XML Type Checking with Macro Tree Transducers

S. Maneth
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
sebastian.maneth@epfl.ch

A. Berlea, T. Perst, H. Seidl
Technische Universität München
85748 Garching, Germany
{berlea,perst,seidl}@in.tum.de

ABSTRACT

MSO logic on unranked trees has been identified as a convenient theoretical framework for reasoning about expressiveness and implementations of practical XML query languages. As a corresponding theoretical foundation of XML transformation languages, the "transformation language" TL is proposed. The language is based on the "document transformation language" DTL of Maneth and Neven which incorporates full MSO pattern matching, arbitrary navigation in the input tree using also MSO patterns, and named procedures. The new language generalizes DTL by additionally allowing procedures to accumulate intermediate results in parameters. It is proved that TL – and thus in particular DTL – despite their expressiveness still allow for effective inverse type inference. This result is obtained by means of a translation of TL programs into compositions of top-down finite state tree transductions with parameters, also called (stay) macro tree transducers.

1. INTRODUCTION

The extensible markup language (XML) has developed into the de facto standard for data exchange on the Internet [1]. Intuitively, data in an XML document is stored together with its structure, represented as a tree, i.e., an XML document is a textual representation of a finite labeled unranked tree. Storing data together with its structure has great benefits: more properties can be checked on the structure than on its plain content. This statement can be exemplified in the context of formal language theory: for a given context-free string language it is *not* decidable whether or not it is regular (= MSO definable). In contrast to that, for a macro tree transducer (a powerful model that translates derivation trees of context-free grammars) it *is* decidable whether or not it is MSO definable [12]. In the setting of XML this means, even a lightweight formal method like type checking becomes a very powerful property. In other words, if an XML transformation F *type checks* against input and output XML types T_{in}, T_{out}, i.e., if

$$\forall\, d \in T_{\text{in}} : \; F(d) \in T_{\text{out}},$$

then, besides the fact that such a transformation will never raise run-time type errors, it computes "quite surely" the transformation that was actually intended by the programmer. It is therefore highly desirable to have algorithms for *static type checking*, i.e., which check the above displayed property statically before the transformation is ever executed. The price for this accuracy in XML types is that type checking becomes more expensive (than the approximative techniques typically used in ordinary programming languages).

Static type checking of XML transformations can be considered in many different settings [28]: first, there is a choice in the XML type formalism (e.g., DTD, XML Schema [16], or RELAX NG [8]), second, there is a choice in the transformation formalism (e.g., XQuery [6], XSLT [21], or *fxt* [4]). Clearly, the decidability of the type checking problem heavily depends on the expressive power of the type and transformation formalisms used. If, for example, a general purpose programming language is used to define the transformation d then type checking is (obviously) undecidable and can only be approximated. Here we are interested in exact algorithms, i.e., we only consider transformation formalisms for which type checking is decidable.

A convenient abstraction of the existing XML type formalisms are regular tree languages [29, 31]. A large class of transformations which was specifically designed in order to model existing XML transformation languages, and for which type checking (w.r.t. regular tree languages) is decidable, is the k-pebble tree transducer (k-ptt) [27]. It is a finite state transducer which can mark nodes of the input tree using up to k different pebbles. When modeling an XML transformation (e.g., XSLT or XQuery), then for each variable that appears in a pattern of the transformation, one extra pebble is used (the pebbles determine variable bindings for patterns). This means that the number k of pebbles depends on the complexity of the patterns that appear in the XML transformation at hand. This number k is important, because the best known time complexity of type checking a k-ptt is $O(h_{k+2}(n))$ with $h_0(n) = n$ and $h_{m+1}(n) = 2^{h_m(n)}$, i.e., a tower of exponents of height $k+2$. This result stems from the fact that any k-ptt can be simulated by the composition of $k+1$ macro tree transducers (mtts) [11], and an m-fold composition of mtts can be type checked in time $O(h_{m+1}(n))$.

How can we obtain more efficient type checking algorithms than the one for k-ptts? One possibility is to restrict the expressive power of the type and transducer formalism. Restrictions under which type checking can be done in polynomial time were investigated by Martens and Neven [25]: the transducers have no pebbles, can only move downwards in the input tree, and may not delete any nodes of the input. Clearly, not many real-world XML transformations fall into this small class of transformations. A milder restriction has recently been considered by Engelfriet and Samwel [13]: k-ptts with "ℓ-bounded visibility": out of the k-pebbles only the ℓ last recently dropped ones can be seen; in this scenario only ℓ (and not k) mtts are needed to realize the transformation; this lowers k

to ℓ in the height of the exponents of the time complexity.

Our approach here is different; instead of restricting the k-ptt, we introduce a new transformation formalism. The new "transformation language" (TL) is powerful enough to (succinctly) describe many real-world XML transformations; on the other hand, all its transformations can be simulated by only a fixed number of macro tree transducers (namely, 3). Our aim was to choose our model as expressive as possible ('high-level'), while still allowing for effective type inference. In previous work [24], the transformation language DTL was presented. Based on the recursion mechanism of XSLT and using MSO (Monadic Second-Order logic) as pattern language, DTL subsumes the essential operations of existing domain specific languages for XML processing. All results concerning DTL in [24], however, are restricted to the fragment of deterministic and top-down transformations only. Here, we lift this restriction. We consider unrestricted DTL programs and also enhance the model by allowing accumulating parameters. Accumulating parameters allow for long-distance transportation of document parts and are therefore a convenient technique to provide access to context information [19]. The resulting transformation language is called TL. Since TL provides full MSO patterns for matching of rules and selection of nodes, it generalizes the pattern language XPath (as used in XSLT [9, 21] and XQuery [6]) without arithmetics and data value comparisons as well as *fxgrep* patterns (for *fxt* [4, 30]). Note that TL is meant as a mathematically convenient model; for implementation purposes MSO is not the preferred formalism because there can be a non-elementary blow up when going from MSO to a finite automaton (and most of our constructions here are based on finite automata). Currently we are implementing the algorithm for type checking which is presented here.

By defining a type checking algorithm for the transformation language TL we obtain a general type checking mechanism for all languages that can be compiled into TL. The key idea of our type checking algorithm for TL is to simulate TL transformations by compositions of top-down finite state tree transducers with parameters, which also have been called (stay) *macro tree transducers*. Macro tree transducers have been extensively studied as a model of syntax directed semantics [14, 19]. In particular, it is well-known that inverses of macro tree transductions (and compositions thereof) preserve recognizability [14]. Therefore, (compositions of) macro tree transductions can be type checked by means of *inverse type inference*: inverse type inference computes the pre-image of all ill-formed outputs. In particular if the set of well-formed outputs is recognizable, its complement is recognizable as well. Given that recognizability is effectively closed under taking the pre-image, type checking then reduces to check whether a given input type has an empty intersection with the pre-image. As recognizable tree languages are effectively closed under intersection, and non-emptiness is decidable, we obtain a type checking algorithm for TL programs.

In general, the macro tree transducer hierarchy (generated by sequentially composing mtts) seems to be a good yardstick for measuring the complexity of an XML transformation formalism: How many mtts are needed to simulate the transformers at hand? The mtt hierarchy has many desirable properties. As an example, for any transformation in the mtt hierarchy (hence, also for any k-ptt) it can be decided if it is of linear size increase (quite common for XML transformations) and if it is then *one* equivalent mtt can be constructed [23].

This paper is organized as follows: After introducing our transformation language in Sections 2 and 3, we show in Section 4 that type checking for a TL program P can be based on an effective decomposition of P into (stay) macro tree transducers. As an intermediate step for proving this decomposition result, we first present a decomposition of TL programs using *macro tree-walking transducers* in Section 5. In Sections 6 and 7, we finally show how deterministic and nondeterministic macro tree-walking transducers can be further decomposed.

2. TL BY EXAMPLE

Consider an XML representation of a mail file containing an `mbox` for incoming mails and a `trash` "folder" for deleted mails. Besides normal `mail` elements, the `mbox` also contains mails inside a `spam` element indicating that these mails have been identified as spam by some automated filter:

```
<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 order to clean up the mail folder, all spam mails should be moved into the trash folder while all other mails should be left untouched in `mbox`. Thus, a corresponding transformation should return:

