Institut für Informatik der Technischen Universität München
Lehrstuhl Informatik II

Type Checking XML Transformations

Thomas Perst

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Alfons Kemper, Ph.D.
Prüfer der Dissertation:
1. Univ.-Prof. Dr. Helmut Seidl
2. Ass. Prof. Dr. Frank Neven, Univ. of Limburg/Belgien

Die Dissertation wurde am 04.09.2006 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.02.2007 angenommen.

Acknowledgments

I wish to thank Helmut Seidl for inspiring me to conduct my research in the field of XML transformations, and I feel deep gratitude towards him for guiding me like Virgil through the circles of tree transducers. He employed me on a project position providing ideal conditions for conducting my research. It was funded by the "Deutsche Forschungs Gemeinschaft" under the headword "MttCheck".

I am grateful to many colleagues who in one way or another contributed to this thesis. Special thanks to Thomas Gawlitza, Ingo Scholtes, and Peter Ziewer for careful reading of the manuscript and for their helpful comments and suggestions.

Last but not least, I want to thank Jennifer for her love and constant support on the stony path of writing this work.

Abstract

XML documents are often generated by some application in order to be processed by a another program. For a correct exchange of information it has to be guaranteed that for correct inputs only correct outputs are produced. The shape of correct documents, i.e., their type, is usually specified by means of schema languages.

This thesis is concerned with methods for statically guaranteeing that transformations are correct with respect to pre-defined types. Therefore, we consider the XML transformation language TL abstracting the key features of common XML transformation languages.

We show that each TL program can be decomposed into at most three stay macro tree transducers. By means of classical results for stay macro tree transducers, this decomposition allows to formulate a type checking algorithm for TL. This method, however, has even for small programs an exorbitant run-time.

Therefore, we develop an alternative approach, which allows at least for a large class of transformations that they can be type checked in polynomial time. The developed algorithms have been implemented and tested for practical examples.

Zusammenfassung

XML-Dokumente werden oft von Anwendungen erzeugt, um von anderen Anwendungen konsumiert zu werden. Für einen korrekten Informationsaustausch muss garantiert werden, dass für korrekte Eingaben nur korrekte Ausgaben erzeugt werden. Die vorliegende Arbeit untersucht Methoden, um statisch zu garantieren, dass eine Transformation nur korrekte Ausgaben erzeugt. Dazu betrachten wir die Sprache TL, die die Eigenschaften gängiger XML-Transformationsprachen abstrahiert. Wir zeigen, dass jedes TL Programm in höchstens drei stay macro tree transducers zerlegt werden kann. Diese Zerlegung erlaubt, ein Typüberprüfungsverfahren zu konstruieren. Dieses weist allerdings selbst für kleine Programme exorbitante Laufzeiten auf. Darum entwickeln wir einen alternativen Ansatz, der zumindest für eine große Klasse praktischer Transfromationen Typüberprüfung in polynomieller Zeit erlaubt. Beide Ansätze wurden implementiert und an praktischen Beispielen getestet.

Contents

1

Introduction

When Axel Thue published his article about logical problems [Thu10;ST00] in 1910, he certainly would not have expected that his ideas of trees and their rewriting will become as popular as 90 years later with the introduction of the *Extensible Markup Language* (XML). The W3C[1], an international consortium to develop Web standards, describes this language as "*a simple, very flexible text format derived from SGML. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of the variety of data on the Web and elsewhere*" [Qui06]. Moreover, by proposing XML the W3C fulfils its mission "*to lead the World Wide Web to its full potential by developing protocols and guidelines that ensure long-term growth of the Web,*" because since the publication of the standard in 1998, XML enjoys an increasing popularity in nearly all fields of information technologies.

XHTML is an XML format for designing Web pages [Pem02], XSL:FO is a language for outlining general layouts [Adl01], and MATHML provides elements for typesetting mathematical formulas [CIMP03]. With SMIL it is possible to create interactive audio-visual presentations [Hos98]. The bibliographic records of the Computer Science bibliography DBLP[2] are organized with XML [Ley02]. The office-suite of OpenOffice.org [Man06] uses the Open Document Format, an XML-based file format for office applications [DBO06]. Even this small selection of XML applications shows that Vianu does not exaggerate if he denotes XML as the *lingua franca* of the Web [Via01].

Technically, XML is a meta-language for storing structured data in a sequential format. Therefore, data are grouped into elements, and each element is enclosed between an opening and closing tag indicating the start and the end of such a logical component. Since elements can be nested, every XML document can be naturally considered as a tree whose nodes are the tags and

[1] Worls Wide Web Consortium

[2] Digital Bibliography & Library Project

whose leafs are the data. Once a document has been structured with XML elements, the actual XML share of the document is called *markup*.

One of XML's design goals and advantages over other data formats is that XML is *typed*. This means, it is possible to specify where each element may occur in a document. For this purpose the W3C recommendation suggests document type definitions (DTDs), which provide rules for how the XML elements, attributes, and other data are defined and logically related in a document [BPSM$^+$04]. A document adhering to the XML syntax specifications and the rules outlined by its DTD is considered to be *valid*. More expressive typing mechanisms are XML Schema [FW04] and Relax NG [Jam01]. They provide more base types for data values like integers, decimals or reals; moreover, they allow users to define their own types by combining base types. Validity, however, can be checked by a type-aware parser which tests for every element whether the sequence of its children conforms to the specifications made in the type definition.

Another advantage of XML certainly is its extensability that allows to create and format your own document markups. The Web publishing language HTML, for example, consists of a fixed set of elements with a more or less predefined meaning. XML, on the other hand, allows to create new markup at will and, moreover, to specify in a DTD or other schema how this markup is related inside a document. Since "*XML data exchange is always done in the context of a fixed type*" [AMF$^+$01; AMF$^+$03], a common scenario is that a community (or industry) agrees on a certain type, and all members have to create documents that are of that type. One possibility to asure validity, is to parse every created document and to check if it conforms to the specified type. This method, however, is very time consuming, because every document has to be reread, which takes some time for large documents. Moreover, the same document is repeatedly checked whenever it is created.

Since XML documents are often generated dynamically by some program [Suc02], e.g. as result of the transformation induced by a stylesheet processor, it would be more convenient to check whether all XML documents generated by a program are valid with respect to a specified type. This is called the *type checking problem* or *static type checking* — in contrast to dynamically checking every generated document. For a program or transformation F, it asks whether for all documents d of an input domain D, the transformation result $F(d)$ is valid with respect to an output type T_{out}. More formally, it asks whether

$$\forall d \in D: \quad F(d) \in T_{\text{out}}?$$

The *type checker*, i.e., the program for solving the type checking problem, analyzes the transformation itself and the output type to decide whether all documents produced by F are valid. If the type checker gives a positive answer, transformation F is called *type-safe*. Besides the fact that a type-safe transformation will never raise run-time type errors, type checking also helps to guarantee that a program computes the transformation intended by the

programmer. The intended transformation is specified via an input and an output type. If the program is type-safe with respect to these types, it surely realizes the intended transformation. Guaranteeing type-safety might even be security relevant. Consider, for example, an XML document, which contains information about students — including their names, examination results, and the visited courses. In order to satisfy red tape, a lot of statistics are automatically generated. For data protection reasons, however, generated documents must not contain information so that examination results can be mapped to a student's name. This can be assured by type checking the transformations against an appropriate output type.

A closely related problem is the *type inference problem*. Here, it is necessary to compute for the given transformation F the type $F(D) = \{F(d) \mid d \in D\}$. Whenever this is possible for a transformation F, the type checking problem can be solved in the following way: first compute $F(D)$, and then check if this inferred type is a subset of the allowed output type.

Type checking techniques heavily depend on the used type model and transformation language. But instead of developing different type checking algorithms for every XML transformation language, our aim is to present an approach that is as general as possible. For that purpose, we focus in the context of this work on formal models which

(1) subsume most of the features of existing languages and also
(2) provide a tractable type checking problem.

As type model we prefer *recognizable tree languages* because they are a convenient abstraction of the popular XML typing facilities like DTD, XML Schema [FW04], or Relax NG [Jam01]. Recognizable tree languages define sets of trees and the membership to such a set can be decided by means of a finite state automaton [GS97].

By presenting our own transformation language TL together with a type checking algorithm, we obtain a general type checking mechanism for all languages that can be compiled into our formalism. From XSLT [XSL99] we borrow the idea to organize the transformation in *named functions* defined by rules. Each rule defines a translation of a set of nodes, which is given by a *match pattern*. Such a pattern, for example, specifies "all nodes that are labeled with `Chapter`" or "all nodes that are below a node labeled with `Doc`". When transforming the matched nodes, *select patterns* determine those subtrees where the transformation should continue. Match patterns as well as select patterns are defined by using *monadic second-order logic*, an extension of first-order logic with set variables [Tho97]. In order to accumulate intermediate results during the transformation, each rule can be enhanced with *parameters*. They allow to transport context information from the current node to a subsequent translation step. These four "ingredients"

- named functions,
- match patterns,

- select patterns, and
- accumulating parameters

form the core of our transformation language TL. The key idea of our type checking algorithm is to simulate TL transformations by compositions of top-down finite state tree transducers with parameters, which are also called *stay macro tree transducers* [EV85]. They can be considered as first-order functional programs performing a top-down traversal over their first argument while possibly accumulating temporary results in additional parameters. While TL is our specification language for XML transformations, the stay macro tree transducer can be viewed as "low-level" model to implement TL functions. By compiling a TL program (or transformation) into compositions of stay macro tree transducers, we obtain at the same time a solution for the type checking problem for TL, because for a composition of macro tree transducers it is decidable whether it produces only outputs of a predefined type [EV85].

The *pre-image* of a set T_{out} with respect to a transformation F is the set of all inputs d such that $F(d) \in T_{\text{out}}$. Since recognizability of sets of output trees is preserved by taking pre-images, the type checking problem can be solved as follows: first, compile TL into a composition τ of stay macro tree transducers. Then determine a finite automaton for the set

$$U = \tau^{-1}(\overline{T_{\text{out}}}),$$

where $\overline{T_{\text{out}}}$ is the complement of the desired output type. Hence, U characterizes all inputs for which τ returns results not conforming to T_{out}. This method is called *inverse type inference* [MSV00]. For a given input type T_{in}, it then suffices to check whether or not the intersection $T_{\text{in}} \cap U$ is empty. In particular, no wrong results are produced if the intersection is empty. The main limitation of this type checking approach is its high complexity due to the non-elementary size of the automaton accepting the pre-image.

Therefore, we investigate classes of stay macro tree transducers for which type checking can be done in acceptable time. For a stay macro tree transducer F, which is restricted in copying its input, we present a type checking algorithm based on forward type inference. For a specified input type T_{in}, this algorithm exactly determines the set $F(T_{\text{in}})$ of outputs generated by the transducer, i.e.,

$$F(T_{\text{in}}) = \{F(t) \mid t \in T_{\text{in}}\}.$$

Then F type checks with respect to T_{in} and a predefined output type T_{out}, if the set $F(T_{\text{in}})$ is contained in T_{out}. Since this inclusion is decidable, we have an algorithm for exact type checking tree transformations, which can be expressed by one stay macro tree transducer with restricted copying.

This work is devided into 8 chapters. Chapter 2 provides some preliminaries. The transformation language TL is defined and characterizations of the induced translations are given in Chapter 3. Then a definition of stay

macro tree transducers is given and extensions of this model are discussed. Inductive characterizations are given for the derivation relation of stay macro tree transducers. In Chapter 5 type checking of TL by means of decomposition into stay macro tree transducers is studied. Chapter 6 is concerned with forward type inference for restricted classes of stay macro tree transducers. In the second last Chapter 7 we consider how to implement the presented algorithms in order to test them for practical examples. Finally, Chapter 8 gives some conclusions and remarks.

2

Preliminaries

In this chapter we briefly introduce the basic concepts used throughout this work. We first formalize the terms "*tree*" and "*forest*" and precisely define the automata model accepting tree languages. Then, we present an extension of first-order logic, called monadic second-order logic.

2.1 Trees and Forests

Since every XML document is a sequential representation of a tree [Suc01], we have to formalize the intuition behind this term. Figure 2.1 shows the tree induced by the following XML document:

Example 2.1

⟨Doc⟩
  ⟨Chapter⟩
    ⟨Title⟩Introduction⟨/Title⟩
    ⟨Body⟩⟨Par⟩ ... ⟨/Par⟩⟨/Body⟩
  ⟨/Chapter⟩
  ⟨Chapter⟩
    ⟨Title⟩Preliminaries⟨/Title⟩
    ⟨Body⟩⟨Par⟩ ... ⟨/Par⟩⟨Par⟩ ... ⟨/Par⟩⟨/Body⟩
  ⟨/Chapter⟩
⟨/Doc⟩

XML documents describe *unranked trees*, which means that the number of direct successors of a node is *not* fixed. In our example this property is illustrated by the fact that the two occurrences of the element labeled Body have different numbers of children. Note that the trees induced by the structure of an XML document are *ordered*, i.e., the order of the children of a node is relevant.

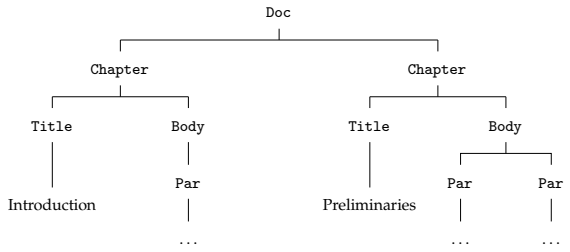

Fig. 2.1. The tree representation of the XML document given in Example 2.1.

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

Definition 2.2 (Forests) Let Σ be a finite unranked alphabet. Then the set $\mathcal{F}_\Sigma$ of *forests* f over Σ is recursively defined as:

$$\begin{aligned} t &::= \mathtt{a}\langle f \rangle, \\ f &::= \epsilon \mid t\, f \end{aligned}$$

where $\mathtt{a} \in \Sigma$ and ϵ denotes the empty forest. ◁

In order to illustrate our definition, we repeat the document of Example 2.1 in this "unranked" or "forest syntax":

```
Doc⟨
  Chapter⟨
    Title⟨Introduction⟩
    Body⟨Par⟨...⟩⟩
  ⟩
  Chapter⟨
    Title⟨Preliminaries⟩
    Body⟨Par⟨...⟩Par⟨...⟩⟩
  ⟩
⟩
```

Given a forest $f = \mathtt{a}\langle f_1 \rangle\, f_2 \in \mathcal{F}_\Sigma$, the tree $t = \mathtt{a}\langle f_1 \rangle$ is called the *head* and the forest f_2 the *tail* of f. The symbol $\mathtt{a} \in \Sigma$ is the *label* of the head. The forest f_1 is the *content* of the tree t and the forest f_2 is its *right context*, i.e., the sequence of its right siblings.

Sometimes forests are also called *hedges*, e.g., by Murata [Mur01], Tozawa [Toz01] or in the publications of Brüggemann-Klein and Wood [BW04].

