as follows:

$$q_1 \to \mathsf{mail}(cp, q_1) \mid q_1 \mid \mathsf{e}$$

where q_1, cp are nonterminals. Note that selection of rules depending on input symbols now is replaced with nondeterministic choice.

Context-free tree grammars were invented in the 70s [24]. See [7] 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 smtts, inside-out (IO) and outside-in evaluation order for nonterminal symbols must be distinguished [7]. 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 smtts — 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 $\mathcal{L}(G) = \bigcup\{[\![q_0]\!] \mid q_0 \in E_0\}$.

By Corollary 5.7 of [8], the output language of a linear smtt M can be characterized by a cftg G_M which can be constructed from M in linear time. During this construction every rule $q(\pi, y_1, \ldots, y_k) \rightarrow t$ (π is either $\mathsf{a}(x_1, \ldots, x_n)$ or x_0) is rewritten as a production $q(y_1, \ldots, y_k) \rightarrow t'$, where t' is obtained from t by deleting all occurrences of input variables x_i. A formal proof that G_M indeed characterizes the output language of M can be found, e.g., in [8].

The characterization of smtt output languages by cftgs is useful because emptiness for (IO-)cftgs is decidable using a similar algorithm as the one for ordinary context-free (word) grammars, see, e.g., [4]. Thus we have:

Theorem 4. *It can be decided in linear time for a cftg G whether or not $\mathcal{L}(G) = \emptyset$.*

In general, we are interested in type checking transformations implemented through smtts. Assume that we have coded the input type specification into the smtt M (cf. the discussion above Theorem 1). Assume further that we are given a dfta A describing (the complement of) the desired output type. **Type checking for** M means to test whether any output of M is accepted by A. If M is linear, then the corresponding intersection smtt M_A is again linear — meaning that its range can be described by a cftg (thus generating all "illegal outputs" of M wrt A). Therefore, Theorem 4 gives us:

Theorem 5. *Type checking for a linear smtt M can be done in time $\mathcal{O}(N \cdot n^{k+1+d})$ where N is the size of the smtt, 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.*

The complexity bound provided for the construction of Theorem 5 is a worst-case estimation. Instead, we want to point out that in case of linear smtts, the triple construction for M_A can be organized in such a way that only "useful" functions are constructed. In order to see this, we introduce for every q of M of rank k, a predicate $q/(k+1)$. Every rule $q(\_, y_1, \ldots, y_k) \rightarrow t$ of M then gives rise to the Datalog implication:

$$q(Y_0, \ldots, Y_k) \;\Leftarrow\; \mathcal{D}[t]_{Y_0}$$

where $\mathcal{D}[t]_X$ (X a variable) is defined by

$$\begin{array}{ll}
\mathcal{D}[y_i]_X & = X = Y_i \\
\mathcal{D}[\mathsf{a}(t_1, \ldots, t_m)]_X & = \delta(X, \mathsf{a}, X_1, \ldots, X_m) \;\wedge\; \mathcal{D}[t_1]_{X_1} \wedge \ldots \wedge \mathcal{D}[t_m]_{X_m} \\
\mathcal{D}[q'(t_1, \ldots, t_m)]_X & = q'(X, X_1, \ldots, X_m) \;\wedge\; \mathcal{D}[t_1]_{X_1} \wedge \ldots \wedge \mathcal{D}[t_m]_{X_m}
\end{array}$$

and the variables $X_1, \ldots, X_m$ in the last two rows are fresh. For subsets $X, X_1, \ldots, X_k$ of the set of states of A, $\delta(X, \mathsf{a}, X_1, \ldots, X_k)$ denotes the fact that $(x, \mathsf{a}, x_1, \ldots, x_k)$ for all $x \in X$ and $x_i \in X_i$, $i = 1, \ldots, k$. A bottom-up evaluation of the resulting program computes for every $q/(k+1)$, the set of all tuples $(p_0, \ldots, p_k)$ such that the translation of $\langle q, p_0 \ldots p_k \rangle$ is non-empty. If we additionally want to restrict these predicates only to tuples which may contribute to a terminal derivation of the initial nonterminal $\langle q_0, p_f \rangle$, we may top-down query the program with $\Leftarrow q_0(p_f)$. 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.