```
<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>
```

The transformation explained above can be defined in our new transformation language TL. Here is the corresponding TL program.

$$
\begin{array}{lcl}
q(\mathsf{lab}_{\mathtt{doc}}(x_1)) & \Rightarrow & \langle \mathtt{doc} \rangle \\
 & & \quad q(\mathsf{lab}_{\mathtt{mbox}}(x_2)) \\
 & & \quad q(\mathsf{lab}_{\mathtt{trash}}(x_2)) \\
 & & \langle / \mathtt{doc} \rangle \\
q(\mathsf{lab}_{\mathtt{mbox}}(x_1)) & \Rightarrow & \langle \mathtt{mbox} \rangle \\
 & & \quad \mathit{copy}(x_1/x_2 \wedge \mathsf{lab}_{\mathtt{mail}}(x_2)) \\
 & & \langle / \mathtt{mbox} \rangle \\
q(\mathsf{lab}_{\mathtt{trash}}(x_1)) & \Rightarrow & \langle \mathtt{trash} \rangle \\
 & & \quad \mathit{copy}(\exists\, z.\, z; x_1 \wedge z/x_2 \wedge \mathsf{lab}_{\mathtt{spam}}(x_2)) \\
 & & \quad \mathit{copy}(x_1/x_2) \\
 & & \langle / \mathtt{trash} \rangle \\
\mathit{copy}(x_1) & \Rightarrow & x_1
\end{array}
$$

A TL program consists of a collection of function definitions; each definition consists of *transformation rules*. A rule for a function f

is of the form $f(m) \Rightarrow A$ where m is a *match pattern* and A is an *action*. The match pattern determines the nodes of the input to which the rule is applicable. The action determines how output is constructed; it consists of output nodes (in usual XML syntax) and function calls. A function call consists of a function name g and a *select pattern*. The pattern determines to which nodes of the input g is applied.

Match and select patterns in TL are MSO formulas over the free variables x_1 and x_1, x_2, respectively. A rule is applicable to a particular node v of the input if its match pattern is satisfied when binding x_1 to v. Applying the rule means to construct the output and function calls of the action. The select patters determine, under the binding of x_1 to v, to which nodes x_2 the function should be applied. In our example, the formula $\mathsf{lab}_{\mathtt{doc}}(x_1)$ in the first rule denotes all nodes labeled by `doc`. Thus, the first rule for q is applicable to the `doc` element of the input. Its action specifies to produce a `doc` element the content of which is obtained by applying q to the nodes determined by the select patterns $\mathsf{lab}_{\mathtt{mbox}}(x_2)$ and $\mathsf{lab}_{\mathtt{trash}}(x_2)$, respectively. Since these select patterns do not contain x_1, the nodes to which they refer are determined *absolutely* in the input tree. The second rule of q is applicable if the current node is an `mbox` element. Its action specifies to generate an `mbox` element the content of which is obtained by applying *copy* to the nodes selected by the pattern $x_1/x_2 \wedge \mathsf{lab}_{\mathtt{mail}}(x_2)$, i.e., the children x_2 of the current node x_1 which are labeled `mail`. The auxiliary function *copy* realizes the identity by copying the subtree to which it is applied to the output. In summary, since the current node is the `mbox` element of the input, the second q-rule copies all `mail` elements from `mbox` into the `mbox` element of the output.

The third q-rule transforms the `trash` element. The first call to the function *copy* is meant to copy all `spam` elements from `mbox` into `trash`. The necessary select pattern is given by the MSO formula $\exists\, z.\, z; x_1 \wedge z/x_2 \wedge \mathsf{lab}_{\mathtt{spam}}(x_2)$. This formula selects all nodes x_2 labeled by `spam` whose ancestor z is a left sibling of the current node x_1. The other call of function *copy* is meant to copy all elements x_2 of the trash folder x_1 into the `trash` element of the output.

3. TL IN DETAIL

An XML document is a linear representation of a sequence of *unranked trees*. An unranked tree consists of a root node (labeled by some label a) and a *forest* f, written $\langle a\rangle f\langle /a\rangle$. A *forest* consists of a sequence of unranked trees, written $t_1 t_2 \cdots t_m$ or ϵ if $m = 0$. The set of all trees (forests) with labels from a set Σ is denoted by $\mathcal{T}_\Sigma$ ($\mathcal{T}_\Sigma^*$, respectively).

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

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

Here, x and X are individual and set variables, respectively. Individual variables x range over nodes of the input forest and set variables X over node sets. The binary relation symbols ";" and "/" denote the *next sibling* and *child* relation between nodes, respectively. The set lab_a denotes all nodes labeled by $a \in \Sigma$. Note that the MSO formulas in our example do not use set variables. As usual, we also feel free to use the equality predicate $=$, conjunctions $\wedge$, as well as universal quantification $\forall$ with their usual meanings. In particular if the MSO formula contains only free variables $x_1, \ldots, x_k$, we write

$$(v_1, \ldots, v_k) \models_f \phi$$

to denote that ϕ holds (matches) in the forest f if we bind x_i to the node v_i, $i = 1, \ldots, k$. Note that in our application, formulas ϕ always serve either as a match or select pattern. Hence, the parameter k only takes the values 1 and 2, respectively.

Definition 1 A TL program P is a pair (R, A_0) where A_0 is an initial action and R is a finite set of rules of the form

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

where q is a function name, ϕ is an MSO match pattern with one free variable x_1, the $y_1, \ldots, y_k$, $k \geq 0$, are accumulating parameters, and A is the action. Possible actions (including the initial action A_0) are described by the grammar:

$$A ::= \epsilon \mid A_1 A_2 \mid \langle a\rangle A\langle /a\rangle \mid x_1 \mid y_j \mid q'(\psi, A_1, \ldots, A_m)$$

where a is the label of a node in the output, y_j is one of the accumulating parameters from the left-hand side (i.e., $1 \leq j \leq k$), q' is a function name, and ψ is a select pattern, i.e., an MSO formula with free variables x_1, x_2.

Intuitively, the meaning of the actions listed in our definition is as follows: The output for a node x_1 in the input forest either can be the empty forest, the concatenation $A_1 A_2$ of the outputs produced by the actions A_1 and A_2, an individual element a whose content is recursively determined by some action, a copy x_1 of the current input (sub) tree, the content of one of the accumulating parameters y_j, or a recursive call to some function q' where the values of the actual parameters are again determined by actions A_i and furthermore, the nodes to be processed next are determined by some binary MSO select pattern ψ.

The evaluation of a TL program begins at the root node of the input document (seen as an unranked tree t). Given an input document (seen as an unranked tree t), a TL program P starts processing by evaluating its initial action A_0 for the root node of t. A function q with parameters $f_1, \ldots, f_k$ is applied to a node v by carrying out the following steps. First, we (nondeterministically) choose one of the rules $q(\phi, y_1, \ldots, y_k) \rightarrow A$ for q where ϕ matches v. Then we bind the formal parameters y_j to the actual parameters f_j and execute the action part A of the chosen rule. In presence of *nested* function calls, we adhere to an *outside-in* or *call-by-name* evaluation strategy, i.e, the outermost call is performed first.[1] Depending on the current node v the next nodes for the application of a successive function call $q'(\psi, A_1, \ldots, A_m)$ are selected in accordance to the binary pattern ψ, i.e., we choose all nodes u for which the pair (v, u) matches ψ.

In general, more than one pattern may match at a point of the computation. In different practical transformation languages, different resolution strategies have been proposed. *fxt*, for example simply chooses the first applicable rule. XSLT, on the other hand, insists on choosing the "most specific" rule [9] or the *last* applicable rule [21]. From an expressiveness point of view, MSO logic as a pattern language is strong enough to make priorities explicit, e.g., by adding the negation of a conjunction of all patterns of higher priorities. Thus, we can w.l.o.g. always assume that match patterns of the same function are all mutually disjoint. In this way we call a TL program *deterministic* if for each function q at every node v of every input document at most one match pattern of q's rules matches. Note that this property is decidable for a given TL program P. Non-determinism in TLprograms can be used if the transformation to be type checked uses predicates which cannot be

[1]In the outside-in strategy the parameters of a function call may themselves contain function calls which are thus transferred to the body in an unevaluated form [14].

expressed in MSO. Then, nondeterministic choice between different potentially chosen actions gives a *type safe* over-approximation of the real transformation.