Since the same tree may occur several times as a subtree of a given forest, it is necessary to distinguish between a subtree and an occurrence of a subtree [GS84]. Assigning unique coordinates to the nodes of a forest makes it possible to indicate a certain occurrence of a subtree by the coordinates of its root. As *coordinates* we define the set $\Pi(f) \subseteq \mathbb{N}^*$ of all *paths* π in forest f

$$\begin{aligned}\Pi(\epsilon) &= \{\varepsilon\}\\ \Pi(t_1 \cdots t_n) &= \{\varepsilon\} \cup \{i\pi \mid 1 \leq i \leq n, \pi \in \Pi(f_i) \text{ for } t_i = \mathtt{a}_i\langle f_i\rangle\}\end{aligned}$$

where $\mathbb{N}^*$ is the set of strings (including the empty string) over the alphabet of positive natural numbers and ε denotes the empty string. The three occurrences of the `Par`-elements of our document in Example 2.1, have the following coordinates. The first occurrence has the coordinate 1121 because the root node has coordinate 1 and the `Par`-element can be reached on the path: to the left `Chapter`-element (1), to the `Body`-element (2), and then to `Par` (1). The other two occurrences have coordinates 1221 and 1222. Figure 2.2 shows the XML tree of Figure 2.1 together with its coordinates.

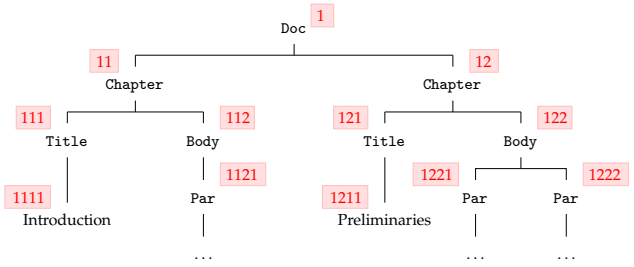

Fig. 2.2. The tree representation of the XML document given in Example 2.1 annotated with the coordinates of the nodes.

The *nodes* of a forest $f = t_1 \cdots t_n$ are then uniquely identified by the elements of the set $\mathcal{N}(f) = \Pi(f) \setminus \{\varepsilon\}$. For a given path (or node) $\pi \in \mathcal{N}(f)$, $f[\pi]$ is called the *subtree of f located at π* and is defined as

$$(t_1 \cdots t_n)[i\pi] = \begin{cases} t_i, & \text{if } \pi = \varepsilon \\ f_i[\pi], & \text{if } \pi \neq \varepsilon \text{ and } t_i = \mathtt{a}\langle f_i\rangle. \end{cases}$$

If the number of direct successors of a node is fixed, then the resulting trees are called *ranked*. A *ranked alphabet* is a set of symbols $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots\}$ together with a mapping $\mathit{rank} : \Sigma \to \mathbb{N}$ which associates each element of Σ with a positive integer, the *rank* of the symbol. The rank defines the number of direct successors of a node. For $k \in \mathbb{N}$ with $k \geq 0$, the subset Σ_k of the ranked alphabet Σ is the set of all symbols with rank k,

$$\Sigma_k = \{\mathtt{a} \in \Sigma \mid \mathit{rank}(\mathtt{a}) = k\}.$$

In order to indicate that a symbol $\mathtt{a} \in \Sigma$ is of rank k, we either write $\mathtt{a} \in \Sigma_k$, or we introduce the symbol with its rank as superscript, written $\mathtt{a}^{(k)}$. Note that for two ranks $k \neq n$, the symbols $\mathtt{a}^{(k)}$ and $\mathtt{a}^{(n)}$ are different.

A *ranked tree* over the ranked alphabet Σ has a designated root labeled with a symbol $\mathtt{a} \in \Sigma_k$ and k direct subtrees $t_1, \ldots, t_k$, written as $\mathtt{a}(t_1, \ldots, t_k)$. For a symbol of rank $k = 0$ we often write $\mathtt{a}$ rather than $\mathtt{a}()$.

Definition 2.3 (Ranked Trees) Let Σ be a finite ranked alphabet. Then the set $\mathcal{T}_\Sigma$ of *ranked trees* t over Σ is defined as:

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

where $\mathtt{a} \in \Sigma_0$, and for $k \geq 0$, $\mathtt{b} \in \Sigma_k$, and $t_1, \ldots, t_k$ are trees over Σ. ◁

Given a tree $t = \mathtt{a}(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$, the trees $t_i, 1 \leq i \leq k$, are the *children* or *sons* of t, while t itself is the *father* of all t_i. The symbol $\mathtt{a} \in \Sigma_k$ is the *label* of t.

As a special case, we define trees whose leafs can be elements of a given set S. For a set S of symbols or trees, $\mathcal{T}_\Sigma(S)$ is the set of trees defined inductively as:

(1) $S \subseteq \mathcal{T}_\Sigma(S)$ and
(2) for a symbol $\mathtt{a} \in \Sigma_k, k \geq 0$, and trees $t_1, \ldots, t_k \in \mathcal{T}_\Sigma(S)$, $\mathtt{a}(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma(S)$.

For the unranked case, the set $\mathcal{F}_\Sigma(S)$ is analogously defined.

The height of a tree is inductively defined on its structure. It intuitively is the number of nodes of the longest path from a leaf to the root.

Definition 2.4 (Tree Height) Let Σ be an alphabet. The *height* of a ranked tree t, is provided by the function $ht(t) : \mathcal{T}_\Sigma \to \mathbb{N}$, which is defined as:

(i) For $\mathtt{a} \in \Sigma_0$, $ht(\mathtt{a}) = 1$ and
(ii) for a symbol $\mathtt{b} \in \Sigma_k$ with $k \geq 1$ and $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$, $ht(\mathtt{b}(t_1, \ldots, t_k)) = 1 + \max\{ht(t_i) \mid i \in \{1, \ldots, k\}\}$. ◁

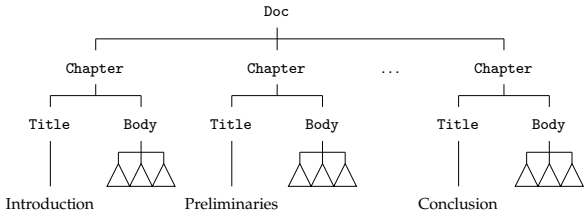

Fig. 2.3. An unranked tree.

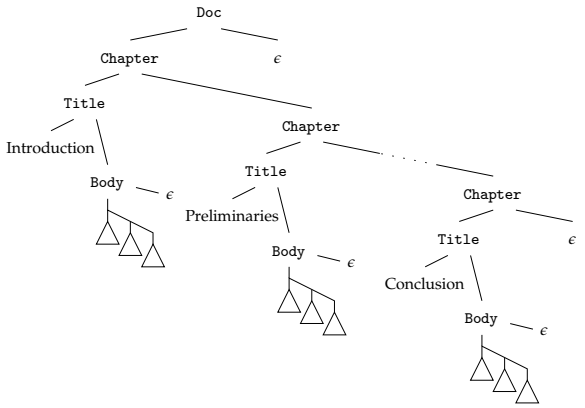

Fig. 2.4. The binary tree representation of the tree in Figure 2.3.

If we fix the rank of all symbols in Σ as 2 and have one element ϵ with rank $rank(\epsilon) = 0$ then we call the ranked trees over Σ *binary trees*.

Definition 2.5 (Binary Trees) Let ϵ be a symbol of rank 0. Let $\Sigma = \Sigma_2 \cup \{\epsilon\}$ be a finite ranked alphabet. Then the set $\mathcal{B}_\Sigma$ of *binary trees* b over Σ is defined as:

$$b \ ::= \ \epsilon \ \mid \ \mathtt{a}(b_1, b_2)$$

for $\mathtt{a} \in \Sigma_2$ and $b_1, b_2 \in \mathcal{B}_\Sigma$ where ϵ identifies the leaves. ◁

Binary trees are in one-to-one correspondence with unranked trees and sequences of unranked trees (forests). Using the well-known *first-child next-sibling encoding* a binary tree t represents a forest in the following way. The label of the root node of t is the label of the left-most tree of the forest. The content of this tree is given by the left child of t's root. The right child accordingly represents the sequence of siblings. Additionally, we have to mark the end of each sequence of siblings with the special symbol "ϵ" of rank 0. Figure 2.3 shows an unranked tree and Figure 2.4 its binary tree representation[1]. Therefore, the two formats

$$\mathtt{a}(t_1, t_2) \quad \text{and} \quad \mathtt{a}\langle t_1 \rangle\, t_2$$

representing a forest are equivalent but in order to distinguish binary trees from binary tree representations of forest, we will use the latter format whenever talking about forests. Based on this representation of forests as binary trees, we define two kinds of height of a forest.

[1] The three little triangles under each `Body`-element represent a forest throughout this work.

Definition 2.6 (Binary height) Let Σ be an alphabet. The *height* and *binary height* of a forest $f \in \mathcal{F}_\Sigma$, denoted by $ht(f)$ and $bht(f)$ respectively, are defined by:

(i) For ϵ, $ht(\epsilon) = bht(\epsilon) = 0$ and
(ii) for $\mathtt{a} \in \Sigma_2$ and $t_1, t_2 \in \mathcal{B}_\Sigma$, $ht(\mathtt{a}\langle t_1\rangle\, t_2) = \max\{ht(t_1)+1, ht(t_2)\}$ and $bht(\mathtt{a}\langle t_1\rangle\, t_2) = 1 + \max\{bht(t_1), bht(t_2)\}$. ◁

2.2 Tree Substitution

Since each transformation can be put down to tree substitutions, we define it in this section.

For a set $Z_k = \{z_1, \ldots, z_k\}$ of symbols ($k \geq 0$) and a forest $f \in \mathcal{F}_\Sigma(Z_k)$ and forests $f_1, \ldots, f_k \in \mathcal{F}_\Sigma$, we denote by

$$f[f_1/z_1, \ldots, f_k/z_k]$$

the result of substituting forest f_i for every occurrence of symbol z_i in forest f. The same is also defined for substituting nodes. For a set $Y = \{y_1, y_2\}$ of symbols and two trees t_1 and t_2 (as a special case of forests) the tree substitution is depicted in Figure 2.5.

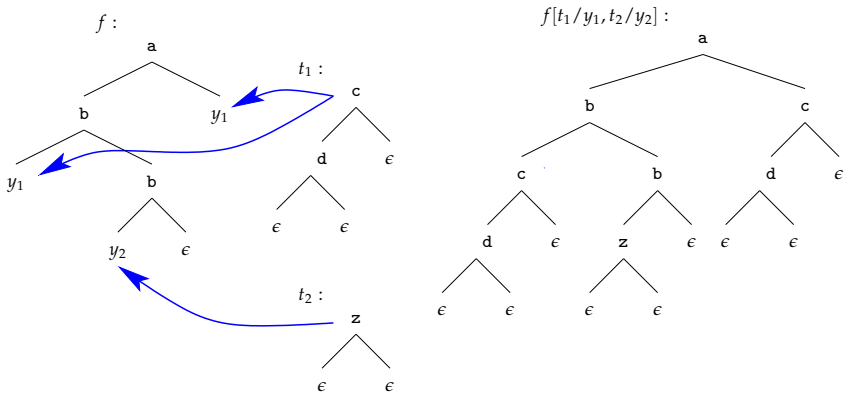

Fig. 2.5. The substitution $f[t_1/y_1, t_2/y_2]$ of trees t_1, t_2 for symbols y_1, y_2.

Let $Y = \{y_1, y_2, y_3, \ldots\}$ be the set of formal parameters and let $Y_k = \{y_1, \ldots, y_k\}$ for $k \geq 0$. For languages $L \in 2^{\mathcal{F}_\Sigma(Y_k)}$ and $L_1, \ldots, L_k \in 2^{\mathcal{F}_\Sigma(Y_k)}$ with $k \geq 0$ and $m \geq 0$, we present here the definition of *inside-out* and *outside-in substitution of* $L_1, \ldots, L_k$ *into* L of [EV85] (see also [ES77; ES78]).

The *inside-out substitution* (IO-substitution), denoted by

$$L[L_1/y_1, \ldots, L_k/y_k]$$

is defined as

$$L[L_1/y_1, \ldots, L_k/y_k] = \{f[f_1/y_1, \ldots, f_k/y_k] \mid t \in L, f_i \in L_i, i = 1, \ldots, k\}.$$

The *ouside-in substitution* (OI-substitution), denoted by

$$L[y_1 \leftarrow L_1, \ldots, y_k \leftarrow L_k]$$

is defined as follows. For a forest $f \in \mathcal{F}_\Sigma(Y_k)$, let $\mathcal{N}_y(f)$ denote the set of all nodes which are labeled with y_j, $j = 1, \ldots, k$.

$$\begin{aligned} L[y_1 \leftarrow L_1, \ldots, y_k \leftarrow L_k] = \\ \{f[f_1/r_1, \ldots, f_n/r_n] \mid f \in L, \mathcal{N}_y(f) = \{r_1, \ldots, r_n\}, \\ \lambda(r_i) = y_j \Rightarrow f_i \in L_j, i = 1, \ldots, n\}. \end{aligned}$$

Here, $\lambda(r_i)$ denotes the label of the node r_i. Note that although all variables are substituted simultaneously, each single occurrence of a variable y_j ($j = 1, \ldots, n$) is replaced by an arbitrary element of the according set L_j.

2.3 Finite-state Tree Automata

Types for XML are usually given as DTDs, XML Schema, or Relax NG. Since its widespread use, we here briefly consider DTDs. A document type definition (DTD) for the document of Figure 2.1 is for example:

```
⟨!ELEMENT Doc     (Chapter*)      ⟩
⟨!ELEMENT Chapter (Title,Body)    ⟩
⟨!ELEMENT Body    (Par+)          ⟩
⟨!ELEMENT Title   #PCDATA         ⟩
⟨!ELEMENT Par     ANY             ⟩
```

Each line defines the *content-model* for an element, i.e., it precisely describes which elements may occur in the forest of children and their order. For element `Chapter`, for example, the DTD defines that the children are a `Title`-element followed by `Body`. A detailed description of the syntax and features of DTDs can be found in the XML recommendation [BPSM$^+$04].

A convenient abstraction of the languages described by schema languages are recognizable tree languages. They are accepted by finite-state tree automata.

Definition 2.7 (Tree Automaton) A *finite-state tree automaton* (fta) is given by a finite set Q of states, a finite ranked alphabet Σ of input symbols, a set $F \subseteq Q$, a set I_σ for each symbol $\sigma \in \Sigma$, and a finite set δ of transitions. Set F

contains all states that can be assigned to the root node of a tree, and I_σ lists all states which can be assigned to a leaf node with label σ. Thus, an fta A is specified by the tuple

$$A = (Q, \Sigma, F, I_\sigma, \delta).$$

δ is a relation between an input symbol together with a sequence of states and a state, i.e., $\delta \subseteq Q \times \Sigma \times Q^k$. A transition is of the form

$$(q, \mathtt{a}, q_1 \cdots q_k)$$

where $q, q_1, \ldots, q_k \in Q$ and $\mathtt{a} \in \Sigma_k$ is a symbol with rank $k \geq 0$.