The algorithm in the proof of Theorem 5 can also be applied to *non-linear* smtts. Then, the constructed Datalog program does no longer precisely characterize the non-empty functions of the intersection smtt because dependencies on input subtrees (viz. several function calls on the same input variable x_i) have been lost. Rather, a *superset* is returned. By means of cftgs, we can express this observation as follows:

Theorem 6. *Let G_M be the cftg constructed for an smtt 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 smtt rule exactly one production of the cftg G_M and obtain:

$$\begin{array}{llll} q & \to \mathsf{mbox}(q_1, p(q_2)) & p(y_1) & \to \mathsf{trash}(app(y_1), \mathsf{e}) \\ q_1 & \to \mathsf{mail}(cp, q_1) \mid q_1 \mid \mathsf{e} & q_2 & \to \mathsf{spam}(cp, q_2) \mid q_2 \mid \mathsf{e} \end{array}$$

where the two function calls q_1 and q_2 on the same input variable x_1 are simply represented by the two nonterminals q_1 and q_2, respectively.

Note that when approximating the output languages of general smtts 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 Proposition 3 to limit the maximal number of occurrences of nonterminals in each right-hand side to a number k which is the maximum of the maximal rank of input symbols and the maximal number of parameters.

5 SMTTs 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 smtts. The goal again is to find precise and tractable characterizations of the output language. If the smtt 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. Quite a few useful transformations on the other hand consult every part of the input only a small number of times [25]. In our running example with mail and spam, every subtree of the input is processed at most twice. Therefore, we consider the subclass of smtts processing every subtree of the input at most b times. Thus in principle, b-bounded copying is a semantic property (cf. [17] and [5]).

Instead of dealing with a semantic definition, we find it more convenient to consider syntactic b-bounded copying only. Assume therefore w.l.o.g. that every state of M is *syntactically productive*, i.e., can produce at least one tree w.r.t. the cftg approximating M. 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 natural numbers 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, $q_1, \ldots, q_m$ 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 constraints for stay-rules are constructed analogously. Let $[q]$, q state of M, denote the least solution of this system. Then the smtt M is *syntactically b-bounded* (*copying*) (or, a b-smtt for short) iff $[q] \leq b$ for all states of M. For the case where every input node is visited only a *small* number of times, we have:

Theorem 7. *For every syntactically b-bounded smtt M the following holds:*

1. *For every dfta A, the intersection smtt M_A is again syntactically b-bounded.*
2. *Translation emptiness can be decided in time $\mathcal{O}(|M|^b)$.*

Proof. For the first assertion, we claim that for every state q of M with k parameters, $b[q] \ \geq \ b[\langle q, p_0 \ldots p_k \rangle]$ for every sequence $p_0, \ldots, p_k$ of dfta states where $\langle q, p_0 \ldots p_k \rangle$ is syntactically productive. This claim is easily verified by fixpoint induction w.r.t. the corresponding constraint systems characterizing $b[q]$ and $b[\langle q, p_0 \ldots p_k \rangle]$, respectively.

For a proof of the second assertion, we observe that, for syntactically b-bounded smtts, the propositional variables $[\{q\}], q \in Q_0$, only depend on propositional variables $[B]$ for sets of states B of cardinality at most b.

Theorem 2 provides us with the technical background to prove our main theorem:

Theorem 8. *Type checking for a b-smtt M can be done in time $\mathcal{O}(N^b \cdot n^{b \cdot (k+1+d)})$ where N is the size of the smtt, 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.*

Instead of first testing b-boundedness and then running a specialized algorithm, we definitely prefer to have a general purpose algorithm which is correct for all smtts but additionally will meet the better complexity bounds on the exhibited subclasses. Indeed, our methods can be combined to construct such an adaptive algorithm. Given an smtt M and a dfta A, we proceed as follows:

1. For M, we compute an equivalent smtt M' where the numbers of occurrences of states in right-hand sides are bounded.
2. For M', we compute a safe superset of the states of the intersection smtt M'_A by means of top-down solving the corresponding Datalog program.
3. If no accepting states of A are found for the predicates $q_0/1$, q_0 initial state of M', the intersection is definitely empty, and we return.
   Otherwise, we precisely check the intersection smtt M'_A for emptiness through locally solving the corresponding system of propositional Horn clauses.

In the worst case, this algorithm will be exponential in the number of states of M and doubly exponential in the number of parameters and, due to the lower bounds for translation emptiness, nothing better can be hoped for. If on the other hand, the smtt is linear or syntactically b-bounded, the algorithm's complexity achieves the upper bounds of Theorems 5 and 8, respectively. Even more, due to the demand-driven algorithms in steps 2 and 3, we obtain very good running times on practical examples.