Let us now formally define the semantics of a TL program P. Instead of reasoning about the *operational* behavior of transformations, we here prefer to take a *denotational* point of view. Let Δ denote the set of symbols possibly occurring in input or output forests and $\mathcal{N}(f)$ denote the set of all nodes of a forest f. For a given input forest $f \in \mathcal{T}_\Delta^*$, each function name q of P can be considered as denoting the function

$$[\![q]\!]_f : \mathcal{N}(f) \times (2^{\mathcal{T}_\Delta^*})^k \to 2^{\mathcal{T}_\Delta^*}$$

which takes a node $v \in \mathcal{N}(f)$ of the input forest f and a list $F_1, \ldots, F_k$ of sets $F_i \subseteq 2^{\mathcal{T}_\Delta^*}$ of actual parameters and returns a set of output forests. The reason for having *sets* of output forests in the parameters of the denotation of q is that in the *outside-in* evaluation mode, further function calls may appear inside of parameter trees (and, due to nondeterminism, they may generate several different output trees); see [14] for more details. The denotation functions are defined as the least fixpoint of:

$$[\![q]\!]_f(v, F_1, \ldots, F_k) \supseteq [\![A]\!]_f\ v\ \eta$$

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

$$\begin{array}{rcl}
[\![\epsilon]\!]_f\ v\ \eta & \supseteq & \{\epsilon\} \\
[\![x_1]\!]_f\ v\ \eta & \supseteq & f_v \\
[\![y_j]\!]_f\ v\ \eta & \supseteq & \eta(y_j) \\
[\![A_1 A_2]\!]_f\ v\ \eta & \supseteq & ([\![A_1]\!]_f\ v\ \eta)\ ([\![A_2]\!]_f\ v\ \eta) \\
[\![\langle a\rangle A_1 \langle /a\rangle]\!]_f\ v\ \eta & \supseteq & \langle a\rangle [\![A_1]\!]_f\ v\ \eta \langle /a\rangle \\
[\![q'(\psi, A_1, \ldots, A_m)]\!]_f\ v\ \eta & \supseteq & [\![q']\!]_f(v_1, \underline{F}) \ldots [\![q']\!]_f(v_l, \underline{F}),
\end{array}$$

where f_v denotes the subtree of f that is rooted at v and where $\underline{F}$ abbreviates the recursive evaluation of all parameters A_i, $i = 1, \ldots, m$ for the current node v and the actual parameter environment η, thus $\underline{F} = [\![A_1]\!]_f\ v\ \eta, \ldots, [\![A_m]\!]_f\ v\ \eta$, and $\{v_1 < \ldots < v_l\} = \{v' \mid v, v' \in \mathcal{N}(f), (v, v') \models_f \psi\}$. The relation "$<$" is induced by the lexicographical order on $\mathcal{N}(t)$ and is also called *document order*. Note that we have denoted the concatenation of sets $F_1, \ldots, F_m$ by juxtaposition, i.e., $F_1 \ldots F_m = \{s_1 \ldots s_m \mid s_i \in F_i\}$.

The *transformation* realized by P on non-empty forests, denoted by τ_P, is defined by

$$\tau_P = \{(f, s) \in \mathcal{T}_\Delta^* \times \mathcal{T}_\Delta^* \mid s \in [\![A_0]\!]_f\, 1\, \emptyset\}.$$

where "1" is the root node of the left-most tree of the input forest f and $\emptyset$ is the empty variable assignment. It can be shown that this denotational semantics by means of mutually recursive functions indeed coincides with (a suitable formalization of) the informally presented operational semantics based on call-by-name rewriting of function calls.

4. TYPE CHECKING

The main result of this paper is that recognizability of sets of forests is (effectively) preserved by taking pre-images of TL transformations.

Theorem 2 *For every* TL *program P and recognizable set $R \subseteq \mathcal{T}_\Delta^*$ of output forests, the pre-image*

$$\tau_P^{-1}(R) = \{f \in \mathcal{T}_\Delta^* \mid \exists\, r \in R:\ (f, r) \in \tau_P\}$$

is again recognizable. Moreover, a finite automaton recognizing $\tau_P^{-1}(R)$ can be effectively constructed from the program P and a finite automaton for R.

As a corollary, this result gives a solution to the type checking problem for TL programs. Assume we are given recognizable sets $T_{\text{in}}, T_{\text{out}} \subseteq \mathcal{T}_\Delta^*$ of the admissible input and output types, respectively. In order to check that the set of images produced by a TL program P for every $f \in T_{\text{in}}$ always conforms to T_{out}, we perform *inverse type inference*. Inverse type inference was used by Milo et al. in [27] to prove that k-pebble tree transducers can be type checked effectively. Note that, in the context of tree transformers, forward type inference does not work well: in general transformations do *not* preserve regularity. Intuitively, whenever a transformation copies, i.e., generates multiple independent outputs which depend on the same input, then it no longer preserves recognizability. In that case forward type inference can only be done approximative. Inverse type inference, on the other hand, is possible whenever the *pre-images* of recognizable sets are effectively recognizable. Interestingly, this is the case for a larger class of transformations and in particular, by Theorem 2, for all TL programs.

Using the fact inverses of TL transformations effectively preserve recognizability (Theorem 2), type checking of TL programs can be done as follows. First, determine a finite automaton for the set

$$U = \tau_P^{-1}(\mathcal{T}_\Delta^* \backslash T_{\text{out}})$$

This is possible by Theorem 2 and because recognizable sets are effectively closed under complement. The set U represents the set of all inputs for which the transformation τ_P can return results not conforming to T_{out}. In the second step, it therefore remains to verify whether or not $T_{\text{in}} \cap U = \emptyset$. In particular, no wrong results are produced iff the intersection is empty. Since intersection emptiness is decidable for finite automata, we obtain at the following corollary.

Corollary 3 *Type checking for* TL *programs is decidable.* $\square$

In order to prove Theorem 2, we show that every TL transformation can be effectively decomposed into a composition of three *(stay) macro tree transducers* (mtts and s-mtts, respectively). Macro tree transducers have been introduced as a model for syntax-directed semantics by Engelfriet and Vogler [14]. Essentially, macro tree transducers are first-order functional programs which recursively descend over their first argument by means of pattern-matching while all other parameters are only used for accumulating intermediate results . A *stay* macro tree transducer is a minor generalization of an mtt which is not obliged to proceed to subterms of their argument parameter in every step, but may process the same argument node arbitrarily often [11]. Since their introduction it has been known that taking pre-images of mtt transductions (and compositions thereof) effectively preserves recognizability (Theorem 7.4 of [14]). The same also holds for s-mtts (Corollary 44 of [11]).

The remainder of this paper deals with the decomposition of a TL program P into a composition of (stay) macro tree transducers. In the next section we state the decomposition theorem and then start proving it by showing how to relabel the input of P, so that a "macro tree-walking transducer" (an intermediate model which generalizes the mtt by up and stay moves on the input tree) can simulate P on the relabeling. Note that all simulations (including the relabeling) work on encodings of forests as binary trees. The final simulation of the walking transducer by macro tree transducers follows in two section, first for the deterministic case and then for the nondeterministic one.

5. DECOMPOSING TL PROGRAMS

Our main decomposition result for TL programs is:

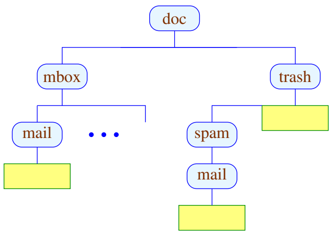
Theorem 4 1. *Every deterministic* TL *program can be compiled into a composition of three deterministic macro tree transducers.*

2. *Every nondeterministic* TL *program can be compiled into the composition of two macro tree transducers and one stay macro tree transducer.*

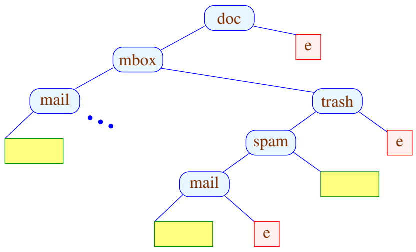
Before we decompose a TL program into a relabeling followed by a macro tree-walking transducer we discuss the coding of forests as binary trees.

5.1 Encoding Forests as Trees

TL programs transform forests, i.e., sequences of unranked trees. Macro tree transducers (as well as macro tree-walking transducers) on the other hand transform ranked trees. One possibility of dealing with this difference is to generalize the mtt model to forests [32]. Here we choose another common approach, namely to code forests as binary trees. The standard way of doing this is to represent the empty forests by a leaf e and to code the content of an element node as the left child while the right child represents the forest of right siblings of the element. Consider, e.g., the (unranked) tree representation of our mailbox document:



The corresponding representation as a binary tree looks as follows:



For convenience, ranked trees are denoted by terms t constructed according to the grammar

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

where b is a nullary symbol and a is a constructor of rank $k > 0$. As usual and in order to avoid case distinctions, we view nullary symbols b as constructors of rank 0.

5.2 Compiling Match and Select Patterns