Tree automata can be viewed as processing their input in a *bottom-up* or in a *top-down* fashion, depending on the roles of the sets F and I_σ.

"Bottom-up" indicates the direction from the leaves to the root of the tree[2]. A bottom-up tree automaton $A = (Q, \Sigma, F, I_\sigma, \delta)$ works as follows. It starts processing by assigning to each leaf node $\mathtt{b} \in \Sigma_0$ a state of $I_\mathtt{b}$. Automaton A moves up all the branches towards the root step by step as follows. If a node v is labeled with the symbol $\mathtt{a} \in \Sigma$ of rank k, then A enters v in state q if there is a transition $(q, \mathtt{a}, q_1 \cdots q_k) \in \delta$, where $q_1, \ldots, q_k$ are the states of A at the direct successors of v. The tree is accepted if A assigns a final state of F to the root.

In the top-down view, an accepting state from set F is assigned to the root. New states are assigned to the children of a node depending on its label and state. A tree is accepted if every leaf labeled with $\mathtt{b}$ is assigned a state from $I_\mathtt{b}$ [Nev02].

For the set of nodes of a tree t, denoted by $\mathcal{N}(t)$, a *run* of automaton A on a tree $t \in \mathcal{T}_\Sigma$ is a mapping

$$run_A : \mathcal{N}(t) \to Q$$

which assigns to each node $v \in \mathcal{N}(t)$ a state $run_A(v) \in Q$ such that the transitions relation is locally respected. For a node $v \in \mathcal{N}(t)$ it is inductively defined as follows: if v is a leaf, then

$$run_A(v) \in \{q \in Q \mid (\epsilon, q) \in \delta\},$$

and otherwise if v is an internal node with label $\mathtt{a} \in \Sigma_k$,

$$run_A(v) \in \{q \in Q \mid \exists\, q_1, \ldots, q_k \in Q : (q, \mathtt{a}, q_1 \ldots q_k) \text{ and } q_i \in run_A(vi),\ i = 1, \ldots, k\}.$$

The tree language accepted by A consists of the trees $t \in \mathcal{T}_\Sigma$ by which A can reach an accepting state

[2] Gécseg and Steinby remark to this that "*this terminology is connected with the common practice of drawing trees upside down*" [GS84].

$$\mathcal{L}(A) = \{t \in \mathcal{T}_\Sigma \mid run_A(t) \in F\}.$$

In other words, the language consists of all trees having runs which map their roots to an accepting state.

If δ is a function, i.e., if there is for each symbol $\mathtt{a} \in \Sigma_k$ of rank k and each k-tupel $q_1, \ldots, q_k$ at most one state q with $(q, \mathtt{a}, q_1 \cdots q_k) \in \delta$, then A is called *deterministic*. Since for the deterministic case δ is a function $\delta : \Sigma \times Q^k \to Q$, we use the following notation $\delta_\mathtt{a}(q_1 \cdots q_k) = q$ to indicate that the automaton changes into state q for symbol $\mathtt{a}$ if $q_1, \ldots, q_k$ are the states at the corresponding subtrees. In theory, deterministic finite tree automata can be exponentially larger than non-deterministic ones. In practice, however, they are usually not much larger than a corresponding non-deterministic one [GS97].

Example 2.8 Let $A = (Q, \Sigma, F, I_\mathtt{e}, \delta)$ such that

- $Q = \{q_{\text{end}}, q_{\text{left}}, q_{\text{right}}, q_{\text{init}}\}$,
- $\Sigma = \{\mathtt{root}, \mathtt{left}, \mathtt{right}, \mathtt{e}\}$ where the symbols `root`, `left`, and `right` have rank 2, and `e` is of rank 0,
- $F = \{q_{\text{end}}\}$,
- $I_\mathtt{e} = \{q_{\text{init}}\}$,

and δ consists of the following transitions:

$$\begin{array}{ll}
(q_{\text{end}}, \mathtt{root}, q_{\text{left}}q_{\text{right}}), & (q_{\text{init}}, \mathtt{e}), \\
(q_{\text{left}}, \mathtt{left}, q_{\text{left}}q_{\text{right}}), & (q_{\text{right}}, \mathtt{right}, q_{\text{left}}q_{\text{right}}), \\
(q_{\text{left}}, \mathtt{left}, q_{\text{left}}q_{\text{init}}), & (q_{\text{right}}, \mathtt{right}, q_{\text{left}}q_{\text{init}}), \\
(q_{\text{left}}, \mathtt{left}, q_{\text{init}}q_{\text{right}}), & (q_{\text{right}}, \mathtt{right}, q_{\text{init}}q_{\text{right}}), \\
(q_{\text{left}}, \mathtt{left}, q_{\text{init}}q_{\text{init}}), & (q_{\text{right}}, \mathtt{right}, q_{\text{init}}q_{\text{init}}).
\end{array}$$

Automaton A accepts all trees with a root node labeled with "`root`"; nodes which are the left son of its father are labeled with "`left`" while right sons are labeled with "`right`" and leafs labeled with "`e`".

2.4 Monadic Second-Order Logic

One of the main tasks in XML processing is *querying*, i.e., locating parts of documents with some specified structural properties. The queries are defined in a *pattern language*, which allows to exactly describe all properties that should be matched by the queried nodes. Common pattern languages are XPath [CD99a] or fxgrep [Ber05]. They both share the idea of using paths as a framework for expressing queries. The following XPath expression

```
//Chapter[Body/Def]/Title
```

asks for all `Title`-elements in a document which are the direct successor of a node labeled with `Chapter`. The additional expression grouped in "`[ ]`"

restricts the `Chapter`-elements to those having a `Def` node inside a `Body`-element.

Monadic second-order logic (MSO) on unranked trees has been identified as a convenient theoretical framework for reasoning about the expressiveness and implementations of practical XML query languages (cf. [NS02]). The above query can be formulated in MSO as follows:

$$\begin{array}{l}\exists z.\, \mathsf{label}_{\texttt{Chapter}}(z) \wedge \\ (\exists x_1.\, z/x_1 \wedge \mathsf{label}_{\texttt{Body}}(x_1) \wedge (\exists x_2.\, x_1/x_2 \wedge \mathsf{label}_{\texttt{Def}}(x_2))) \wedge \\ z/x_1 \wedge \mathsf{label}_{\texttt{Title}}(x)\end{array}$$

Here, the formula in braces corresponds to the additional structural restriction [`Body/Def`]. The first sub-formula $\exists z.\, \mathsf{label}_{\texttt{Chapter}}(z)$ corresponds to the XPath expression `//Chapter`. The formula $\mathsf{label}_{\texttt{Chapter}}(z)$ returns true for all nodes that are labeled with `Chapter`. The last part of the formula, i.e., $z/x_1 \wedge \mathsf{label}_{\texttt{Title}}(x)$ completes the path to the queried nodes. The sub-formula z/x_1 denotes all direct successors x_1 of node z and $\mathsf{label}_{\texttt{Title}}(x)$ restricts these nodes to those that are labeled with `Title`.

Monadic second-order logic is an extension of first-order logic by second-order variables denoted with $X, X_1, X_2 \ldots$ which range over sets of elements of models. Each second-order variable X implies a corresponding atomic formula $X(x), X(y)$ with the intended meaning "x belongs to X", "y belongs to X". An overview of MSO and its connection to automata can be found in [Tho97].

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

$$\phi ::= x_1; x_2 \mid x_1/x_2 \mid \mathsf{label}_{\mathtt{a}}(x_1) \mid X(x_1) \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists\, x_1.\phi \mid \exists\, X.\phi$$

Here, x_1, x_2 are individual variables ranging over nodes of the input forest, while X is a set variable ranging over sets of nodes. The binary relation symbols "$;$" and "$/$" denote the *next sibling* and *child* relation between nodes, respectively. The expression $\mathsf{label}_{\mathtt{a}}(x)$ is true for all nodes labeled with the symbol $\mathtt{a} \in \Sigma$.

For a given forest f, an assignment ρ_1 of the individual variables occurring free in a formula ϕ to nodes in $\mathcal{N}(f)$, and an assignment ρ_2 of set variables to node sets, the satisfiablity assertion $\rho_1, \rho_2 \models_f \phi$ is defined by

$$\begin{array}{ll}
\rho_1,\rho_2 \models_f x_1; x_2 & \text{iff } \exists s \in \Pi(f), i \in \mathbb{N} : \rho_1(x_1) = si \text{ and } \rho_1(x_2) = s(i+1) \\
\rho_1,\rho_2 \models_f x_1/x_2 & \text{iff } \exists i \in \mathbb{N} : \rho_1(x_1)i = \rho_1(x_2) \\
\rho_1,\rho_2 \models_f \mathsf{label}_{\mathtt{a}}(x_1) & \text{iff } f[\rho_1(x_1)] = \mathtt{a}\langle f' \rangle \text{ for } \mathtt{a} \in \Sigma,\ f' \in \mathcal{F}_\Sigma \\
\rho_1,\rho_2 \models_f X(x_1) & \text{iff } \rho_1(x_1) \in \rho_2(X) \\
\rho_1,\rho_2 \models_f \phi_1 \vee \phi_2 & \text{iff } (\rho_1,\rho_2 \models_f \phi_1) \text{ or } (\rho_1,\rho_2 \models_f \phi_2) \\
\rho_1,\rho_2 \models_f \neg\phi & \text{iff not } \rho_1,\rho_2 \models_f \neg\phi \\
\rho_1,\rho_2 \models_f \exists\, x_1.\phi & \text{iff } \rho_1 \oplus \{x_1 \mapsto v\}, \rho_2 \models_f \phi \text{ for some } v \in \mathcal{N}(f) \\
\rho_1,\rho_2 \models_f \exists\, X.\phi & \text{iff } \rho_1,\rho_2 \oplus \{X \mapsto V\} \models_f \phi \text{ for some } V \subseteq \mathcal{N}(f)
\end{array}$$

where "$\oplus$" is the operation which updates or extends an environment with the new bindings to the right[3].

In particular, if an MSO formula ϕ contains the free variables $x_1, \ldots, x_k$, we write

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

to indicate that ϕ holds (or matches) in the forest f if we bind variable x_i to the node $v_i \in \mathcal{N}(f)$ for $i = 1, \ldots, k$.

We close this short introduction to monadic second-order logic with a classical result of Thatcher and Wright [TW68] and Doner [Don70], respectively. It relates recognizablity with MSO-definability:

Theorem 2.9 A set of finite trees is recognizable by a finite tree automaton if and only if it is MSO-definable.

This result implies that for every MSO formula, there exists a nondeterministic finite automaton accepting exactly the same set as described by the formula.

2.5 Notes and References

The connection between XML and unranked trees is described in several publications. We mention here on behalf of the existing introductions the survey of Suciu [Suc98], the books of Goldfarb [Gol90], Abiteboul et al. [ABS00], and the recently published XML introduction of Møller and Schwartzbach [MS06]. Our approach is also inspired by the works of Neumann [Neu99] and Berlea [Ber05].

The term *binary height* in Definition 2.6 was introduced in an article about a finite-state transducer, which directly works on forests [PS04].

A full explanation of inside-out and outside-in substitution can be found in the survey "IO and OI" of Engelfriet and Schmidt [ES77; ES78].

Tree automata were introduced by Doner [Don65; Don70] and Thatcher and Wright [TW65; TW68]. Their goal was to prove the decidability of weak second-order theory with multiple successors. Overviews of tree automata are given in the excellent publications of Gécseg and Steinby [GS84; GS97] and Hopcroft et al. [HMU01]. Additionally, we want to mention the "TATA" textbook presenting basics of tree automata as well as several variants [CDG$^+$97].

A concise overview of monadic second-order logic and its connection to automata can be found in [Tho97].

[3] For two variable environments σ and ω,

$$(\sigma \oplus \omega)(x_i) = \begin{cases} \omega(x_i), & \text{if defined} \\ \sigma(x_i), & \text{otherwise.} \end{cases}$$

3

The Transformation Language TL

Due to XML's increasingly important role in the exchange of data on the Web, it is highly desirable to have algorithms for static type checking XML transformations. This means, before transforming concrete documents, we want to analyse whether each document, which conforms to a specified input type, is transformed into a valid document of the predefined output type.

Static type checking of XML transformations can be considered in a wide variety of settings [MS05]. In general, one first has the choice in the type and second in the transformation formalism. Usually, the input and output types are specified by DTDs [BPSM$^+$04] or XML Schema [FW04]. Now one can also use Relax NG to define a pattern for the structure and content of an XML document [Jam01]. Transformation formalisms can be XQuery [BCF$^+$06], XSLT [Cla99; Kay05], or the pure functional approach fxt [BS02]. 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 like OCaml is used to define the transformation, then type checking is obviously undecidable and can only be approximated. In the context of this work we are interested in *exact* algorithms, and thus we only consider transformation formalisms for which static type checking is decidable.

In order to be as flexible as possible and to abtract from concrete limitations of existing XML transformation languages, we want to have a transformation language, which

(1) subsumes the key features of the usual languages, and
(2) allows exact static type checking.

Developing such a "universal" transformation language together with a type checking algorithm, provides a type checking method for *all* languages captured by our "universal" transformation language.

Before introducing our XML transformation language, we first present the type formalism for specifying the correct inputs and outputs of our programs. A convenient abstraction of the existing XML type formalisms are

regular tree languages [MLM01; Nev02] (see Section 2.3). They essentially capture the expressiveness of DTDs as well as that of XML Schema. XML Schema, for example, goes beyond regular tree languages because it allows to precisely define the types of the data nodes [FW04]. In the context of this thesis, we are only concerned with the structural properties of XML documents. This means, we understand an XML type as description how the XML elements are related in a document.

In order to illustrate fundamental elements of XML transformation languages, we consider the following XSLT program, which creates a table of contents for a given document:

Example 3.1

```
1  ⟨xsl : template  match = "Doc"⟩
2    ⟨Doc⟩
3      ⟨Toc⟩
4        ⟨xsl : for-each  select = "Chapter"⟩
5          ⟨Entry⟩⟨xsl : value-of  select = "Title"/⟩⟨/Entry⟩
6        ⟨/xsl : for-each⟩
7      ⟨/Toc⟩
8      ⟨xsl : element⟩⟨xsl : apply-templates/⟩⟨/xsl : element⟩
9    ⟨/Doc⟩
10 ⟨/xsl : template⟩

11 ⟨xsl : template  match = "∗"⟩
12   ⟨xsl : element⟩⟨xsl : apply-templates/⟩⟨/xsl : element⟩
13 ⟨/xsl : template⟩
```

Each template is applicable to those input nodes that are matched by the XPath expression given as attribute with name "match". In line 1 it is laid down that this template is meant to transform all nodes of the input labeled with "Doc". Line 11 uses the wild-card "∗" to indicate that this template matches all other nodes which are not labeled with "Doc". These expressions are called *match patterns* and are one of the most important elements of transformation languages.