6 Macro Forest Transducers

Macro tree transducers have the disadvantage that they do not operate on forests directly. In [23], this limitation is lifted. Thus, *stay macro forest transducers* (smfts) generalize smtts by providing concatenation as additional operation on output forests. Although, smfts are more expressive than smtts, we obtain closure under intersection with recognizable forest languages also for output languages of smfts. This result is again based on a generalized triple construction. This time, however, we additionally must take care that our deterministic finite-state representation of the output type is compatible with concatenations. Therefore, we replace the concept of a dfta by a *finite forest monoid* (ffm) which is a finite monoid S extended with an operation up : $S \times \Sigma \to S$ that is used to handle upward movement in the forest (cf. [2]).

Since the notion of linearity for smfts is completely analogous to linearity for smtts, the type checking algorithm for a linear smft T is almost the same as for linear smtts. As in the ranked tree case, we can also extend the methods to syntactically b-bounded copying smfts (b-mfts) and obtain as our main result for smfts:

Theorem 9. *Type checking for a b-smft M can be done in time $\mathcal{O}(N^b \cdot n^{b\cdot(k+3)})$ where N is the size of the smft, k is the maximal number of accumulating parameters, and n is the size of a ffm for the output type.*

7 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 methods, we also provided approximate type checking based on context-free tree grammars. Finally, we combined our techniques to a simple adaptive algorithm which is provably efficient on the exhibited subclasses but may be promising also in other practical contexts. In case sets of possibly illegal outputs are described by cf tree grammars, we can also check in PTIME whether only *finitely many* illegal outputs may occur. This is called "almost always type checking" in [6]. It remains open in how far these techniques can be applied to smtts with outside-in (call-by-name) evaluation order.

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 M. Murata et al. *RelaxNG Specification*. OASIS. Available online http://www.oasis-open.org/committees/relax-ng.
4. J. Engelfriet. Context-Free Graph Grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 125–213. Springer-Verlag, Berlin, 1997.
5. J. Engelfriet and S. Maneth. Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. *Inform. and Comput.*, 154(1):34–91, 1999.
6. J. Engelfriet and S. Maneth. A Comparison of Pebble Tree Transducers with Macro Tree Transducers. *Acta Informatica*, 39:613–698, 2003.
7. J. Engelfriet and E.M. Schmidt. IO and OI. (I&II). *J. Comp. Syst. Sci.*, 15:328–353, 1977. and 16:67–99, 1978.
8. J. Engelfriet and H. Vogler. Macro Tree Transducers. *J. Comp. Syst. Sci.*, 31:71–146, 1985.
9. D.C. Fallside, editor. XML Schema. W3C Recommendation, World Wide Web Consortium, 2 May 2001. Available online http://www.w3.org/TR/xmlschema-0/.
10. A. Frisch. Regular Tree Language Recognition with Static Information. PLAN-X 2004.
11. 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.
12. H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6):961–1004, 2002.
13. H. Hosoya and B.C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.
14. C. Kirkegaard, A. Møller, and M.I. Schwartzbach. Static Analysis of XML Transformations in Java. *IEEE Transactions on Software Engineering*, 30:181–192, 2004.
15. 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.
16. 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.
17. 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.
18. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. *J. Comp. Syst. Sci.*, 66:66–97, 2003.
19. A. Møller, M. Olesen, and M. Schwartzbach. Static Validation of XSL Transformations. Technical Report RS-05-32, BRICS, October 2005.
20. 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.
21. M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages 2000*, 2000.
22. F. Neven. Automata Theory for XML Researchers. *SIGMOD Record*, 31(3):39–46, 2002.
23. T. Perst and H. Seidl. Macro Forest Transducers. *Information Processing Letters*, 89:141–149, 2004.
24. W.C. Rounds. Mappings and Grammars on Trees. *Math. Systems Theory*, 4:257–287, 1970.
25. Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *28th Int. Conf. on Very Large Data Bases (VLDB)*, pages 974–985. Morgan Kaufmann, 2002.
26. H. Seidl. Haskell Overloading is DEXPTIME Complete. *Information Processing Letters (IPL)*, 52:57–60, 1994.
27. A. Tozawa. Towards Static Type Inference for XSLT. In *ACM Symp. on Document Engineering*, pages 18–27, 2001.
28. W3C. *Extensible Markup Language (XML) 1.0*, second edition, 6 October 2000. Available online http://www.w3.org/TR/2000/REC-xml-20001006.