In the next step we want to make available at every node v of (the binary tree representation of) the input forest enough information to decide locally which patterns occurring in the program P are matches or selections of v. In order to do so, we here first compile all MSO formulas occurring in the TL program P into one *finite automaton*. As usual, a (bottom-up) *finite* (state tree) *automaton* is a tuple $B = (S, \Sigma, \delta, F)$ where S is a set of *states*, $F \subseteq S$ is a set of *accepting states*, and δ is a finite set of transitions of the form $(s, a, s_1 \ldots s_k)$ where a is a symbol of rank $k \geq 0$ from the *ranked alphabet* Σ and $s, s_1, \ldots, s_k$ are states in S. Our finite automata will operate on binary representations of document forests, i.e., $\Sigma = \Delta \cup \{e\}$ where all elements in Δ have rank 2 and e (representing the empty forest) has rank 0.

A transition $(s, a, s_1 \ldots s_k)$ denotes that if B reaches states s_i for trees t_i, then it can reach state s for the tree $a(t_1, \ldots, t_k)$. A *run* of A on a tree $t \in \mathcal{T}_\Sigma$ is a mapping $r : \mathcal{N}(t) \to S$ which assigns to each node v a state $r(v) \in S$ such that locally the transition relation is respected. The tree language accepted by B consists of the trees $t \in \mathcal{T}_\Sigma$ by which B can reach an accepting state, or, equivalently, all trees having runs r which map their roots to accepting states.

Let Φ and Ψ be the finite sets of MSO match and select patterns of P, respectively. By using suitable encodings of the basic predicates "/" and ";", the formulas in Φ and Ψ can be equivalently expressed by MSO formulas over binary trees. Then using the well-known translation of MSO formulas over finite ranked trees by Thatcher and Wright [33], we can constructing a nondeterministic tree automaton $\mathcal{P} = (S, \Delta, \delta, F)$ together with sets $U_\phi \subseteq S, \phi \in \Phi$, and relations $B_\psi \subseteq S^2, \psi \in \Psi$, such that for every document forest f the following holds:

1. For every $\phi \in \Phi$, $(v) \models_f \phi$ iff there exists an accepting run r of $\mathcal{P}$ on (the binary encoding of) f such that $r(v) \in U_\phi$;

2. For every $\psi \in \Psi$, $(v_1, v_2) \models_f \psi$ iff there exists an accepting run r of $\mathcal{P}$ on (the binary encoding of) f such that $(r(v_1), r(v_2)) \in B_\psi$.

5.3 Annotating the Input

For the binary tree representation t of a forest, let $S(t) \subseteq S$ denote the set of states s in which the match automaton may possibly reach the root of t. The set $S(t)$ is recursively defined by:

$$\begin{array}{rcl} S(e) & = & \{s \in S \mid (s, e) \in \delta\} \\ S(a(t_1, t_2)) & = & \{s \in S \mid \exists s_i \in S(t_i) : (s, a, s_1 s_2) \in \delta\} \end{array}$$

For our implementation of P through mtts we relabel each node of the input as follows:

1. First, each node v of the input tree t labeled by a symbol $a \in \Delta$ is relabeled by $\langle a, S_1, S_2 \rangle$ where S_i is the set of states reachable at the subtree rooted at the i-th child of v. According to the recursive definition of the sets $S(t')$, this can be done by means of a single deterministic post-order traversal over t or, technically speaking, by a total deterministic *bottom-up relabeling* [10].

2. Then, each node v additionally receives the value 0 if v is the root of t or its child number $i \in \{1, 2\}$ if v is the i-th child of its father node. Moreover, we place at v the set $T(v) = \{r(v) \mid r \text{ accepting run for } f\}$, i.e., the set of all states to which v is mapped by accepting runs of the match automaton. Thus, now internal nodes v with original label a have labels $\langle a, j, T(v), S(v_1), S(v_2) \rangle$ (v_1, v_2 the first and second child of v, respectively) whereas leaf nodes v have the labels $\langle e, j, T(v) \rangle$. This second relabeling can be implemented by means of a single deterministic pre-order traversal over the annotated t, or, technically speaking, by a total deterministic *top-down relabeling* [10].

The composition of a bottom-up relabeling followed by a top-down relabeling is equivalent to a top-down relabeling with regular look-ahead, or to an MSO definable relabeling [10, 5]. In what follows, denote the class of all such relabelings by REL.

5.4 Removing Global Selection

In this section we consider how to determine the nodes that are selected by a select pattern during the execution of a TL program. The intermediate model (on our way to prove Theorem 4) is a generalization of mtts called *macro tree-walking transducer* (for short, 2-mtt) (equal to RT(Tree-walk) in [15], the 0-pebble macro tree transducer of [11], and to the macro attributed tree transducer of [22]). The idea is to compile the global selection steps of the TL program P into *local movements*. Intuitively, a 2-mtt is similar to a TL program but operates on ranked trees instead of forests. Moreover, match patterns of rules are restricted to the form $\mathsf{lab}_a(x_1)$ only, i.e., are only allowed to check the label of the current input node x_1. Also, select patterns may only be of one of the following forms:

- $x_1 = x_2$, i.e., the current node x_1 itself is selected;
- $x_2 = \mathsf{father}(x_1)$, i.e., the father of x_1 is selected;
- $x_2 = \mathsf{child}_i(x_1)$, i.e., the i-th child of x_1 is selected; on ranked trees $\mathsf{child}_i(x_1)$ replaces the ";" and "/" predicates.

Note that each of these select patterns selects at most one next node v', given any binding of x_1 to a node v of the input tree. In our construction we insist that always *exactly one next node* is selected. This is possible due to our relabeling from Section 5.3 which allows to determine whether the current node is the root (excluding the selection of the father) or a leaf (excluding the selection of children).

For the following, we w.l.o.g. assume that the TL program P does not contain the copy action x_1 in right-hand sides. The key idea of the simulation of the TL program P by a 2-mtt is as follows: Assume that a call $q(\psi, A_1, \ldots, A_k)$ of the TL function q is to be evaluated at a node v of the input forest f where ψ is a binary select pattern. This means that the function q must be called for the nodes $v_1 < \ldots < v_m$ where the sequence of pairs $(v, v_1), \ldots, (v, v_m)$ represents all matches of the pattern ψ in the input forest. In order to collect the forests produced by $[\![q]\!]_f(v_i, [\![A_1]\!]_f, \ldots, [\![A_k]\!]_f)$, the 2-mtt successively proceeds from v to the root node of the (annotated) input where it returns a representation of the list

$$[\![q]\!]_f(v_1, [\![A_1]\!]_f, \ldots, [\![A_k]\!]_f) \ldots [\![q]\!]_f(v_m, [\![A_1]\!]_f, \ldots, [\![A_k]\!]_f)$$

The applications to the nodes v_i are collected on the path to the root by traversing the prevailing left or right siblings, depending on the direction from where the simulation has moved up. For the up movements, we therefore introduce the function up (with sub- and superscripts for storing extra auxiliary information), whereas for the collections of matches to the left and right of the current path, we introduce the function down (again with sub- and superscripts).

Technically, the new initial action A_0' equals $[A_0]_0$ and each TL rule

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

is simulated by the rules

$$\begin{array}{lcl} q(\mathsf{lab}_{\langle a,j,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k) & \to & [A]_j \\ q(\mathsf{lab}_{\langle e,j,T\rangle}(x_1), y_1, \ldots, y_k) & \to & [A]_j \end{array}$$

whenever the TL rule is indeed applicable to the current node, i.e., whenever $U_\phi \cap T \neq \emptyset$. Depending on the child number $j \in \{0, 1, 2\}$ of the current node, the new right-hand sides are determined by means of the transformation $[.]_j$:

$$\begin{array}{rcl} [\epsilon]_j & = & e \\ [y_i]_j & = & y_i \\ [x_1]_j & = & x_1 \\ [\langle b\rangle A\langle /b\rangle]_j & = & b([A]_j, e) \\ [A_1A_2]_j & = & [A_1]_j @ [A_2]_j \end{array}$$

Since 2-mtts operate on ranked trees, they do not support concatenation as a base operation. Therefore, we express concatenation symbolically by means of the new binary constructor "@" (written infix for better readability). The occurrences of this symbolic operator will be evaluated in a subsequent transformation. In order to simulate function calls $q'(\psi, \ldots)$ of the action part, we make use of the functions up and down. If $q'(\psi, A_1, \ldots, A_k)$ occurs at the root node r and (r, r) is a match for ψ, i.e., r is also selected by ψ, then the 2-mtt calls q' at r and concatenates the results of calling down for the child nodes:

$$\begin{array}{l} [q'(\psi, A_1, \ldots, A_k)]_0 \quad = \\ \qquad q'(x_1 = x_2, [A_1]_0, \ldots, [A_k]_0) @ \\ \qquad \mathsf{down}^{q'}_{\psi,g_1}(x_2 = \mathsf{child}_1(x_1), [A_1]_0, \ldots, [A_k]_0) @ \\ \qquad \mathsf{down}^{q'}_{\psi,g_2}(x_2 = \mathsf{child}_2(x_1), [A_1]_0, \ldots, [A_k]_0) \end{array}$$

Note that the function down records the name q' of the called function together with the select pattern ψ and some extra g_i which is explained below. Note also that the current node is selected by ψ iff T contains a state s such that $(s, s) \in B_\psi$. If the current node is *not* selected by ψ, then we omit the call $q'(x_1 = x_2, [A_1]_0, \ldots, [A_k]_0)$. If the function call $q'(\psi, A_1, \ldots, A_n)$ occurs at a node v different from the root node of the input, i.e., $j \neq 0$, and v is also selected by pattern ψ, then the subsequent call for q' is simulated by:

$$\begin{array}{l} [q'(\psi, A_1, \ldots, A_k)]_j \quad = \\ \qquad \mathsf{up}^{j,q'}_{\psi,h}(x_2 = \mathsf{father}(x_1), [A_1]_j, \ldots, [A_k]_j, \\ \qquad\qquad q'(x_1 = x_2, [A_1]_j, \ldots, [A_k]_j) @ \\ \qquad\qquad \mathsf{down}^{q'}_{\psi,g_1}(x_2 = \mathsf{child}_1(x_1), [A_1]_j, \ldots, [A_k]_j) @ \\ \qquad\qquad \mathsf{down}^{q'}_{\psi,g_2}(x_2 = \mathsf{child}_2(x_1), [A_1]_j, \ldots, [A_k]_j)) \end{array}$$

As before, we compute the list of outputs for matches inside the subtree rooted at the current node. Now, however, we pass (a tree representation of) this forest in an accumulating parameter to the function up which is meant to proceed upward to the root to collect also the calls for the remaining nodes selected by ψ. Similar to the function down, up receives as extra information the name q' of the function to be called, the select pattern ψ, but furthermore also the current child number j and the mapping h which is explained below. Again, if the current node v is not selected by ψ, then we omit the call $q'(x_1 = x_2, [A_1]_j, \ldots, [A_k]_j)$ in the accumulating parameter of up. The extra information h and g_i for the functions up and down, respectively, are relations on states which are meant to relate states at the node where the current function q has been applied to those states at present nodes where up and down are to be evaluated which are possibly connected through a common run of the match automaton $\mathcal{P}$:

$$\begin{array}{lcl} h & = & \{(s, s) \mid s \in T\} \\ g_i & = & \{(s, s_i) \in T \times S_i \mid \exists\, s_{3-i} \in S_{3-i} \,:\, (s, a, s_1 s_2) \in \delta\} \end{array}$$

The relation g_i stores information about the node for which down is called. On the other hand, the relation h used by up stores information about the node at which the particular call was issued.

It remains to provide definitions for the functions up and down. The call $\mathsf{down}^{q'}_{\psi,h}$ is meant to traverse the subtree at the current node v and to return (a representation of) the list of results obtained by applying the function q' to all nodes $v_1, \ldots, v_k$ in document order which are selected by ψ. A call $\mathsf{up}^{j,q'}_{\psi,h}$ means that the current node v was reached coming from its j-th child. The function up moves the current node to the root node, and then, depending whether it came from the first or second subtree (recorded in the superscript j) it calls down on the other subtree. For the case that the current node is selected by the pattern ψ, i.e., $h_i \cap B_\psi \neq \emptyset$, we define:

$$
\begin{array}{l}
\mathsf{up}^{1,q'}_{\psi,h}(\mathsf{lab}_{\langle a,0,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k, y_{k+1}) \quad \to \\
\quad q'(x_1 = x_2, y_1, \ldots, y_k) \mathbin{@} y_{k+1} \mathbin{@} \\
\quad \mathsf{down}^{q'}_{\psi,g_1}(x_2 = \mathsf{child}_2(x_1), y_1, \ldots, y_k) \\
\mathsf{up}^{1,q'}_{\psi,h}(\mathsf{lab}_{\langle a,j,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k, y_{k+1}) \quad \to \\
\qquad \mathsf{up}^{j,q'}_{\psi,h_1}(x_2 = \mathsf{father}(x_1), y_1, \ldots, y_k, \\
\qquad\qquad q'(x_1 = x_2, y_1, \ldots, y_k) \mathbin{@} y_{k+1} \mathbin{@} \\
\qquad\qquad \mathsf{down}^{q'}_{\psi,g_1}(x_2 = \mathsf{child}_2(x_1), y_1, \ldots, y_k)) \\
\mathsf{up}^{2,q'}_{\psi,h}(\mathsf{lab}_{\langle a,0,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k, y_{k+1}) \quad \to \\
\quad q'(x_1 = x_2, y_1, \ldots, y_k) \mathbin{@} \\
\quad \mathsf{down}^{q'}_{\psi,g_2}(x_2 = \mathsf{child}_1(x_1), y_1, \ldots, y_k) \mathbin{@} y_{k+1} \\
\mathsf{up}^{2,q'}_{\psi,h}(\mathsf{lab}_{\langle a,j,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k, y_{k+1}) \quad \to \\
\qquad \mathsf{up}^{j,q'}_{\psi,h_2}(x_2 = \mathsf{father}(x_1), y_1, \ldots, y_k, \\
\qquad\qquad q'(x_1 = x_2, y_1, \ldots, y_k) \mathbin{@} \\
\qquad\qquad \mathsf{down}^{q'}_{\psi,g_2}(x_2 = \mathsf{child}_1(x_1), y_1, \ldots, y_k) \mathbin{@} y_{k+1})
\end{array}
$$

Observe that the recursive calls to up receive modified values h_i. Also, we have to generate the correct values g_i for the subsequent calls of down. Before we give their formal definitions, we just remark that by construction the current node will be selected by ψ iff $h_i \cap B_\psi \neq \emptyset$. In case the current node is not selected, we exclude the call $q'(x_1 = x_2, y_1, \ldots, y_k)$ from the right-hand side of the corresponding rule. The new values h_i and g_i are defined by:

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

A call to the function down at a node v is meant to concatenate the values of q' for all selected nodes in the subtree rooted at v. Thus, it can recursively be defined by the rules:

$$
\begin{array}{l}
\mathsf{down}^{q'}_{\psi,h}(\mathsf{lab}_{\langle \epsilon,j,T\rangle}(x_1), y_1, \ldots, y_k) \qquad \to \\
\quad q'(x_1 = x_2, y_1, \ldots, y_k) \\
\mathsf{down}^{q'}_{\psi,h}(\mathsf{lab}_{\langle a,j,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k) \quad \to \\
\quad q'(x_1 = x_2, y_1, \ldots, y_k) \mathbin{@} \\
\quad \mathsf{down}^{q'}_{\psi,g'_1}(x_2 = \mathsf{child}_1(x_1), y_1, \ldots, y_k) \mathbin{@} \\
\quad \mathsf{down}^{q'}_{\psi,g'_2}(x_2 = \mathsf{child}_2(x_1), y_1, \ldots, y_k)
\end{array}
$$

with

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

Again, these rules deal with the case where the current node is a match, i.e. $h \cap B_\psi \neq \emptyset$. Otherwise, the call $q'(x_1 = x_2, y_1, \ldots, y_k)$ in the right-hand sides must be omitted. Thus, assume that the node v is no proper descendant of v' where h describes a relation between the states at v and the states at v'. Then g'_i describes the corresponding relation between the states at v and the states at the i-th child of v'.

Consider the example from Section 2. Following the construction of this section, the rule

$$
\begin{array}{lll}
q(\mathsf{lab}_{\mathtt{mbox}}(x_1)) & \Rightarrow & \langle \mathtt{mbox} \rangle \\
 & & \quad copy(x_1/x_2 \wedge \mathsf{lab}_{\mathtt{mail}}(x_2)) \\
 & & \langle /\mathtt{mbox} \rangle
\end{array} \tag{1}
$$

is simulated by the following 2-mtt rule

$$
\begin{array}{l}
q(\mathsf{lab}_{\langle \mathtt{mbox},\ldots\rangle}(x_1)) \to \\
\qquad \mathtt{mbox}(\mathsf{up}^{1,copy}_{\psi,\ldots}(x_2 = \mathsf{father}(x_1), \\
\qquad\qquad \mathsf{down}^{copy}_{\psi,\ldots}(x_2 = \mathsf{child}_1(x_1)) \mathbin{@} \\
\qquad\qquad \mathsf{down}^{copy}_{\psi,\ldots}(x_2 = \mathsf{child}_2(x_1))), e)
\end{array} \tag{2}
$$

where ψ denotes the select pattern $x_1/x_2 \wedge \mathsf{lab}_{\mathtt{mail}}(x_2)$. (A sketch of the correctness proof can be found in the appendix.)

5.5 Wrapping it Up

In the previous section we compiled the TL program P into a 2-mtt operating on suitably annotated tree representations of forests. The 2-mtt, however, does not compute the (tree representations of the) output forests of P directly. Instead, it generates occurrences of the symbolic operator @ whenever needed. So far, we succeeded to prove the following decomposition result:

Theorem 5 *For every* TL *program P there is a composition*

$$\text{APP} \circ \tau_{P'} \circ \text{REL}$$

which computes the binary tree encoding of the transformation realized by P; APP *is the mapping which evaluates all occurrences of @ in binary trees, P' is a 2-mtt, and* REL *is a relabeling in* REL. *Moreover, if P is deterministic then the 2-mtt P' can be chosen to be deterministic too.* ◇

6. DETERMINISTIC 2-MTTS

In this section we want to change the macro tree-walking transducer constructed in the previous section in such a way that the new transducer evaluates the input *strictly top-down*. For simplicity we first consider the case where the original TL program and the 2-mtt P' constructed in the previous section are deterministic. Then, P' can be realized by the composition of two 2-mtt which both only use selections of children nodes. Such transducers are known as *macro tree transducers* (mtts) [14, 19]. Macro tree transducers combine top-down tree transducers and macro grammars [17]. Technically, they can be considered as first-order functional programs performing a top-down traversal over the first argument while possibly accumulating intermediate results in additional parameters. These accumulating parameters are also called *context* parameters because they implicitly give access to the context of the currently processed tree in the first argument. Formally, mtts are 2-mtts where the only allowed select patterns are $x_2 = \mathsf{child}_i(x_1)$. Instead of using the 2-mtt formalism, we write mtt rules in the style of functional programs using pattern matching on the first argument. As an example of a macro tree transducer consider the APP-function of Theorem 5 which evaluates the "@"-symbols in the tree representation of a forest. This function can be implemented by a deterministic mtt with initial action $q(z, e)$ and the rules:

$$
\begin{array}{rcl}
q(e, y_1) & \to & y_1 \\
q(a(z_1, z_2), y_1) & \to & a(q(z_1, e), q(z_2, y_1)) \\
q(z_1 \mathbin{@} z_2, y_1) & \to & q(z_1, q(z_2, y_1))
\end{array}
$$

In particular, the variable z in the initial action is meant to be bound to the input tree of the mtt, while the variables z_1, z_2 in the rules refer to the first and second child, respectively.

A *macro tree transducer* is a pair (R, s_0) where s_0 is an initial action and R is a finite set of rules. A rule is of the form

$$q(a(z_1, \ldots, z_n), y_1, \ldots, y_k) \to t,$$

where q is a function symbol, a is the label of a node in the input, z_i are (pattern) variables, and $y_1, \ldots, y_k$, $k \geq 0$, are the accumulating parameters. The right-hand sides (actions) t are defined by the grammar

$$t ::= q'(z_i, t_1, \ldots, t_m) \mid b(t_1, \ldots, t_k) \mid y_j$$

where q' again is a function symbol, z_i, $1 \leq i \leq n$, is a variable from the left-hand side, b is the label of a node in the output, and y_j is one of the accumulating parameters of the left-hand side (i.e., $1 \leq j \leq k$). In initial actions we do not allow variables y_j and use z instead of the z_i:

$$s ::= q'(z, s_1, \ldots, s_m) \mid b(s_1, \ldots, s_k)$$

Since mtts cannot move upwards in their input, we need an appropriate method to translate moves to ancestors of the 2-mtt. The key idea is, instead of moving up to a node u, we generate at u all possible function calls and pass them in the parameter positions to any further calls. In this way a call of q for the father node can be simulated by selecting the corresponding parameter y_q. This simulation, however, is not able to deal properly with accumulating parameters of the 2-mtt which are modified during up moves. Therefore, we make use of a decomposition idea already used, e.g., in [14, 11]: we decompose the deterministic 2-mtt P' into a deterministic 2-mtt P'' *without accumulating parameters* followed by some extra transformation YIELD. The 2-mtt P'' executes only the calling behavior of P', while performing parameter passing symbolically through formal substitution symbols α_i (representing the formal parameter y_i) and σ_m (representing the substitution of m actual parameters). The 2-mtt P'' is obtained from P' by rewriting the initial action A'_0 of P' into $\mathcal{S}[A'_0]$ as well as each rule

$$q(\mathsf{lab}_a(x_1), y_1, \ldots, y_n) \to t$$

of P' into

$$q(\mathsf{lab}_a(x_1)) \to \mathcal{S}[t]$$

where the new actions are obtained by:

$$\begin{array}{lcl}
\mathcal{S}[y_i] & = & \alpha_i \\
\mathcal{S}[b(t_1, \ldots, t_n)] & = & b(\mathcal{S}[t_1], \ldots, \mathcal{S}[t_n]) \\
\mathcal{S}[q'(\psi, t_1, \ldots, t_m)] & = & \sigma_m(q'(\psi), \mathcal{S}[t_1], \ldots, \mathcal{S}[t_m])
\end{array}$$

The intermediate trees produced by the new 2-mtt P'' now contain auxiliary occurrences of the symbols α_i and σ_m which are then evaluated by the transformation YIELD. A prerequisite for this decomposition to work is that the newly constructed 2-mtt P'' never gets "stuck" meaning that it never reaches a node v with a function q for which no rule is applicable. This can be achieved by adding for all pairs (q, b) for which P' does not have a rule with left-hand side $q(\mathsf{lab}_b(x_1))$ the new rule:

$$q(\mathsf{lab}_b(x_1)) \to \perp$$

where "$\perp$" is a new symbol denoting *undefined*. The function YIELD is meant to evaluate the formal substitution symbols by interpreting the leaf labels α_j as variables and the symbols σ_k as formal substitution operations. In fact, the transformation YIELD can be implemented by a deterministic mtt with initial action $\text{YIELD}_0(z)$ where for $k \geq 0$, the function YIELD_k has k extra arguments $y_1, \ldots, y_k$ and is defined as:

$$\begin{array}{lcl}
\text{YIELD}_k(\alpha_j, y_1, \ldots, y_k) & \to & y_j \quad \text{if } j \leq k \\
\text{YIELD}_k(\sigma_m(z_0, \ldots, z_m), y_1, \ldots y_k) & \to & \\
\quad \text{YIELD}_m(z_0, \ \text{YIELD}_k(z_1, y_1, \ldots, y_k), \ldots, & & \\
\quad\quad \text{YIELD}_k(z_m, y_1, \ldots, y_k)) & & \\
\text{YIELD}_k(b, y_1, \ldots, y_k) & \to & b \\
\text{YIELD}_k(a(z_1, z_2), y_1, \ldots, y_k) & \to & \\
\quad a(\ \text{YIELD}_k(z_1, y_1, \ldots, y_k), \text{YIELD}_k(z_2, y_1, \ldots, y_k)) & &
\end{array}$$

with $j = 1, \ldots, n$ and a, b from the output alphabet of P'. Note in particular, that none of the functions YIELD_k is defined for the input symbol $\perp$.

Returning to our example program, the simulating rule (2) is now rewritten as a 2-mtt rule without accumulating parameters:

$$\begin{array}{l}
q(\mathsf{lab}_{\langle \mathtt{mbox}, \ldots \rangle}(x_1)) \to \\
\quad \mathtt{mbox}(\sigma_1(\mathsf{up}^{1,copy}_{\psi,\ldots}(x_2 = \mathsf{father}(x_1)), \\
\quad\quad \sigma_0(\mathsf{down}^{copy}_{\psi,\ldots}(x_2 = \mathsf{child}_1(x_1))) \ @ \\
\quad\quad \sigma_0(\mathsf{down}^{copy}_{\psi,\ldots}(x_2 = \mathsf{child}_2(x_1))))), e)
\end{array} \tag{3}$$

Note that the symbol σ_0 represents formal substitution of 0 parameters and therefore can also be omitted.

The new deterministic 2-mtt P'' without accumulating parameters now can be transformed into a total and deterministic mtt M. Assume that P'' has functions $q_1, \ldots, q_n$. For each of these n functions, the mtt M has two variants:

- $q_i^{(0)}$ is used if the current node is the root node, i.e., no up-moves can be performed and therefore no parameters are necessary;
- q_i is used for every other node and has n accumulating parameters in order to capture all possible next actions.

On the basis of this set of functions, we replace each rule

$$q_\nu(\mathsf{lab}_b(x_1)) \to t$$

with $b = \langle a, j, \ldots \rangle$ a symbol of the (annotated) input, by the new rules

$$\begin{array}{lcll}
q_\nu^{(0)}(b(z_1, z_2)) & \to & \mathcal{U}_0[t] & \text{if } j = 0 \\
q_\nu(b(z_1, z_2), y_1, \ldots, y_n) & \to & \mathcal{U}[t] & \text{for } j \neq 0
\end{array}$$

where $\mathcal{U}[.]$ is defined by:

$$\begin{array}{lcl}
\mathcal{U}[b'(A_1, \ldots, A_k)] & = & b'(\mathcal{U}[A_1], \ldots, \mathcal{U}[A_k]) \\
\mathcal{U}[q_\mu(x_2 = \mathsf{father}(x_1))] & = & y_\mu \\
\mathcal{U}[q_\mu(x_1 = x_2)] & = & \mathcal{U}[t_\mu] \\
\mathcal{U}[q_\mu(x_2 = \mathsf{child}_i(x_1))] & = & q_\mu(z_i, \mathcal{U}[t_1], \ldots, \mathcal{U}[t_n])
\end{array}$$

The tree t_i in the two last rules of $\mathcal{U}[.]$ is the right-hand side of the q_i-rule of P' with match pattern $\mathsf{lab}_b(x_1)$ (which by construction always exists). Thus, when calling the transformation $\mathcal{U}[.]$ for one right-hand side, it may recursively start to evaluate other right-hand sides for parameter positions. Since, however, the 2-mtt P' is assumed to be deterministic, P' will fail to *terminate* whenever it processes the same node with the same function q_i more than once. For representing non-terminating computations, we again use the auxiliary symbol $\perp$. Thus, we replace the $(n+1)$-th recursive application of $\mathcal{U}[.]$ by $\perp$. The definition of $\mathcal{U}_0[.]$ is similar and therefore is omitted. The rules for leaf nodes $\langle e, j, T \rangle$ are rewritten analogously, with only difference that no variables z_i occur in the corresponding right-hand sides.

Going back to our example, the 2-mtt rule (3) (without accumulating parameters) now become the following mtt rule (assuming that $\mathsf{up}^{1,copy}_{\psi,\ldots}$ is the k-th function):

$$q(\langle\mathtt{mbox},\ldots\rangle(z_1,z_2),y_1,\ldots,y_k,\ldots,y_n) \rightarrow \mathtt{mbox}(\sigma_1(y_k,\ \mathsf{down}^{copy}_{\psi,\ldots}(z_1,s_1,\ldots,s_n)\ @\ \mathsf{down}^{copy}_{\psi,\ldots}(z_2,s_1,\ldots,s_n)),e) \quad (4)$$

where s_i are results of applying $\mathcal{U}[.]$ to the right-hand side of the i-th function for nodes labeled $\langle\mathtt{mbox},\ldots\rangle$.

Summarizing, we have constructed from the deterministic 2-mtt P' a total and deterministic mtt together with a transformation YIELD (which also can be implemented by a deterministic, not necessarily total, mtt) whose composition is equivalent to P':

Lemma 6 *The transformation of the deterministic 2-mtt P' can be effectively decomposed into* YIELD $\circ\, \tau_M$, *where M is a total and deterministic mtt.* $\diamond$

Lemma 6 essentially has already been proved in [15]. Together with Theorem 5 we obtain our main decomposition result for deterministic TL programs.

Theorem 7 *For every deterministic* TL *program P there is effectively a total and deterministic mtt M such that*

$$\tau_P = \text{APP} \circ \text{YIELD} \circ \tau_M.$$

Proof. According to our constructions so far, we have decomposed the transformation realized by a TL program P into a composition APP$\circ$YIELD$\circ\tau_M\circ$REL. The theorem follows from the fact that mtts are effectively closed under relabelings of the input from REL, by Theorem 4.21 and Corollary 4.10 of [14]. $\diamond$

7. NONDETERMINISTIC 2-MTTS