In the first template the pattern "Chapter" in line 4 defines that the code enclosed with the xsl : for-each tags (lines 4 and 6) is executed for each node labeled with "Chapter". For each Chapter-element an Entry-element is written into the output whose content is determined as the text of the Title-element in the currently transformed Chapter-element (line 5). This second sort of patterns like the one in line 4 are called *select patterns*.

For structuring the transformation it might be useful to organize it in *functions*. The for-each part for example could also be realized as a function that is called for each Chapter-element. The concept of functions corresponds to "modes" in XSLT: all templates with the same mode form a function [XSL99].

Moreover, it seems to be useful to borrow the idea of parameters from programming languages. In this way it is possible to transport context information from the currently transformed element to a later transformation step. Due to the fact that these parameters are a restricted variant whose content cannot be transformed but only copied, they are called *accumulating parameters*.

Summing up, our "universal"XML transformation language should provide:

(1) *match patterns* for selecting rules,
(2) *select patterns* determining where the transformation continues,
(3) *named functions*, which allow to devide the whole transformation into components, and
(4) *accumulating parameters* to refer to context information.

Therefore, we present the transformation language TL which is powerful enough to express many real-world XML applications and nevertheless can effectively be type checked. This language is based on the deterministic and parameter-less transformation language DTL suggested in [MN99]. Since it combines the recursion mechanism of XSLT and monadic second-order logic (MSO) as pattern facility, TL subsumes the essential operations of existing domain specific languages for XML processing, i.e., the so-called *tree transformation core*.

This chapter is devided into 4 sections. We will first define the syntax of a TL program (Section 3.1). After exemplifying TL by means of a typical use-case of text processing, we formalize its denotational and operational semantics in Section 3.3. The chapter is closed by discussing other transformation models.

3.1 Definition

Before discussing its functionality, we first present a formal definition of TL programs. Sharing the rule based approach with most of the well established XML transformation languages such as XSLT, TL can be imagined as a collection of rules, each defining how to transform an exactly differentiated set of sub-forests of the input. This set of sub-forests to which a specific rule is applicable is determined by means of a monadic second-order (MSO) *match pattern* which is given in the definition of that rule. The body of a rule is built up of constant XML parts combined with accumulating parameters and function calls. Function calls again use MSO patterns — the so-called *select patterns* — to specify the parts of the input document for which the function should be called. Accumulating parameters allow to "transport" context information to a subsequent transformation step. Accumulating parameters have to be carefully distinguished from parameters known from programming languages because — in contrast to the latter ones — the content of

accumulating parameters cannot be queried or transformed. It is only possible to copy their content into the output or a new parameter.

Definition 3.2 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

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

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 the accumulating parameters, and A is an action. Possible actions are ϵ, or one of accumulating parameters y_j of the left-hand side. Moreover, assuming that $A_1, \ldots, A_m$ are actions, then the following expressions are also actions

$$A_1\, A_2, \quad \langle \mathtt{a} \rangle A_1 \langle /\mathtt{a} \rangle, \quad q'(\psi, A_1, \ldots, A_m)$$

where a is the label of a node in the output, q' is a function name, and ψ is a select pattern, i.e., an MSO formula with two free variables x_1, x_2. ◁

Given a rule $q(\phi, y_1, \ldots, y_k) \longrightarrow A$ of a TL program P, ϕ is called the *match pattern* and defines to which nodes of the input this rule can be applied. The right-hand side A is called *action part* or simply *action*. The accumulating parameters $y_1, \ldots, y_k$ determine the *rank* of a function q, denoted by $rank(q)$: a function without any parameters is of rank 0 and a function with $k > 0$ parameters is of rank k.

Intuitively, the meaning of the actions listed in the definition is as follows: the output for a node x_1 in the input forest can be either the empty forest ϵ, or the content of one of the accumulating parameters y_j listed in the left-hand side of the rule. It can be the concatenation $A_1\, A_2$ of the outputs produced by the actions A_1 and A_2, or a single element labeled a whose content is recursively determined by some action A_1. Finally, it can be 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 processed next by q' are precisely specified by means of some binary MSO pattern ψ, the *select pattern*.

Since TL programs usually are nondeterministic, we combine the possible action parts $A_1, \ldots, A_n$ in one rule and write

$$q(\phi, y_1, \ldots, y_k) \longrightarrow A_1 \mid \ldots \mid A_n.$$

In general, more than one pattern may match at a point of the computation, i.e., for a call $q(\psi, s_1, \ldots, s_k)$ to function q with the actual parameters $s_1, \ldots, s_k$ there may be rules

$$\begin{aligned} q(\phi_1, y_1, \ldots, y_k) &\longrightarrow A_1 \\ &\vdots \\ q(\phi_n, y_1, \ldots, y_k) &\longrightarrow A_n \end{aligned}$$

matching at the nodes specified by a select pattern ψ. In different practical transformation languages, different resolution strategies have been proposed. fxt, for example, simply chooses the first applicable rule [BS02]. XSLT [Cla99] and its revised second version [Kay05], on the other hand, both recommend on choosing the "most specific" rule. A rule is more specific than another if its priority — a decimal number — is higher. If a node in a source document matches more than one rule, only the rule with the highest priority is considered. The priority of a rule can be explicitly specified by the programmer or if it is not specified, a default priority is computed, based on the syntax of the pattern. For example, a pattern that matches nodes according to their name *and* context has a higher priority than a pattern that matches nodes only according to their names. In many cases, the suggested process to determine priorities computes for highly selective patterns a higher priority than for less selective patterns. However, since the priority is determined only based on the syntax of the pattern, a pattern that matches a subset of the nodes matched by another one has not necessarily a higher priority. The patterns `attribute(*,xs:decimal)` and `attribute(*,xs:short)`, for example — matching attributes of type `decimal` or `short` —, have the same priority, despite the fact that the latter pattern matches a subset of the nodes matched by the former. Therefore, the second version of XSLT recommends to allocate explicit priorities [Kay05].

MSO logic as a pattern language is strong enough to make priorities for selecting rules explicit by adding the negation of a conjunction of all patterns of higher priorities. For three rules, for example,

$$\begin{aligned} q(\phi_1, y_1, \ldots, y_k) &\longrightarrow A_1 \\ q(\phi_2, y_1, \ldots, y_k) &\longrightarrow A_2 \\ q(\phi_3, y_1, \ldots, y_k) &\longrightarrow A_3 \end{aligned}$$

where ψ_1 has the highest priority and ψ_3 the lowest, we construct the following patterns:

$$\begin{aligned} q(\phi_1, y_1, \ldots, y_k) &\longrightarrow A_1 \\ q(\phi_2 \wedge \neg\phi_1, y_1, \ldots, y_k) &\longrightarrow A_2 \\ q(\phi_3 \wedge \neg(\phi_1 \vee \phi_2), y_1, \ldots, y_k) &\longrightarrow A_3 \end{aligned}$$

Thus, it is always possible to write TL programs for which the 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 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.

Definition 3.3 (Deterministic TL programs) A TL program $P = (R, A_0)$ is *deterministic* if for all q and all select patterns ψ occuring in a call to q holds: at most one match pattern ϕ of q's rules matches the selected set of nodes. ◁

In order to be more flexible, we consider nondeterministic TL programs because nondeterminism can be used to paraphrase predicates that go beyond MSO's expressive power. This means — with regard to types — that at such a point of the transformation, the type of the result is the type returned by one of the potentially chosen actions. The nondeterministic choice between the different potentially chosen actions gives a type safe over-approximation of the intended transformation.

3.2 Example

The following TL program illustrates the different language features. A very common task in document processing is to make a table of contents. Thinking of a PhD thesis for example, it is arranged in chapters each having a title and a body containing the text of that chapter. An example document can be seen in Figure 3.1.

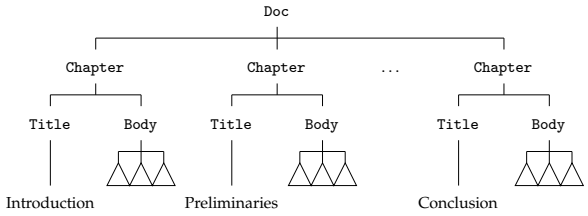

Fig. 3.1. A document tree.

The table of contents should contain all chapter titles in the correct order, i.e., in our case it lists "Introduction", "Preliminaries",..., and finally "Conclusion". Figure 3.2 shows the transformed document which now contains an element `Toc` as new child of the root element. This new element contains an `Entry`-element for each chapter.

In order to create the table of contents automatically, one has to successively transform each `Chapter`-element, identify the string of its title, and copy it as `Entry` into the `Toc`-element. More precisely, we start the transformation at the root node and write a `Doc`-node followed by a `Toc`-element into the output. Then, we have to search for every `Title`-node occurring under the root node. Each of these is transformed by grouping its content under an `Entry`-element, and then copied as child into `Toc`. To complete the transformation, an exact copy of the sequence of `Chapter`-elements has to be added as right siblings of `Toc`. This operation can be expressed with the TL program

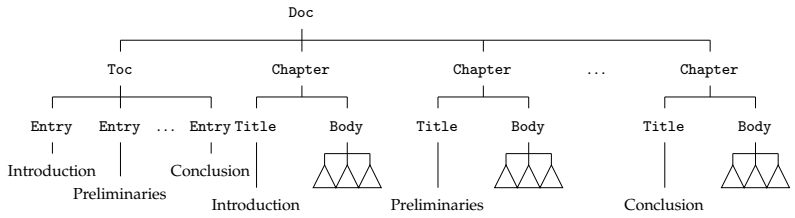

Fig. 3.2. The transformed document.

Example 3.4

```
1  q0(label_Doc(x1))      ⟶ ⟨Doc⟩
2                              ⟨Toc⟩
3                                 q1(∃ z. x1/z ∧ z/x2 ∧ label_Title(x2))
4                              ⟨/Toc⟩
5                              q2(x1/x2)
6                           ⟨/Doc⟩
7  q1(label_Title(x1))    ⟶ ⟨Entry⟩q2(x1/x2)⟨/Entry⟩
8  q2(x1)                 ⟶ x1
```

The left-hand sides of the rules (cf. lines 1, 7, and 8) define at which nodes of the input the respective function is applicable. Line 1 defines function q_0. The match pattern $\mathsf{label}_{\texttt{Doc}}(x_1)$ signalizes that q_0 can be applied to all nodes with label `Doc`. Each right-hand side defines how the matched nodes have to be transformed. They contain XML fragments like $\langle\texttt{Doc}\rangle \ldots \langle\texttt{/Doc}\rangle$, and calls of other TL functions. In line 3 function q_1 is called for all nodes that are matched by the MSO select pattern $\exists\, z.\, x_1/z \wedge z/x_2 \wedge \mathsf{label}_{\texttt{Title}}(x_2)$. It matches all nodes x_2 that are a child of a node z, which itself is a child of the current node x_1. Moreover, this node must be labeled with `Title`. Thus, this select pattern searches for every `Title` elements occurring in the input document. Line 5 completes the transformation by adding an exact copy of the sequence of `Chapter`-elements as children of `Doc`.

The transformation of the `Title`-elements is defined in line 7. Function q_1 groups the content of each `Title` (expressed by $q_2(x_1/x_2)$) between tags labeled with `Entry`.

Last but not least, function q_2 (cf. line 8) returns an exact copy of the current input.

This short TL program obviously preforms the desired transformation and creates an output document for a document arranged in chapters where the sequence of chapters is preceeded by a `Toc`-element containing all chapter titles. Thus, it is equivalent to the XSLT-program presented in Example 3.1.

3.3 Induced Transformation

The evaluation of a TL Program $P = (R, A_0)$ begins at the root node of the input document. Given an input document d, the program P starts processing by evaluating its initial action A_0 for the root node of d. A function q with actual parameters $f_1, \ldots, f_k$ is applied to a node v of the input by carrying out the following steps. First, we nondeterministically choose one of the rules $q(\phi, y_1, \ldots, y_k) \longrightarrow A$ for function q where pattern ϕ matches the input node v. Then we bind the formal parameters $y_j, j = 1, \ldots, k$, to the actual parameters f_j and execute the action part A of the chosen rule. 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 of the input document for which the pair (v, u) matches pattern ψ.

We first characterize the transformation induced by a TL program P by means of a fixpoint semantics. Afterwards, we define it by using a "traditional" operational semantics via interpreting each TL function as a rewrite-system working on forests.

Due to the use of accumulating parameters, function calls can be nested. Accordingly, we have the choice to which function we apply a rule first. Therefore, we have to distinguish between two different evaluation modes: outside-in and inside-out [EV85]. In general, both strategies compute different results because of the order of copying and nondeterminism. Consider the following TL program:

$$\begin{array}{ll} q_0(\phi_0) & \longrightarrow q_1(x_1, q_2(x_1)) \\ q_1(\phi_1, y) & \longrightarrow \langle \mathtt{a} \rangle y\, y \langle / \mathtt{a} \rangle \\ q_2(\phi_1) & \longrightarrow \langle / \mathtt{l} \rangle | \langle / \mathtt{r} \rangle \end{array}$$

For a node v of the input, the program is evaluated with the following transformation steps:

$$q_0(v) \Rightarrow q_1(v, q_2(v))$$

Evaluating the inner-most function call first, results in

$$\begin{array}{ll} q_1(v, \langle / \mathtt{l} \rangle) \Rightarrow \langle \mathtt{a} \rangle \langle / \mathtt{l} \rangle\, \langle / \mathtt{l} \rangle \langle / \mathtt{a} \rangle & \text{or} \\ q_1(v, \langle / \mathtt{r} \rangle) \Rightarrow \langle \mathtt{a} \rangle \langle / \mathtt{r} \rangle\, \langle / \mathtt{r} \rangle \langle / \mathtt{a} \rangle & \end{array}$$

This means inside-out evaluation results in trees where the leafs are identical. For the outside-in mode, we get

$$q_0(v) \Rightarrow q_1(v, q_2(v)) \Rightarrow \langle \mathtt{a} \rangle q_2(v)\, q_2(v) \langle / \mathtt{a} \rangle$$

Then, each occurrence of $q_2(v)$ is nondeterministically evaluated and the resulting tree is one of:

$$\langle \mathtt{a} \rangle \langle / \mathtt{r} \rangle\, \langle / \mathtt{r} \rangle \langle / \mathtt{a} \rangle,\ \langle \mathtt{a} \rangle \langle / \mathtt{r} \rangle\, \langle / \mathtt{l} \rangle \langle / \mathtt{a} \rangle,\ \langle \mathtt{a} \rangle \langle / \mathtt{l} \rangle\, \langle / \mathtt{r} \rangle \langle / \mathtt{a} \rangle,\ \langle \mathtt{a} \rangle \langle / \mathtt{l} \rangle\, \langle / \mathtt{l} \rangle \langle / \mathtt{a} \rangle$$

In the outside-in mode parameters may contain unevaluated function calls which are first copied and then nondeterministically evaluated in a subsequent step. The inside-out mode first chooses nondeterministically how to evaluate the innermost call and copies the evaluation result to every corresponding parameter position [EV85; ES77; ES78].

3.3.1 Denotational Semantics

Let $P = (R, A_0)$ be a TL program. Let Σ denote the set of symbols possibly occurring in input or output forests and let Q denote the set of all TL functions occurring in program P.

Before presenting the semantics, we first define the concatenation of languages $L_1, \ldots, L_k$ as the set of forests assembled from forests of the k languages, i.e.,

$$L_1 \cdots L_k = \{f_1 \cdots f_k \mid f_i \in L_i, i = 1, \ldots, k\}.$$

Outside-In Evaluation

Outside-in evaluation means that the evaluation of parameters is passed to a subsequent transformation step. Accordingly, they may contain unevaluated function calls.

For a given input forest $f \in \mathcal{F}_\Sigma$, each function symbol $q \in Q$ with $rank(q) = k + 1$ is associated with a function from input nodes to sets of forests with parameters in $Y = \{y_1, \ldots, y_k\}$,

$$[\![q]\!]_{f,\mathrm{OI}} : \mathcal{N}(f) \to 2^{\mathcal{F}_\Sigma(Y)}.$$

For all rules $q(\phi, y_1, \ldots, y_k) \longrightarrow A$ of R, the denotation functions are defined as the least fixpoint of

$$[\![q]\!]_{f,\mathrm{OI}}(v) \supseteq [\![A]\!]_{f,\mathrm{OI}} v$$

where v is a node of the input matching pattern ϕ, i.e., $v \models_f \phi$.

Here, $[\![\cdot]\!]_{f,\mathrm{OI}} v$ denotes the evaluation of a right-hand side expression with respect to the given input forest f and the current node v

$$\begin{aligned}
[\![\epsilon]\!]_{f,\mathrm{OI}} v &= \{\epsilon\} \\
[\![y_j]\!]_{f,\mathrm{OI}} v &= \{y_j\} \\
[\![A_1\ A_2]\!]_{f,\mathrm{OI}} v &= \{f_1\, f_2 \mid f_i \in [\![A_i]\!]_{f,\mathrm{OI}} v,\ i = 1, 2\} \\
[\![\langle \mathtt{a} \rangle A_1 \langle / \mathtt{a} \rangle]\!]_{f,\mathrm{OI}} v &= \{\langle \mathtt{a} \rangle f' \langle / \mathtt{a} \rangle \mid f' \in [\![A_1]\!]_{f,\mathrm{OI}} v\} \\
[\![q'(\psi, A_1, \ldots, A_m)]\!]_{f,\mathrm{OI}} v &= L'_1 \cdots L'_l,
\end{aligned}$$

where $L'_i = \tilde{L}_i[y_1 \leftarrow L_1, \ldots, y_m \leftarrow L_m]$ with $\tilde{L}_i = [\![q']\!]_{f,\mathrm{OI}}(v_i)$ and $L_\mu = [\![A_\mu]\!]_{f,\mathrm{OI}} v$ for all $\mu = 1, \ldots, m$, and where $v_1 < \ldots < v_l$ are all nodes $v_i \in \mathcal{N}(f)$ in document order "$<$" which form together with the current node v a match of the binary pattern ψ, i.e., $(v, v_i) \models_f \psi$ for all $i = 1, \ldots, l$.

The simultaneous OI-substitution is defined in Section 2.2.

Inside-Out Evaluation

In the case of inside-out evaluation, the inner-most function call of nested functions in an accumulating parameter is evaluated first. As a consequence, every parameter position contains only fully evaluated results. This evaluation mode corresponds to parameter passing in most of the general purpose programming languages like OCaml or Java.

For a given input forest $f \in \mathcal{F}_\Sigma$ and a set of parameter variables $Y = \{y_1, \ldots, y_k\}$, every function symbol q with k accumulating parameters again denotes a function

$$[\![q]\!]_{f,\text{IO}} : \mathcal{N}(f) \to 2^{\mathcal{F}_\Sigma(Y)}.$$

which are defined as the least fixpoint of

$$[\![q]\!]_{f,\text{IO}}(v) \supseteq ([\![A]\!]_{f,\text{IO}} v)[f[v]/x_1]$$

if $v \models_f \phi$, and $q(\phi, y_1, \ldots, y_k) \longrightarrow A$ is a rule in R. The interpretation of the right-hand side expressions is formalized over their structure, i.e.,

$$\begin{aligned}
[\![\epsilon]\!]_{f,\text{IO}} v &= \{\epsilon\} \\
[\![y_j]\!]_{f,\text{IO}} v &= \{y_j\} \\
[\![A_1\ A_2]\!]_{f,\text{IO}} v &= \{f_1\ f_2 \mid f_i \in [\![A_i]\!]_{f,\text{IO}} v,\ i = 1,2\} \\
[\![\langle \mathtt{a} \rangle A_1 \langle / \mathtt{a} \rangle]\!]_{f,\text{IO}} v &= \{\langle \mathtt{a} \rangle f' \langle / \mathtt{a} \rangle \mid f' \in [\![A_1]\!]_{f,\text{OI}} v\} \\
[\![q'(\psi, A_1, \ldots, A_m)]\!]_{f,\text{IO}} v &= L'_1 \cdots L'_l
\end{aligned}$$

where $L'_1 \cdots L_l$ again denotes the concatenation of all possible forests

$$L'_i = \tilde{L}_i[L_1/y_1, \ldots, L_m/y_m]$$

with $\tilde{L}_i = [\![q']\!]_{f,\text{IO}}(v_i)$ and $L_\nu = [\![A_m]\!]_{f,\text{IO}} v$ for all $\nu = 1, \ldots, m$. As for the outside-in mode, $v_1 < \ldots < v_l$ are all nodes $v_i \in \mathcal{N}(f)$ in document order "$<$" which form together with the current node v a match of the binary pattern ψ, i.e., $(v, v_i) \models_f \psi$ for all $i = 1, \ldots, l$.

The IO-substitution is explained in Section 2.2.

Definition 3.5 Let $\mu \in OI, IO$. The μ-transformation realized by a TL program $P = (R, A_0)$ on non-empty forests, denoted by $\tau_{P,\mu}$, is defined by

$$\tau_{P,\mu} = \{(f, s) \in \mathcal{F}_\Sigma \times \mathcal{F}_\Sigma \mid s \in [\![A_0]\!]_{f,\mu}\ root(f)\}$$

where $root(f)$ is the root node of the left-most tree of the input forest f.

3.3.2 Operational Semantics

After characterizing TL programs with the help of fixpoint semantics, we now associate with each TL program a translation by means of a derivation

relation. Therefore, each function is interpreted as a rewriting system working on forests.

Let $P = (R, A_0)$ be a TL program. Let Q denote the set of all function symbols occurring in the rules of P and let Σ be the set of all input and output symbols.

For a set Q of TL functions, an alphabet Σ, and an input forest $f \in \mathcal{F}_\Sigma$, the set $\mathcal{DT}(Q, \Sigma, f)$ of terms occurring in a derivation is defined inductively as follows:

(1) $\epsilon \in \mathcal{DT}(Q, \Sigma, f)$;
(2) for the maximal number r of parameters of all functions in Q, term y_j with $1 \leq j \leq r$ is in $\mathcal{DT}(Q, \Sigma, f)$;
(3) for $d \in \mathcal{DT}(Q, \Sigma, f)$ and $\mathtt{a} \in \Sigma$, term $\langle \mathtt{a} \rangle d \langle / \mathtt{a} \rangle \in \mathcal{DT}(Q, \Sigma, f)$;
(4) if d_1 and d_2 are terms in $\mathcal{DT}(Q, \Sigma, f)$, then $d_1\, d_2 \in \mathcal{DT}(Q, \Sigma, f)$;
(5) for a function $q \in Q$ with k parameters, a node $v \in \mathcal{N}(f)$ and terms $d_1, \ldots, d_k \in \mathcal{DT}(Q, \Sigma, f)$, then $q(v, d_1, \ldots, d_k)$ is a term in $\mathcal{DT}(Q, \Sigma, f)$.

Particularly, every result forest is contained in the set of derivation terms, i.e., $\mathcal{F}_\Sigma \subseteq \mathcal{DT}(Q, \Sigma, f)$.

Before defining the derivation relations, we first specify how the action part of a TL rule is evaluated with respect to an input forest $f \in \mathcal{F}_\Sigma$ and a node $v \in \mathcal{N}(f)$.

$$
\begin{aligned}
[\epsilon]_{f,v} &= \epsilon \\
[\langle \mathtt{a} \rangle A \langle / \mathtt{a} \rangle]_{f,v} &= \langle \mathtt{a} \rangle [A]_{f,v} \langle / \mathtt{a} \rangle \\
[A_1\, A_2]_{f,v} &= [A_1]_{f,v}\, [A_2]_{f,v} \\
[y_j]_{f,v} &= y_j \\
[q(\psi, A_1, \ldots, A_k)]_{f,v} &= q(u_1, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \ldots q(u_r, [A_1]_{f,v}, \ldots, [A_k]_{f,v}),
\end{aligned}
$$

where $\{u_1 < \ldots < u_r\} = \{u \in \mathcal{N}(f) \mid (u, v) \models_f \psi\}$. The order $<$ is the natural document order of f. In particular, the evaluation of the initial action A_0 of program P is contained in the set of derivation terms $\mathcal{DT}(Q, \Sigma, f)$.

In order to define the derivation relation, let $\alpha, \beta \in \mathcal{DT}(Q, \Sigma, f)$. The binary relations $\Rightarrow_{P,f}, \Rightarrow_{P,f,\mathrm{IO}}, \Rightarrow_{P,f,\mathrm{OI}} \subseteq \mathcal{DT}(Q, \Sigma, f) \times \mathcal{DT}(Q, \Sigma, f)$ are given by

(i) $\alpha \Rightarrow_{P,f} \beta$ if there are forests $s_1, \ldots, s_k \in \mathcal{DT}(Q, \Sigma, f)$, there is an occurrence of an unevaluated function call $q(v, s_1, \ldots, s_k)$ at node $u \in \mathcal{N}(\alpha)$ of α, and there is a rule $q(\phi, y_1, \ldots, y_k) \longrightarrow A$ in R with $v \models_f \phi$, such that

$$\beta = \alpha[[A]_{f,v}/u][s_1/y_1, \ldots, s_k/y_k],$$

i.e., the unvaluated call of q at node u is replaced by its result $[A]_{f,v}$.

(ii) $\alpha \Rightarrow_{P,f,\mathrm{OI}} \beta$ if $\alpha \Rightarrow_{P,f} \beta$, but for u in (i) the following holds: on the path from u to the root of α no function symbol occurs. This restriction guarantees that the outermost function call is evaluated first.

(iii) $\alpha \Rightarrow_{P,f,\mathrm{IO}} \beta$ if $\alpha \Rightarrow_{P,f} \beta$, but the parameters $s_1, \ldots, s_k$ are restricted to be elements of $\mathcal{F}_\Sigma$, i.e., parameters are fully evaluated when they are passed to a subsequent transformation step.

For both binary relations $\Rightarrow_{\mathrm{IO}}$ and $\Rightarrow_{\mathrm{OI}}$, $\overset{*}{\Rightarrow}$ and $\overset{+}{\Rightarrow}$ are defined as usual.

Definition 3.6 Let $\mu \in \{OI, IO\}$. The μ-translation induced by TL program $P = (R, A_0)$ on non-empty forests, denoted by $\pi_{P,\mu}$, is defined as

$$\pi_{P,\mu} = \{(f, s) \in \mathcal{F}_\Sigma \times \mathcal{F}_\Sigma \mid [A_0]_{f,root(f)} \Rightarrow_{P,f,\mu} s\}$$

where $root(f)$ denotes the root node of the left-most tree of the input forest f and $[A_0]_{f,root(f)}$ denotes the evaluation of the initial action. ◁

The following theorem states that the fixpoint characterization and the operational semantics of TL programs coincide. As a consequence, this result justifies the use of the fixpoint semantics in the following discussions.

Theorem 3.7 Let $P = (R, A_0)$ a TL program and let $\mu \in \{IO, OI\}$. Then $\pi_{P,\mu} = \tau_{P,\mu}$.

3.4 Notes and References

Our XML transformation language TL has been previously presented in [MBPS05]. It is based on the transformation language DTL [MN99]. Based on the recursion mechanism of XSLT [XSL99] and using monadic second-order logic as pattern language, DTL subsumes the essential operations of domain specific languages for XML processing. All results concerning DTL, however, are restricted to to the fragment of deterministic and top-down transformations only. This fragment is a natural generalization to *unranked* trees of top-down tree transducers with look-ahead. Moreover, it is shown that the emptiness and finiteness problems are decidable for ranges of DTL programs. Here, we lift the restriction and consider general DTL programs (deterministic and nondeterministic). Moreover, we enhance the original model by accumulating parameters. They allow for long-distance tranportation of document parts and are therefore a convenient technique to provide access to context information.

There are a number of well-established XML transformation languages with varying support for type checking.

Unquestionably, the most popular transformation languages are XSLT1.0 [Cla99] and its follow-up version XSLT2.0 [Kay05]. They are stand-alone domain-specific language, i.e., they have their own compiler or interpreter. Like TL, it is rule-based and these rules are similar to those of TL. Match and select patterns are defined in XPath1.0 [CD99b] and XPath2.0 [BBC$^+$06], respectively, which are both languages for addressing nodes or parts of an

XML document. The first version itself is untyped and does not support type checking, but recently, a tool for flow-based type checking of XSLT has been designed [MOS05]. XSLT2.0 is prepared as a schema-aware transformation processor but it still has only the status of a Candidate Recommendation.

XQuery [BCF+06] is proposed by the W3C. It is a strongly-typed functional language for querying and transforming XML documents. While XSLT is more intuitive as a stylesheet language, XQuery is especially expressive for typical database operations like joins and sorting in so-called *data-oriented applications* [Ber05]. As XML trees may be seen to generalize relational database tables, XQuery is designed to generalize the SQL query language [MS05]. A term language is used for constructing values, and XPath [CD99b] is used for deconstructing and pattern matching.

XDuce [HP03] and CDuce [BCF03] are a family of functional transformation languages. The intention has been to investigate the type-safe integration of XML into a functional programming language. Navigation and deconstruction are based on an extension of the pattern matching mechanism of functional languages with regular expressions. Patterns may contain variables which are bound to parts of the matched structure. The bound parts then can be addressed in subsequent processing. CDuce provides some language features borrowed from general purpose functional languages, like overloaded functions, iterators on sequences and trees, and a very comfortable pattern algebra.