In order to prove the second part of Theorem 4, it suffices to generalize our decomposition technique from deterministic 2-mtts to nondeterministic ones. Interestingly, it turns out that ordinary macro tree transducers are not sufficient for such a decomposition. To see this, consider the nondeterministic 2-mtt:

$$\begin{array}{lcl} q(\mathsf{lab}_a(x_1)) & \rightarrow & b(q(x_1 = x_2)) \\ q(\mathsf{lab}_a(x_1)) & \rightarrow & a \end{array}$$

This program transforms a fixed node a into infinitely many outputs $a, b(a), b(b(a)),\ldots$. Such transformations *cannot* be expressed by our notion of macro tree transducers. It can be expressed, however, by a generalization of the mtt which is not obliged to proceed to subterms of their argument parameter in every step, but may process the same argument node arbitrarily often. Such mtts are called *stay macro tree transducers* (s-mtts for short) [11]. Clearly, we could formally define s-mtts as 2-mtts without select patterns of the form $x_2 = \mathsf{father}(x_1)$. However, since s-mtts are very close to mtts, we prefer to extend the mtt formalism by a new left-hand side $q(z \text{ as } a)$ which binds z to the current node; then, $q'(z)$ in the corresponding right-hand side means to call q' at the current node. An s-mtt for the above example TL program looks as follows:

$$\begin{array}{lcl} q(z \text{ as } a) & \rightarrow & b(q(z)) \\ q(z \text{ as } a) & \rightarrow & a \end{array}$$

Following the approach of Section 6, we propose the strategy:

1. decompose the 2-mtt P' into a 2-mtt P'' without accumulating parameters followed by a function for evaluation of symbolic substitutions;
2. simulate P'' with an s-mtt M.

A construction for the decomposition under 1) has already been provided by Engelfriet and Vogler in [15]. The idea is as follows. First, we collect all rules

$$q(\mathsf{lab}_b(x_1),y_1,\ldots,y_k) \rightarrow t_i \quad , \quad i = 1,\ldots,r$$

for a function q with the same left-hand side into a single rule:

$$q(\mathsf{lab}_b(x_1),y_1,\ldots,y_k) \rightarrow t_1 \mid \ldots \mid t_r$$

where "|" is a new binary operator symbol (written infix) denoting binary choice. The resulting transducer then is obviously deterministic. To the initial action as well as to these rules, we then apply the transformation $\mathcal{S}[.]$ from the last section. Additionally and in order to allow termination of a function call at any moment, we add for *every* pair (q,b), a rule

$$q(\mathsf{lab}_b(x_1)) \rightarrow \bot$$

as a second rule. Intuitively, the resulting transducer does not only delay substitution into formal variables but also the choice between different alternatives for a given function call. The substitutions as well as the formal choice operators are to be evaluated by second transformation YIELD$'$ which can be implemented by an mtt which essentially equals the mtt for the transformation YIELD from Section 6, which, however, has the following additional rules for "|":

$$\text{YIELD}_k(z_1 \mid z_2, y_1,\ldots,y_k) \rightarrow \text{YIELD}_k(z_i,y_1,\ldots,y_k)$$

for $i = 1,2$. In particular, the mtt for YIELD$'$ is no longer deterministic. It remains to provide an implementation of the 2-mtt P'' without accumulating parameters by a suitable stay macro tree transducer. Indeed, we here essentially use the same construction as in Section 6. The only difference is that now the transformations $\mathcal{U}_0[.]$ and $\mathcal{U}[.]$ of the right-hand sides can be simplified in such a way that we now can use the pattern variable z of the left-hand side for expressing a repeated transformation of the current node. Thus, we modify the definition of $\mathcal{U}[.]$ to:

$$\begin{array}{lcl} \mathcal{U}[b'(A_1,\ldots,A_k)] & = & b'(\mathcal{U}[A_1],\ldots,\mathcal{U}[A_k]) \\ \mathcal{U}[q_\mu(x_2 = \mathsf{father}(x_1))] & = & y_\mu \\ \mathcal{U}[q_\mu(x_1 = x_2)] & = & q_\mu(z) \\ \mathcal{U}[q_\mu(x_2 = \mathsf{child}_i(x_1))] & = & \\ \multicolumn{3}{r}{q_\mu(z_i, q_1(z,y_1,\ldots,y_n),\ldots,q_n(z,y_1,\ldots,y_n))} \end{array}$$

Since s-mtts are closed under relabelings REL of the input for REL $\in$ REL [11], we arrive at the following theorem:

Theorem 8 *The transformation of every nondeterministic* TL *program P can be decomposed into the composition:*

$$\text{APP} \circ \text{YIELD}' \circ \tau_M$$

where M is an s-mtt that can be effectively constructed from P. $\diamond$

This completes the proof of Theorem 4.

8. RELATED WORK

There are a number of well established XML transformation languages with varying support for type checking. XSLT [9, 21] is a *rule-based* language for transforming XML documents. Transformation rules in XSLT are similar to those of TL. For match and select patterns XPath 1.0 [7] and 2.0 [3] are used, which are languages for addressing nodes of an XML document. Type checking has not been supported for XSLT. Only recently a tool for type checking XSLT has been designed [28].

XQuery [6] is a strongly-typed functional language for querying XML documents which is again based on XPath. Type checking is performed via *forward* type inference; it is thus only approximative and type errors might occur at run-time. Type inference is also used in the XDuce [20] and CDuce [2, 18] family of functional languages. These languages extend the pattern matching mechanism of functional languages by regular expression constructs. Another functional approach to XML processing is XMλ [26]. Here, the type information is mapped onto (extended) Haskell types. As in XQuery and XDuce/CDuce, type inference is approximate.

Besides in the publications about type checking k-pebble tree transducers [27, 11], *inverse* type inference has also been used by Tozawa for a subset of XSLT which essentially consists of top-down tree transformations [34].

9. CONCLUSION

We introduced the small but expressive tree transformation language TL which subsumes the tree transformation core of most existing XML transformation languages. Any TL program can be effectively decomposed into three (stay) macro tree transducers, independent of the complexity of the transformation or the used patterns. Applying the known result that regularity is preserved by inverse images of (stay) mtts, an elegant procedure for type checking TL programs is obtained. It remains a challenging engineering problem whether this decomposition gives rise to an implementation which is also sufficiently efficient on real world transformations. We are currently working on the implementation.

10. REFERENCES

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 2000.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proc. ICFP'03*, pages 51–63. ACM Press, 2003.

[3] A. Berglund and S. Boag et.al., editors. XML Path Language (XPath) 2.0. W3C Working Draft, World Wide Web Consortium, November 2003. Available online http://www.w3.org/TR/xpath20.

[4] A. Berlea and H. Seidl. fxt – A Transformation Language for XML Documents. *Journal of Computing and Information Technology*, 10(1):19–35, 2002.

[5] R. Bloem and J. Engelfriet. Monadic Second Order Logic and Node Relations on Graphs and Trees. pages 144–161. LNCS 1261, Springer-Verlag, 1997.

[6] 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/.

[7] 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.

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

[9] J. Clark, editor. XSL Transformations (XSLT) 1.0. W3C Recommendation, World Wide Web Consortium, November 1999. Available online http://www.w3.org/TR/xslt.

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

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

[12] J. Engelfriet and S. Maneth. Macro Tree Translations of Linear Size Increase are MSO Definable. *SIAM J. Comput.*, 32:950–1006, 2003.

[13] J. Engelfriet and B. Samwel. Personal communication (work in progress). 2004.

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

[15] J. Engelfriet and H. Vogler. Pushdown Machines for the Macro Tree Transducer. *Theoret. Comp. Sci.*, 42:251–368, 1986.

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

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

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

[19] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics; Formal Models Based on Tree Transducers*. Springer-Verlag, 1998.

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

[21] M. Kay, editor. XSL Transformations (XSLT) 2.0. W3C Working Draft, World Wide Web Consortium, November 2003. Available online http://www.w3.org/TR/xslt20.

[22] A. Kühnemann and H. Vogler. Synthesized and Inherited Functions. A new Computational Model for Syntax-Directed Semantics. *Acta Inf.*, 31:431–477, 1994.

[23] S. Maneth. The Macro Tree Transducer Hierarchy Collapses for Functions of Linear Size Increase. In *Proc. FSTTCS'03*, pages 326–337. LNCS 2914, Springer-Verlag, 2003.

[24] S. Maneth and F. Neven. Structured Document Transformations Based on XSL. In *Proc. DBPL'99*, pages 80–98. LNCS 1949, Springer-Verlag, 1999.

[25] W. Martens and F. Neven. Frontiers of Tractability for Typechecking Simple XML Transformations. In *Proc. PODS'04*, pages 23–34. ACM Press, 2004.