Another functional approach to XML processing is fxt [BS02]. Although it does not support type checking, it is the force behind some features of TL, especially the binary select patterns. Moreover, the lack of schema-awareness in fxt was a decisive motivation to think about an XML transformation model subsuming the tree transformation core of most of the existing transformation languages.

The same reason induced also Milo, Suciu, and Vianu to develop an abstract and rather general model of XML transformation languages called *k*-pebble tree transducer [MSV00; MSV03]. The *k*-pebble tree transducer (*k*-ptt) expresses transformations on ordered, labeled trees. Inspired by tree-walking pebble automata [EH99], up to *k* pebbles can be placed on the input tree, moved up and down the tree, and removed. Engelfriet illustrates the basic idea of pebbles with the following statement: "*The input is not fed into the [transducer] (like money into a coffee machine), but the [transducer] walks on the input [tree] (like a mouse in a maze)*" [EH99]. Transitions of the *k*-ptt are determined by the symbol of the current node, the state, and the placement of the pebbles. Once started by placing the first pebble on the root node of the input, the *k*-ptt is able to move in every (possible) direction in the tree or to stay at the current node. If the transducer emits some output it does not move on the input. A tree-walk automaton [EHV99] is a 1-pebble tree transducer without output transitions and with an accepting state. As observed in [MSV00], there is also a natural connection of pebble tree transducers and the concept of tree-walking automata introduced by Aho and Ullmann [AU71] (see

also [ERS80]). Pebble transducers can be obtained from these automata by adding pebbles and generalizing them from strings to trees [EM03a]. Top-down tree transducers [GS97; GS84] correspond also to the case when $k = 1$ and the pebble is allowed to move only downwards.

4

Stay Macro Tree Transducers

Macro tree transducers were introduced by Engelfriet in 1980 [Eng80]. They combine features of top-down tree transducers [Rou70b] and macro grammars [Fis68a; Fis68b]. Top-down tree transducers map an input tree to an output tree by using finite state rules. They work strictly top-down, which means they start working on the root node of the input tree, then on the second level and so forth until the leaf nodes have been processed. Top-down tree transducers were invented as a formal model of syntax-directed translations. Macro grammars generalize the rewriting rules of context-free grammars by enhancing them with the idea of macros which were originally developed for programming languages. A concise and nevertheless thorough introduction and investigation of macro tree transducers can be found in [EV85].

In principle, macro tree transdcucers consist of rules, each of them meaning to transform a node of the input tree depending on the node's label. Subsequent functions are called for the children of the transformed node. This means that every macro tree transducer can be seen as a restricted TL program where match patterns only test for the label and select patterns choose only nodes that are children of the current node.

On the other hand, a macro tree transducer can be considered as a first-order functional program performing a top-down traversal over its input. Therefore it has named functions with one input argument and a possibly empty list of additional arguments, often called accumulating or context parameters. With these parameters it is possible to accumulate intermediate results and thus they implicitly give access to the context of the tree currently processed in the first argument. In contrast to ordinary parameters known from programming languages, it is not possible to query or even transform a once accumulated temporary result in an accumulating parameter. Compared to the top-down tree transducer, the ability of carrying context information to further processing steps makes the macro tree transducer the more expressive model.

4.1 Definition

Here, we consider a slight generalization of macro tree transducers. Our model is not obliged to proceed to subterms of its current input in every step. This variant is called *stay macro tree transducer* (stay-mtt) because it has the ability to change its state while "staying" at a node. It was invented by Engelfriet and Maneth in [EM03a] in order to have a convenient model capturing all features of k-pebble tree transducers [MSV00; MSV03].

Definition 4.1 (Stay macro tree transducer) A *stay macro tree transducer* (stay-mtt) M is defined by

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

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

$$\begin{array}{ll} q(x_0, & y_1, \ldots, y_k) \to t \quad \text{or} \\ q(\mathtt{a}(x_1, \ldots, x_n), & y_1, \ldots, y_k) \to t \end{array}$$

where $q \in Q$ is a function symbol of rank $k+1$, $\mathtt{a}$ is an input symbol from Σ, x_0 as well as $x_1, \ldots, x_n$ are input variables, $y_1, \ldots, y_k$ ($k \geq 0$) are the accumulating parameters of q, and t is an expression describing the output actions of the corresponding rule. Possible actions are recursively composed by the grammar

$$t ::= \mathtt{b}(t_1, \ldots, t_r) \mid y_j \mid q'(x_i, t_1, \ldots, t_s)$$

where $r, s \geq 0$, $\mathtt{b} \in \Sigma$ is a label of an output node, $y_j, j = 1, \ldots, k$, is one of the accumulating parameters, $q' \in Q$ is of rank $s+1$, and x_i is one of the input variables of the left-hand side. ◁

The first argument of a function q is called *input*, and the others are called *accumulating parameters* or shorter *parameters*. Correspondingly, we will call variables x_i *input variables* and variables y_j *accumulating variables*.

Every function $q \in Q$ of a stay-mtt is at least of rank 1. For a function q with k parameters its rank is denoted by $\mathit{rank}(q) = k+1$. We here restrict initial function symbols to not having accumulating parameters, i.e., we fix for every $q \in Q_0$ that $\mathit{rank}(q) = 1$. Accordingly, right-hand sides of rules for initial functions do not contain parameters y_j.

Rules like $q(x_0, y_1, \ldots, y_k) \to t$ are called *stay-rules* and rules of the form $q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t$ are called $q, \mathtt{a}$-*rules*. In case of several rules for the same function symbol q and the same symbol $\mathtt{a}$ we also write

$$q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t_1 \mid \ldots \mid t_r$$

or in case of a stay-rule

$$q(x_0, y_1, \ldots, y_k) \to t_1 \mid \ldots \mid t_s$$

to enumerate all occurring right-hand sides. The set of right-hand sides of a $q, \mathtt{a}$-rule is denoted by $rhs(q, \mathtt{a}) = \{t_1, \ldots, t_r\}$ and accordingly, the set of right-hand sides of a stay-rule of function q is defined as $rhs(q) = \{t_1, \ldots, t_s\}$.

Intuitively, the meaning of the action expressions is as follows. The output can either be a node labeled with a symbol $\mathtt{b} \in \Sigma$, whose content is recursively determined by expressions $t_1, \ldots, t_r$. The output can be the current content of one of the accumulating parameters y_j of the left-hand side. It can be a recursive call to some function q' on the i-th subtree of the current input node or on the current input node itself, if it is a stay-rule, where the parameters of q' are again determined by some expressions $t_1, \ldots, t_s$.

In general, the input and output symbols are defined seperately by two different alphabets Σ and Δ [EV85; FV98]. Although we do not distinguish between input and output alphabet, it is obviously possible to define transductions where the symbols in the input and output differ. The set of input symbols for a stay-mtt M consists of all symbols occurring in a left-hand side of a rule. Thus, the input alphabet is implicitly given by the rules of M:

$$\mathfrak{A}_{\text{in}} := \{\mathtt{a} \in \Sigma \mid \exists q \in Q : rhs(q, \mathtt{a}) \neq \emptyset\}$$

Analogously we define the set $\mathfrak{A}_{\text{out}}$ of output symbols as the set of all symbols $\mathtt{b} \in \Sigma$ occurring in a right-hand side of a rule of M:

$$\mathfrak{A}_{\text{out}} := \{\mathtt{b} \in \Sigma \mid \exists\, q \in Q : \mathtt{b} \in r, \text{ such that } r \in rhs(q, \mathtt{a}) \cup rhs(q),\ \mathtt{a} \in \mathfrak{A}_{\text{in}}\}$$

Definition 4.2 (i) A stay macro tree transducer M is *deterministic*, if for each function $q \in Q$ either the only rule for q is a stay-rule $q(x_0, y_1, \ldots, y_k) \to t$ or for each input symbol $\mathtt{a} \in \Sigma$ there is at most one rule in R with left-hand side $q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k)$ and in R is no stay-rule for q.

(ii) A stay macro tree transducer is *linear*, if in the right-hand side of every rule in R, each input variable x_i of the left-hand side occurs at most once [EV85].

(iii) A stay macro tree transducer without stay-rules $q(x_0, y_1, \ldots, y_k) \to t$ is called *macro tree transducer* (mtt). ◁

The class of all transductions, which can be realized by a stay macro tree transducer is denoted by s-MTT.

4.2 Induced Tree Transformation

A stay-mtt can transform input trees into output trees and hence it realizes or induces a tree transformation, sometimes also called tree transduction, since the model itself is named transducer. The evaluation of a stay macro tree transducer $M = (Q, \Sigma, Q_0, R)$ begins at the root node of the input. Given an input tree s_0, the stay-mtt M starts to process by evaluating one of its initial functions $q_0 \in Q_0$ for the root node of s_0. A function $q \in Q$ with actual

accumulating parameters $t_1, \ldots, t_k$ is applied to a subtree $s = \mathtt{a}(s_1, \ldots, s_n)$ by carrying out the following steps. First, it is nondeterministically chosen one of the rules

$$\begin{array}{ll} q(x_0, & y_1, \ldots, y_k) \to t \quad \text{or} \\ q(\mathtt{a}(x_1, \ldots, x_n), & y_1, \ldots, y_k) \to t' \end{array}$$

for function q. If we have chosen a stay-rule, s is substituted for the input variable x_0 and the actual parameter t_j for the accumulating variable y_j in t. Otherwise, we substitute the subterms s_i and t_j for the variables x_i and y_j in t'.

Since function calls may be nested, the order in which nested calls are evaluated influences the value of the final output. Consider the following stay-mtt $M = (Q, \Sigma, Q_0, R)$ with the three states $Q = \{q_0, q_1, q_2\}$, the alphabet $\Sigma = \{\mathtt{a}, \mathtt{e}, \mathtt{l}, \mathtt{r}\}$ where $\mathtt{a}$ is of rank 2 and the other symbols are of rank 0. M has one initial state q_0 and the rule set R consists of

$$\begin{array}{ll} q_0(\mathtt{a}(x_1, x_2)) & \to q_1(x_1, q_2(x_1)) \\ q_1(\mathtt{a}(x_1, x_2), y) & \to \mathtt{a}(y, y) \\ q_2(\mathtt{a}(x_1, x_2)) & \to \mathtt{l} \mid \mathtt{r} \\ q_2(\mathtt{e}) & \to \mathtt{e} \end{array}$$

The first rule calls function q_1 for the first input variable with a call of function q_2 on the second input variable. Let s be an input tree of the form

$$\mathtt{a}(\mathtt{a}(\mathtt{e}, \mathtt{e}), \mathtt{e}).$$

If we evaluate the outermost call first, we get

$$\begin{array}{ll} & q_0(\mathtt{a}(\mathtt{a}(\mathtt{e}, \mathtt{e}), \mathtt{e}) \\ \Rightarrow & q_1(\mathtt{a}(\mathtt{e}, \mathtt{e}), q_2(\mathtt{a}(\mathtt{e}, \mathtt{e})) \\ \Rightarrow & \mathtt{a}(q_2(\mathtt{a}(\mathtt{e}, \mathtt{e})), q_2(\mathtt{a}(\mathtt{e}, \mathtt{e}))) \end{array}$$

Then we can nondeterministically choose how to evaluate function q_2 and we get the following possible results for our transformation:

$$\mathtt{a}(\mathtt{l}, \mathtt{l}), \mathtt{a}(\mathtt{l}, \mathtt{r}), \mathtt{a}(\mathtt{r}, \mathtt{l}), \text{ or } \mathtt{a}(\mathtt{r}, \mathtt{r}).$$

If we otherwise evaluate the innermost call to function q_2 first, we nondeterministically choose once how to evaluate q_2. As a consequence, we only get those result trees where the left and right leaf are the same, i.e.,

$$\mathtt{a}(\mathtt{l}, \mathtt{l}), \text{ or } \mathtt{a}(\mathtt{r}, \mathtt{r}).$$

The difference between these two evaluation methods for nested function calls is caused by the order of copying and nondeterminism: whereas in the latter case nondeterminism is followed by copying, in the former case $q_2(\mathtt{a}(\mathtt{e}, \mathtt{e}))$ is copied first and then evaluated nondeterministically [EV85].

These two different evaluation orders are known as outside-in or inside-out [ES77; ES78]. In *outside-in* order (OI), outermost calls are evaluated first. The parameters of a function call may themselves contain function calls which are thus transferred to the body of the function in an unevaluated form. This evaluation order corresponds to the *call-by-name* passing of function arguments in programming languages. Engelfriet and Vogler state already in [EV85], that it does not have any influence on the translation of a stay macro tree transducer if it is evaluated in outside-in mode or if the order, in which nested functins calls are evaluated, is completely unrestricted. The proof idea in [EV85] is as follows: each language derivable from an initial function application can be obtained as the tree language generated by a particular context-free tree grammar. The construction encodes a transducer function together with its input into the nonterminals of the grammar. The grammar rules are obtained from the rule set of the transducer by propagating the according input information to all occurring function calls in the right-hand sides and omitting the input variables x_i. Then they conclude from the fact that the languages generated from a context-free tree grammar by means of outside-in or unrestricted derivation are the same [Fis68a], that the outside-in and the unrestricted evaluation mode are equivalent for macro tree transducers.

In the *inside-out* (IO) order of function evaluation the innermost function call is evaluated first. This means that only fully evaluated output trees are passed in accumulating parameters when a function call is evaluated. This evaluation strategy corresponds to *call-by-value* parameter passing as provided by mainstream imperative programming languages like C [KR88] or functional languages such as ML [MTRD97] or OCaml [LDJ+04].

4.2.1 Denotational Semantics

In this section we define the transformation induced by a stay macro tree transducer from a denotational point of view. We first consider the inside-out evaluation strategy for nested function calls and then define the transduction for outside-in evaluation. The following concerns an arbitrary but fixed stay-mtt $M = (Q, \Sigma, Q_0, R)$.

Inside-Out Evaluation

The meaning of a function symbol $q \in Q$ with k accumulating parameters is a function from input trees to sets of trees with parameters in $Y = \{y_1, \ldots, y_k\}$

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

These functions are defined as the least functions satisfying the following inclusions

- for stay-rules $q(x_0, y_1, \ldots, y_k) \to t_1$

$$[\![q]\!]_{\text{IO}}(s) \supseteq [\![t_1[s/x_0]]\!]_{\text{IO}}$$

  with $q \in Q_{k+1}$ and input tree $s \in \mathcal{T}_\Sigma$, and
- for $q, \mathtt{a}$-rules $q(\mathtt{a}(x_1, \ldots, x_l), y_1, \ldots, y_k) \to t_2$

$$[\![q]\!]_{\text{IO}}(s) \supseteq [\![t_2[s_1/x_1, \ldots, s_l/x_l]]\!]_{\text{IO}}$$

  with $q \in Q_{k+1}$ and input $s = \mathtt{a}(s_1, \ldots, s_l)$ for $\mathtt{a} \in \Sigma$ and $s_1, \ldots, s_l \in \mathcal{T}_\Sigma$,

where the interpretation of a right-hand side expression is given by

$$\begin{aligned}
[\![y_j]\!]_{\text{IO}} &= \{y_j\} \\
[\![\mathtt{b}(t_1, \ldots, t_m)]\!]_{\text{IO}} &= \{\mathtt{b}(t'_1, \ldots, t'_m) \mid t'_i \in [\![t_i]\!]_{\text{IO}}, i = 1, \ldots, m\} \\
[\![q'(s', t_1, \ldots, t_n)]\!]_{\text{IO}} &= \{t'[t'_1/y_1, \ldots, t'_n/y_n] \mid\ t' \in [\![q']\!]_{\text{IO}}(s'), \\
&\qquad\qquad t'_i \in [\![t_i]\!]_{\text{IO}}, i = 1, \ldots, n\},
\end{aligned}$$

where $t'[t'_1/y_1, \ldots, t'_n/y_n]$ denotes the simultaneous IO substitution of all occurrences of variable y_i by the tree t'_i for all $i = 1, \ldots, n$.

An accumulating variable is mapped to itself. The output of a tree with the constant label $\mathtt{b}$ of rank m and children given by the expressions $t_1, \ldots, t_m$, i.e., a right-hand side expression of the form $\mathtt{b}(t_1, \ldots, t_m)$, is mapped to the set of all potential trees labeled with $\mathtt{b}$ whose children are given by evaluating the trees $t_1, \ldots, t_m$. A function call $q'(s', t_1, \ldots, t_n)$ is replaced by the set of all potential evaluations t' of the function symbol q'. The inside-out evaluation strategy finds expression in the fact that every occurrence of a parameter y_i, $i = 1, \ldots, n$, in the tree t' is replaced by the same tree $t'_i \in [\![t_i]\!]_{\text{IO}}$.

Outside-In Evaluation

Again, each function symbol $q \in Q$ with $\mathit{rank}(q) = k + 1$ is associated with a function from input trees to sets of trees with parameters in $Y = \{y_1, \ldots, y_k\}$,

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

These functions are defined as the least functions satisfying the following inclusions

- stay-rules $q(x_0, y_1, \ldots, y_k) \to t_1$ in R have to satisfy

$$[\![q]\!]_{\text{OI}}(s) \supseteq [\![t_1]\!]_{\text{OI}}\sigma$$

  where s is the current input and $\sigma(x_0) = s$ and
- $(q, \mathtt{a})$-rules $q(\mathtt{a}(x_1, \ldots, x_l), y_1, \ldots, y_k) \to t_2$ in R with $a \in \Sigma$ have to satisfy

$$[\![q]\!]_{\text{OI}}(s) \supseteq [\![t_2]\!]_{\text{OI}}\sigma$$

  where $s = \mathtt{a}(s_1, \ldots, s_l)$ with $s_1, \ldots, s_l \in \mathcal{T}_\Sigma$ is the current input and $\sigma(x_i) = s_i, i = 1, \ldots, l$.

Here, $[\![\cdot]\!]_{\mathrm{OI}}\sigma$ denotes the evaluation of a right-hand side expression with respect to the binding σ of the input variables x_i, namely

$$\begin{aligned}[\![y_j]\!]_{\mathrm{OI}}\sigma &= \{y_i\}\\ [\![\mathtt{b}(t_1,\ldots,t_m)]\!]_{\mathrm{OI}}\sigma &= \{\mathtt{b}(t'_1,\ldots,t'_m) \mid t'_i \in [\![t_i]\!]_{\mathrm{OI}}\sigma, i = 1,\ldots,m\}\\ [\![q'(x_i,t_1,\ldots,t_n)]\!]_{\mathrm{OI}}\sigma &= L[y_1 \leftarrow L_1,\ldots,y_n \leftarrow L_n],\end{aligned}$$

where $L = [\![q']\!]_{\mathrm{OI}}(\sigma(x_i))$ and $L_j = [\![t_j]\!]_{\mathrm{OI}}\sigma$ for each $i = 1,\ldots,n$, and $L[y_1 \leftarrow L_1,\ldots,y_n \leftarrow L_n]$ denotes the simultaneous OI substitution (cf. Section 2.2).

An accumulating variable is mapped to itself. The expression $\mathtt{b}(t_1,\ldots,t_m)$ is mapped to the set of all potential trees labeled with $\mathtt{b}$ whose children are given by evaluating the trees $t_1,\ldots,t_m$. A function call $q'(x_i,t_1,\ldots,t_n)$ is first evaluated according to the current input and in the results simultaneously all occurrences of accumulating variables y_j are replaced. Here, the outside-in evaluation strategy becomes apparent in the definition of the simultaneous parameter substitution.

Definition 4.3 Let $\mu \in \{\mathrm{IO},\mathrm{OI}\}$ and let $M = (Q,\Sigma,Q_0,R)$ be a stay-mtt. The μ-transformation realized by M on a non-empty input tree s, denoted by $\tau_{M,\mu}$, is for both evaluation modes a mapping from trees to sets of trees $\tau_{M,\mu} : \mathcal{T}_\Sigma \to 2^{\mathcal{T}_\Sigma}$. These mappings are induced by the initial functions from Q_0 and are defined as

$$\tau_{M,\mu}(s) = \bigcup_{q\in Q_0} [\![q]\!]_\mu(s).\triangleleft$$

For a given set $S \subseteq \mathcal{T}_\Sigma$ we denote by $\tau_{M,\mu}(S)$ the set of all outputs which are produced by M on input trees in S, i.e.,

$$\tau_{M,\mu}(S) = \bigcup_{s\in S} \tau_{M,\mu}(s),$$

where μ again denotes the evaluation strategies IO or OI, respectively.

4.2.2 Operational Semantics

We now associate a tree translation with a stay macro tree transducer by means of a derivation relation. By substituting input trees and trees possibly containing function symbols for the input variables and the accumulating parameters, respectively, a stay-mtt rule describes the rewriting of subtrees. With these rewriting rules it is possible to define the derivation relation.

Let $M = (Q,\Sigma,Q_0,R)$ be a stay macro tree transducer and $\phi,\psi \in \mathcal{T}_{Q\cup\Sigma}$. The set $\mathcal{DT}(Q,\Sigma)$ of terms occurring in a derivation is defined inductively as follows:

(1) for the maximal number r of parameters of all functions in Q, term y_j with $1 \le j \le r$ is in $\mathcal{DT}(Q,\Sigma)$;

(2) for $d_1, \ldots, d_k \in \mathcal{DT}(Q, \Sigma)$ and $\mathtt{a} \in \Sigma_k$, the term $\mathtt{a}(d_1, \ldots, d_k) \in \mathcal{DT}(Q, \Sigma)$;
(3) for a function $q \in Q$ with k parameters, an input tree $s \in \mathcal{T}_\Sigma$ and terms $d_1, \ldots, d_k \in \mathcal{DT}(Q, \Sigma)$, then $q(s, d_1, \ldots, d_k)$ is a term in $\mathcal{DT}(Q, \Sigma)$.

Particularly, every result tree is contained in the set of derivation terms, i.e., $\mathcal{T}_\Sigma \subseteq \mathcal{DT}(Q, \Sigma)$.

In order to define the derivation relation, let $\alpha, \beta \in \mathcal{DT}(Q, \Sigma)$. The binary relations $\Rightarrow_M, \Rightarrow_{M,\mathrm{OI}}, \Rightarrow_{M,\mathrm{IO}} \subseteq \mathcal{DT}(Q, \Sigma) \times \mathcal{DT}(Q, \Sigma)$ are given by

(i) $\alpha \Rightarrow_M \beta$ if there are trees $s_1, \ldots, s_k \in \mathcal{DT}(Q, \Sigma)$, there is an input tree $s = \mathtt{a}(s_1, \ldots, s_n)$, and there is an occurrence of an unevaluated function call $q(s, s_1, \ldots, s_k)$ at node $u \in \mathcal{N}(\alpha)$ of α, and there is a rule

$$\begin{array}{ll} q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) & \to t \quad \text{or} \\ q(x_0, y_1, \ldots, y_k) & \to t' \end{array}$$

in R, such that

$$\begin{array}{l} \beta = \alpha[t[s_1/x_1, \ldots, s_n/x_n, s_1/y_1, \ldots, s_k/y_k]/u], \quad \text{or} \\ \beta = \alpha[t'[s/x_0, s_1/y_1, \ldots, s_k/y_k]/u]. \end{array}$$

The occurrence of the unevaluated call is replaced by the according right-hand side, in which the input variables and formal parameters are substituted by the appropriated input trees and actual parameters, respectively.

(ii) $\alpha \Rightarrow_{M,\mathrm{OI}} \beta$ if $\alpha \Rightarrow_M \beta$, but for u in (i) the following holds: on the path from u to the root of α no function symbol occurs. This restriction guarantees that the outermost function call is evaluated first.

(iii) $\alpha \Rightarrow_{M,\mathrm{IO}} \beta$ if $\alpha \Rightarrow_M \beta$, but the parameters $s_1, \ldots, s_k$ are restricted to be elements of $\mathcal{T}_\Sigma$, i.e., parameters are fully evaluated when they are passed to a subsequent transformation step.

For both binary relations $\Rightarrow_{\mathrm{IO}}$ and $\Rightarrow_{\mathrm{OI}}$, $\overset{*}{\Rightarrow}$ and $\overset{+}{\Rightarrow}$ are defined as usual.

Definition 4.4 Let $\mu \in \{\mathrm{IO}, \mathrm{OI}\}$ and let $M = (Q, \Sigma, Q_0, R)$ be a stay-mtt. The μ-translation realized by M on a non-empty input tree s, denoted by $\pi_{M,\mu}$, is

$$\pi_{M,\mu} = \{(s, t) \in \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma \mid \exists q \in Q_0 : q(s) \overset{*}{\Rightarrow}_{M,\mu} t\}. \triangleleft$$

After having presented a fixpoint characterization of stay-mtts and a formalization of the induced transformation by means of a derivation relation, we conclude with the fact that both characterizations are equivalent. As for TL this justifies the usage of the fixpoint semantics in our proofs and considerations.

Theorem 4.5 Let $M = (Q, \Sigma, Q_0, R)$ be a stay macro tree transducer and let $\mu \in \{\mathrm{IO}, \mathrm{OI}\}$. Then for all input trees $s \in \mathcal{T}_\Sigma$ holds: $t \in \tau_{M,\mu}(s)$ implies that $(s, t) \in \pi_{M,\mu}$.

A formal proof that the two semantics coincide for mtts, can be found in [EV85; FV98]. The proof for stay-mtts is a straight-forward generalization, because the only feature going beyond the functionality of macro tree transducers are the stay-rules. A stay-rule for function q can be chosen instead of every other matching q, $\mathtt{a}$-rule (for a symbol $\mathtt{a}$); and it does not proceed to the subtrees of the current input, but passes the complete input to the subsequent transformation steps.

4.3 Basic Properties

The first theorem recalls a classical result for macro tree transducers. It states that taking pre-images of macro tree transductions and compositions of mtts effectively preserve recognizability.

Theorem 4.6 (7.4 of [EV85]) Let F be the composition of a finite number of relations of inside-out and outside-in mtts. Then, the class of recognizable tree languages RECOG is closed under F^{-1}.

As an immediate consequence, we obtain that type checking is decidable for compositions of mtts. For a given output type T_{out}, we compute the pre-image $F^{-1}(\overline{T_{\text{out}}})$, where $\overline{\Rightarrow}$ denotes the complement of T_{out}. Check whether the intersection with a specified input type is empty or not, solves the type checking problem. In particular, if the intersection is empty, the transformation of a correct input document never results in an incorrect output document.

Corollary 44 of [EM03a] generalizes this result to stay macro tree transducers. This means, for compositions of stay-mtts, it is decidable for an output language in $\mathsf{s\text{-}MTT}(\mathsf{RECOG})$ whether or not it is included in a given regular tree language. The decision algorithm is similar to that for macro tree transducers.

The proof for the next theorem proceeds along the lines of the one given at the end of [EM03a]. We give the explicit construction, in order to illustrate the and to exhibit the according the complexity.

Theorem 4.7 Let M be an outside-in stay macro tree transducer and let T be a recognizable set. Then,

(i) the pre-image $\tau_{M,T}$ is recognizable, and
(ii) the computation of the pre-image for a fixed T can be performed in deterministic exponential time.

Proof. Let $M = (Q, \Sigma, Q_0, R)$ be a stay-mtt. Furthermore, let $A = (P, \Sigma, \delta, F_A)$ be a *right-to-left* bottom-up tree automaton such that $\mathcal{L}(A) = T$.

From automaton A and transducer M we construct a deterministic automaton $B = (D, \Sigma, \beta, F_B)$ recognizing $\tau_M^{-1}(T)$, i.e., $\mathcal{L}(B) = \tau_M^{-1}(T)$. The key

issue of B is to simulate the accepting computations of A on the right-hand sides of the transformation rules of M.

Let Dom denote the set of all combinations of a function $q \in Q$ and all suitable tuples of binary relations on automata states

$$Dom = \{(q, S_1, \ldots, S_k) \mid q \in Q_k \text{ and } S_i \subseteq P, i = 1, \ldots, k\}.$$

Then, the set D of states of automaton B is given by

$$D = Dom \to 2^P.$$

A state $d \in D$ assigned to an input tree t records for every function q of the stay-mtt and every possible sequence $S_1, \ldots, S_k$ of state sets for the accumulating parameters the set of all states which can be obtained by calling q on t and actual parameters satisfying the S_i.

Given this set of new states D, we define $\delta_{\mathtt{a}}(d_1 \ldots d_n) = d$ where

$$d(q, S_1, \ldots, S_k) = \bigcup_{i=1}^{m} [t_i]\ \sigma\ \rho$$

if $q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t_1 | \ldots | t_m$ is a rule in M, $\sigma(x_j) = d_j$ for $j = 1, \ldots, n$, and $\rho(y_i) = S_i$ for all $i = 1, \ldots, k$. Accordingly, for every symbol $\mathtt{a} \in \Sigma_n$, $n \geq 0$, and $d_1, \ldots, d_n \in D$, let $\delta_{\mathtt{a}}(d_1 \ldots d_n) = d$ where d is defined as follows. For every stay-rule $q(x_0, y_1, \ldots, y_k) \to t_1 | \ldots | t_m$, holds $d(q, S_1, \ldots, S_k) = \bigcup_{i=1}^{m} [t_i]\ \sigma\ \rho$ with $\sigma(x_0) = d$ and $\rho(y_i) = S_i$ for all $i = 1, \ldots, k$.

The set F_B of final states of the pre-image automaton B consists of all mappings d such that for each $q_0 \in Q$, set $d(q_0)$ contains a state $p_0 \in F_A$.