[26] E. Meijer and M. Shields. XMλ: A Functional Language for Constructing and Manipulating XML Documents. 1999. Available online http://www.cse.ogi.edu/~mbs/pub/xmlambda/.

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

[28] A. Møller and M. I. Schwartzbach. The Design Space of Type Checkers for XML Transformation Languages. In *Proc. ICDT'05*, 2005. To appear.

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

[30] A. Neumann and A. Berlea. fxgrep 4.0. Source Code, 2004.

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

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

[33] J.W. Thatcher and J.B. Wright. Generalized Finite Automata with an Application to a Decision Problem of Second Order Logic. *Mathematical Systems Theory*, 2:57–82, 1968.

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

APPENDIX

A. SKETCH OF PROOF

In the following we present a sketch of the correctness proof of the construction to compile global selections of the TL program into local movements (Section 5.4).

Let f be an arbitrary but fixed binary tree representation of the input forest which is annotated as described in Section 5.3. For two nodes v_1 and v_2 of f labeled by $\langle \_, \_, T_1, \_, \_\rangle$ and $\langle \_, \_, T_2, \_, \_\rangle$, respectively, the relation $g_f(v_1, v_2) \subseteq T_1 \times T_2$ relates states in T_1 at v_1 and T_2 at v_2 which are possibly connected through a common run r on the input f of the match automaton $\mathcal{P}$ (cf. Section 5.2). This relation is defined by

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

where s_1, s_2 are states of $\mathcal{P}$.

Claim 1:

$$\text{APP}([\![\mathsf{down}^q_{\psi,h}]\!]_f(v, T_1, \ldots, T_k)) = \text{APP}([\![q]\!]_f(u_1, T_1, \ldots, T_k)) \ldots \text{APP}([\![q]\!]_f(u_l, T_1, \ldots, T_k))$$

for all inputs f, and $v, u_1, \ldots, u_l \in \mathcal{N}(f)$, a function q of the TL program and actual parameters T_i. $u_1 < \ldots < u_l$ are exactly the nodes u' in document order occurring in the subtree f/v of f rooted at v such that

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

i.e., the sequence of pairs $(v, u_1), \ldots, (v, u_l)$ represents all matches of the binary select pattern ψ in the input forest. The proof is by induction on the depth n of the subtree f/v. For $n = 0$, consider the node $u' = v$. Then

$$[\![\mathsf{down}^q_{\psi,h}]\!]_f(v, T_1, \ldots, T_k)$$

either equals $[\![q]\!]_f$ applied to v and $T_1, \ldots, T_k$ given that $h \cap B_\psi \neq \emptyset$, or equals the empty sequence ϵ otherwise. Since $g_t(v, u')$ is the identity relation, the assertion follows. If $n > 0$, v is not a leaf of f and therefore has two children v_1 and v_2, respectively. Let us first consider the case where $h \cap B_\phi = \emptyset$. Then, all descendants u' of v with

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

either are descendants of v_1 or of v_2. Accordingly, the sequence of nodes $u_1, \ldots, u_l$ is the concatenation of two sequences $u_1, \ldots, u_{l'}$ and $u_{l'+1}, \ldots, u_l$ where the nodes in the first sequence are descendants of v_1 and the others are descendants of v_2. By inspecting the definition of g'_i and g_t, we verify that

$$g_f(v, u') \circ h = g_f(v_i, u') \circ g'_i$$

whenever u' is a descendant of v_i. Therefore by induction hypothesis,

$$\text{APP}([\![\mathsf{down}^q_{\psi,g'_1}]\!]_f(v_1, T_1, \ldots, T_k)) = \text{APP}([\![q]\!]_f(u_1, T_1, \ldots, T_k)) \;\ldots\; \text{APP}([\![q]\!]_f(u_{l'}, T_1, \ldots, T_k))$$

and

$$\text{APP}([\![\mathsf{down}^q_{\psi,g'_2}]\!]_f(v_2, T_1, \ldots, T_k)) = \text{APP}([\![q]\!]_f(u_{l'+1}, T_1, \ldots, T_k)) \;\ldots\; \text{APP}([\![q]\!]_f(u_l, T_1, \ldots, T_k))$$

Since by definition,

$$\begin{aligned}[\![\mathsf{down}^q_{\psi,h}]\!]_f(v, T_1, \ldots, T_k) = {} & [\![\mathsf{down}^q_{\psi,g'_1}]\!]_f(v_1, T_1, \ldots, T_k) \;@\\ & [\![\mathsf{down}^q_{\psi,g'_2}]\!]_f(v_2, T_1, \ldots, T_k)\end{aligned}$$

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

$$\begin{aligned}&\text{APP}(\;[\![\mathsf{down}^q_{\psi,g'_1}]\!]_f(v_1, T_1, \ldots, T_k) \;@\\ &\qquad [\![\mathsf{down}^q_{\psi,g'_2}]\!]_f(v_2, T_1, \ldots, T_k))\\ &= \text{APP}([\![q]\!]_f(u_2, T_1, \ldots, T_k)) \;\ldots\; \text{APP}([\![q]\!]_f(u_l, T_1, \ldots, T_k))\end{aligned}$$

Since now by definition,

$$\begin{aligned}[\![\mathsf{down}^q_{\psi,h}]\!]_f(v, T_1, \ldots, T_k) = {} & [\![q]\!]_f(v, T_1, \ldots, T_k) \;@\\ & [\![\mathsf{down}^q_{\psi,g'_1}]\!]_f(v_1, T_1, \ldots, T_k) \;@\\ & [\![\mathsf{down}^q_{\psi,g'_2}]\!]_f(v_2, T_1, \ldots, T_k)\end{aligned}$$

the assertion of Claim 1 again follows.

Assume that the node v of f is a descendant of the node v_j which in turn is the j-th child of the node v'. Assume further that $h = g_t(v, v_j)$ and that the value of the parameter T_{k+1} satisfies:

$$\text{APP}(T_{k+1}) = \text{APP}([\![q]\!]_t(u_1, T_1, \ldots, T_k)) \ldots \text{APP}([\![q]\!]_t(u_l, T_1, \ldots, T_k))$$

where $u_1 < \ldots < u_l$ are the descendants u' of v_j with

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

Thus the sequence $u_1 < \ldots < u_l$ consists of all descendants u' of v_j in document order such that (v, u') is a match of ψ in f. Assume that v' is labeled $\langle a, i, T, S_1, S_2\rangle$. Consider the call

$$[\![\mathsf{up}^{j,q}_{\psi,h}]\!]_f(v', T_1, \ldots, T_k, T_{k+1}).$$

Now let h_j, g_j be defined according to the possible right-hand side of $\mathsf{up}^{j,q}_{\psi,h}$ and the label of the tree rooted at v'. Moreover, define T'_{k+1} as the new value of the $k+1$-st accumulating parameter in the recursive call for the father of v' (in case $i > 0$) or the complete right-hand side (in case $i = 0$). Thus,

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

where $\underline{T}$ abbreviates the sequence $T_1, \ldots, T_k$ of actual parameters. We claim:

Claim 2:

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

   $$\text{APP}(T'_{k+1}) = \text{APP}([\![q]\!]_f(u_1, T_1, \ldots, T_k)) \ldots \text{APP}([\![q]\!]_f(u_m, T_1, \ldots, T_k))$$

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

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

The first item of the claim follows from the property of h by inspection of the definitions of the h_j and g_j. For the second item, we recall from Claim 1 that the call

$$[\![\mathsf{down}^q_{\psi,g_j}]\!]_f(v_{3-j}, T_1, \ldots, T_k)$$

returns (the representation of) the list of all calls

$$[\![q]\!]_f(u', T_1, \ldots, T_k)$$

where $g_f(v_{3-j}, u') \circ g_j \cap B_\psi \neq \emptyset$. From the definitions of g_j and g_t, we verify that

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

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

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

$$[\![A]\!]_t(v, T_1, \ldots, T_k) = \text{APP}([\![[A]_j]\!](v, T_1, \ldots, T_k)),$$

where $[.]_j$ is our transformation of right-hand sides in order to compile the global selection steps of the TL program into local movements (cf. Section 5.4). The only difficult case is the simulation of calls

$$q'(\psi, A_1, \ldots, A_k).$$

First, let us assume that the current node v is no match of ψ. For that case, we iteratively may apply Claim 2 to the ancestors of the node v. Thus, we deduce for

$$\begin{array}{lcl} h & = & \{(s,s) \mid s \in T\} \\ g_i & = & \{(s, s_i) \in T \times S_i \mid \exists\, s_{3-i} \in S_{3-i} \,:\, (s, a, s_1 s_2) \in \delta\} \end{array}$$

that the calls:

$$[\![\mathsf{down}^{q'}_{\psi,g_1}]\!]_f(v_1, T_1, \ldots, T_k) \mathbin{@} [\![\mathsf{down}^{q'}_{\psi,g_2}]\!]_f(v_2, T_1, \ldots, T_k)$$

if v is the root of t or

$$\begin{array}{ll} [\![\mathsf{up}^{j,q'}_{\psi,h}]\!]_f(v^-, & T_1, \ldots, T_k, \\ & [\![\mathsf{down}^{q'}_{\psi,g_1}]\!]_f(v_1, T_1, \ldots, T_k)) \mathbin{@} \\ & [\![\mathsf{down}^{q'}_{\psi,g_2}]\!]_f(v_2, T_1, \ldots, T_k)) \end{array}$$

if v^- is the father of v and v_1, v_2 the first and second child, respectively, result in a representation of

$$[\![q]\!]_f(u_1, T_1, \ldots, T_k) \ldots [\![q]\!]_f(u_n, T_1, \ldots, T_k)$$

where $u_1 < \ldots < u_n$ is the sequence of nodes u' such that (v, u') are precisely all matches of ψ in t, i.e., $g_t(v, u') \cap B_\psi \neq \emptyset$ — which we needed to verify. If on the other hand v is a match of ψ then we have to include the call

$$[\![q]\!]_f(v, T_1, \ldots, T_k)$$

into the accumulating parameter of up:

$$\begin{array}{ll} [\![\mathsf{up}^{j,q'}_{\psi,h}]\!]_f(v^-, & T_1, \ldots, T_k, \\ & [\![q]\!]_f(v, T_1, \ldots, T_k) \mathbin{@} \\ & [\![\mathsf{down}^{q'}_{\psi,g_1}]\!]_f(v_1, T_1, \ldots, T_k)) \mathbin{@} \\ & [\![\mathsf{down}^{q'}_{\psi,g_2}]\!]_f(v_2, T_1, \ldots, T_k)) \end{array}$$

or into the final result if v is the root of f:

$$\begin{array}{l} [\![q]\!]_f(v, T_1, \ldots, T_k) \mathbin{@} \\ [\![\mathsf{down}^{q'}_{\psi,g_1}]\!]_f(v_1, T_1, \ldots, T_k) \mathbin{@} [\![\mathsf{down}^{q'}_{\psi,g_2}]\!]_f(v_2, T_1, \ldots, T_k) \end{array}$$

This ends the proof sketch for compiling global selections of TL programs.