The mapping $[]$, which we used to define d_0 and the transition function δ, simulates the computations of automaton A on the right-hand sides of the transformation rules. It is formalized with respect to assigments σ for input variables and ρ for parameters as follows

$$\begin{aligned}
[y_j]\ \sigma\ \rho &= \rho(y_j) \\
[\mathtt{b}(t_1, \ldots, t_n)]\ \sigma\ \rho &= \{p \in P \mid \exists p_1, \ldots, p_n \in P : \\
&\qquad p_i \in [t_i]\ \sigma\ \rho, p = \beta_{\mathtt{a}}(p_1 \ldots p_n)\} \\
[q'(x_i, t_1, \ldots, t_k)]\ \sigma\ \rho &= (\sigma(x_i))(q', [t_1]\ \sigma\ \rho, \ldots, [t_k]\ \sigma\ \rho).
\end{aligned}$$

For the maximal number k of parameters of a function, the number of states of automaton B is at most

$$|Q \to (2^P)^k \to 2^P| = (2^{|P|})^{(2^{|P|})^k \cdot |Q|} \leq 2^{|P| \cdot |Q| \cdot 2^{|P| \cdot k}}$$

This means, the pre-image is double-exponential in the size of the automaton A representing the output type, but only single-exponential in the size of transducer M. Since the construction can be performed in time polynomial in

the size of the computed automaton B, the upper complexity bound follows. □

Intuitively, the idea of the pre-image automaton B is to run automaton A on the right-hand sides of the rules of M. This is possible by extending A appropriately, because a right-hand side ξ might contain parameters y_j or recursive function calls of the form $q'(x_i, t_1, \ldots, t_k)$.

The most fiddly situation is the handling of parameters. Since the state p_j in which A arrives after processing the actual parameter forest f_j of parameter y_j is not *a priori* determined, a state d of the pre-image automaton B has to record all possible choices of states of A for the parameters, and thus $d(q)$ is a function mapping k-tuples of sets of states in P to a set of states in P. A concrete mapping $d(q, p_1, \ldots, p_k) = p$ epitomizes the following situation: assuming state p_j for each actual parameter t_j, then automaton A is in state p after processing the right-hand side ξ.

Then recursive calls can be simulated by applying d_i to the called function q' provided that d_i represents the state in which A arrives at the i-th input sub-tree.

Consequently, whenever reasoning about the transition function of the pre-image automaton B, one has to keep in mind that it takes all state transitions into account which can be obtained by calling a function on some input sub-forest. In other words, if B changes its state to d on some input forest f, this has to be interpreted as follows. In $d(q)$ is exactly that state transition tabulated which is performed by automaton A on a tree t' provided that t' is the result of applying function q to input tree t. This elucidates the immense size of the pre-image automaton compared with the size of the output type.

For deterministic stay macro tree transducers, it suffices to define the pre-image automaton over a state set D, which consists of all mappings d such that for every stay-mtt function $q \in Q_{k+1}$, $d(q)$ is a mapping from P^k to P. The number k again denotes the maximal number of parameters of a function in Q. This means, every state d records for every parameter position only a single state of the automaton. Thus, D consists of all functions

$$Q \to P^k \to P.$$

Accordingly, the number of states reduces to $|P|^{|Q|\cdot|P|^k}$.

So far we have the following complexity results for computing the pre-image of outside-in macro tree transductions:

OI	**dfta**	**nfta**
dstay-mtt	$\lvert P\rvert^{\lvert Q\rvert\cdot\lvert P\rvert^k}$	
stay-mtt		$2^{\lvert P\rvert\cdot\lvert Q\rvert\cdot 2^{\lvert P\rvert\cdot k}}$

The situation for inside-out macro tree transductions differs in the complexity for computing the pre-image of nondeterministic stay-mtts. In the inside-out mode for evaluating nested function calls, the innermost call is

evaluated first. This means that we have to record in the states of the pre-image automaton only a single state for each parameter position. Thus, we have the following complexity results:

IO	dfta	nfta
dstay-mtt	$\lvert P\rvert^{\lvert Q\rvert \cdot \lvert P\rvert^k}$	
stay-mtt		$2^{\lvert P\rvert^{k+1} \cdot \lvert Q\rvert}$

In the next lemma, we repeat a result about the connection between stay macro tree transducers and the original macro tree transducer [EM03a]. The difference between mtts and stay-mtts is that the translation of the latter ones may be infinite, i.e., there are stay-mtts that generate infinitely many output trees for one input tree. A typical example is the following transducer mon_Σ:

$$
\begin{array}{lll}
1 & q_0(x_0) & \to q(x_0, {q'}_\epsilon(x_0,\epsilon)) \\
2 & q(\mathtt{a}(x_1,\ldots,x_n), y) & \to {q'}_\mathtt{a}(x_1, \mathtt{a}(q(x_1, {q'}_\epsilon(x_1,\epsilon)),\ldots,q(x_n, {q'}_\epsilon(x_n,\epsilon)))) \\
3 & q(\epsilon, & y) \to y \\
4 & {q'}_\mathtt{a}(x_0, & y) \to \overline{\mathtt{a}}({q'}_\mathtt{a}(x_0,y)) \mid y \\
5 & {q'}_\epsilon(x_0, & y) \to \overline{\epsilon}({q'}_\epsilon(x_0,y)) \mid y
\end{array}
$$

This transducer generates above each occurring symbol $\mathtt{a}$ of the input tree an arbitrary long sequence of monadic symbols $\overline{\mathtt{a}}$ (i.e., $rank(\overline{\mathtt{a}}) = 1$). Figure 4.3 illustrates the induced transformation.

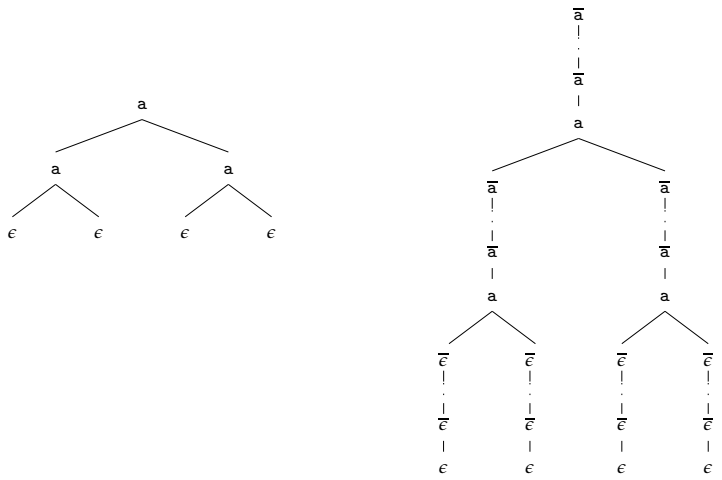

Fig. 4.1. Transducer mon_Σ transforms the tree on the left into a tree of the form as shown on the right-hand side.

The transformation starts with function q_0 at the root node of the input tree. It calls function q for the root and generates in the parameter position

of q a sequence of $\overline{\epsilon}$ symbols. Function q decides depending on the label of the current node whether it has to generate a new sequence of monadic symbols or, if the current node is labeled with ϵ, to output the pre-computed $\overline{\epsilon}$ sequence. This sequence has to be generated in the previous step because when the transducer reaches ϵ with function q, there does not exist any input variable for which the generation process could be started. When function q transforms a node labeled with a, then it generates via function q' the sequence of overlined symbols and passes to q' the symbol a together with its recursively transformed successors. Since q' consists only of a stay-rule, the node to which it is applied is not relevant. Function q' recursively produces one overlined symbol above the other, or ends by writing the pre-computed tree rooted at the original node.

This translation can be used to simulate an arbitrary stay-mtt M by an mtt: first, mon_Σ translates input s into the "stretched" version s', then a macro tree transducer M' can be constructed that simulates on s' the stay-mtt M. Whenever M executes a stay, M' moves down the unary symbols inserted by mon_Σ. Thus, we have for MON denoting the class of all translations mon_Σ:

Lemma 4.8 (Lemma 27 of [EM03a]) $\text{s-MTT} \subseteq \text{MTT} \circ \text{MON}$

One of the important questions in formal language theory is whether a given language contains any words. This means, one wants to know, whether an given automaton accepts or a grammar generates the empty language.

The same question arises for transductions: the most common problem for a given stay-mtt M is to decide whether or not the transduction is nonempty. In particular, this problem is important in the context of type checking. The reason that a transducer produces no output can be twofold — the transducer is defined for the wrong input type or the functions producing output are unspecified or never reached. We recall a well-known result of tree transducer theory (see for example [MPS06]):

Theorem 4.9 Deciding whether the transduction τ_M for a stay macro tree transducer M is not empty is DEXPTIME-complete.

Proof. The lower bound follows since the translation nonemptiness is hard in DEXPTIME already in absence of parameters, i.e., for top-down tree transducers [Sei94a].

In order to show the construction, assume that $M = (Q, \Sigma, Q_0, R)$ is the stay-mtt. Furthermore, assume w.l.o.g. that for every state $q \in Q$ with k accumulating parameters there is a rule $q(x_0, y_1, \ldots, y_k) \to q(x_0, y_1, \ldots, y_k)$ in R. These rule will be used when checking the nonemptiness of several functions simultaneously where for some states stay-rules are selected.

For every subset $S \subseteq Q$, we introduce a propositional variable $[S]$ where $[S] = \mathit{true}$ denotes the fact that

$$\exists t \in \mathcal{T}_\Sigma \forall q \in S : [\![q]\!](t) \neq \emptyset.$$

In particular, for the empty set $\{\}$ we have the implication $[\{\}] \Leftarrow \mathit{true}$. For all selections of stay-rules $q(x_0, y_1, \ldots, y_k) \to t_q$ of R, $q \in S$, we consider all propositional implications

$$[S] \Leftarrow [S_0]$$

with $S_0 = \{p \in Q \mid \exists q \in S : p(x_0, \ldots) \text{ occurs in } t_q\}$. Accordingly, we consider all implications

$$[S] \Leftarrow [S_1] \wedge \ldots \wedge [S_n]$$

for all selections of rules $q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t'_q$, $q \in S$, for the same symbol $\mathtt{a} \in \Sigma$ where $S_i = \{p \in Q \mid \exists q \in S : p(x_i, \ldots) \text{ occurs in } t'_q\}$.

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 τ_M is nonempty if and only if for some $q \in Q_0$, $[\{q\}] = \mathit{true}$ follows from $\mathcal{C}_M$. Since systems of such propositional Horn clauses can be solved in linear time [DG84], the assertion follows. □

4.4 Stay Macro Forest Transducers

Generally, XML transformation languages are defined to work on forests rather than on trees, because the XML standard does not fix the number of children of the elements. Since stay macro tree transducers are defined on ranked alphabets, their major deficiency is that they do not operate on forests directly but refer to representations of forests as ranked trees. Thus, they do not support concatenation of intermediate forests. For two forests $f = f_1 \cdots f_n$ and $f' = f'_1 \cdots f'_m$ the concatenation is defined as $f\, f' = f_1 \cdots f_n f'_1 \cdots f'_m$.

In [PS04] it is shown that this limitation can be lifted. There, the so-called *macro forest transducer* is proposed which operates on forests directly. They generalize macro tree transducers by providing concatenation as additional basic operation on output forests. This extra feature implies that some transductions realized by a macro forest transducer cannot be performed by a single macro tree transducer alone but only by the composition of a macro tree transducer with a transformation depending on the used alphabet.

Here, we additionally enhance this model by allowing to process a node of the input arbitrarily often. This means that we introduce stay-rules and thus, generalize macro forest transducers in the same way as stay-mtts generalize mtts.

The *stay macro forest transducer* has to deal with arbitrary long sequences of direct successors of a node. Therefore, we define it on a *first-child next-sibling* encoding of forests. Recall that this encoding describes a forest in the following way: the left child of a node represents its content and the right child its right sibling (cf. Figure 2.3 on page 10). In order to indicate that these transducers work on forests, we use here a slightly different notation

for the input patterns and the output. $q, \mathtt{a}$-rules match now forests of the form $\mathtt{a}\langle x_1\rangle\, x_2$ and stay-rules differ only in the right-hand sides. Additionally, we need a new form of rules that match the empty forest because it indicates that the computation has reached a leaf. These rules are of the form $q(\epsilon, y_1, \ldots, y_k) \to f$ where symbol ϵ signalizes the empty forest.

Definition 4.10 (Stay Macro Forest Transducer) A *stay macro forest transducer* (stay-mft) is a tuple

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

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

$$\begin{array}{lll} q(\epsilon, & y_1, \ldots, y_k) \to f & \text{or} \\ q(x_0, & y_1, \ldots, y_k) \to f & \text{or} \\ q(\mathtt{a}\langle x_1\rangle\, x_2, & y_1, \ldots, y_k) \to f & \end{array}$$

where $q \in Q$ is a function symbol of rank $k+1$, $\mathtt{a}$ is an input symbol from Σ, x_0, x_1 and x_2 are input variables, $y_1, \ldots, y_k, k \geq 0$ are the accumulating parameters of q, and f is an expression describing the output actions of the corresponding rule. Possible actions are composed by the grammar

$$f ::= \epsilon \mid \mathtt{b}\langle f\rangle \mid y_j \mid q'(x_i, t_1, \ldots, t_m) \mid f_1\, f_2$$

where $\mathtt{b} \in \Sigma$ is a label of an output node, y_j is one of the accumulating parameters ($1 \leq j \leq k$), $q' \in Q$ is of rank $m+1$, and x_i is one of the input variables of the left-hand side. ◁

Intuitively, the meaning of the actions is as follows. The transducer can produce the empty forest ϵ or a single tree $\mathtt{b}\langle f\rangle$ whose children are determined by expression f. It is also possible to emit the content of the accumulating parameter y_j and to call function q recursively for one of the input variables of the left-hand side. This means, right-hand sides of stay-rules can contain only variable x_0 while in all other $q, \mathtt{a}$-rules the variables x_1 and x_2 can occur. Finally, $f_1\, f_2$ concatenates the two recursively determined forests f_1 and f_2.

4.4.1 Induced Transformation

In order to formalize the transformation induced by a stay-mft, we carefully have to distinguish between the two evaluation strategies for nested function calls. We will first define the semantics of inside-out transducers and afterwards of the opposite evaluation method. For the following let $M = (Q, \Sigma, Q_0, R)$ be an arbitrary stay macro forest transducer.

Inside-Out Evaluation

The meaning of a function symbol $q \in Q$ with k accumulating parameters is a function from input forests to sets of forests which may contain parameters from the set $Y = \{y_1, \ldots, y_k\}$

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

These functions are inductively defined by

$$\begin{array}{ll} [\![q]\!]_{\text{IO}}(s) & \supseteq [\![t_1[s/x_0]]\!]_{\text{IO}} \\ [\![q]\!]_{\text{IO}}(\mathtt{a}\langle s_1\rangle\, s_2) & \supseteq [\![t_2[s_1/x_1, s_2/x_2]]\!]_{\text{IO}} \\ [\![q]\!]_{\text{IO}}(\epsilon) & \supseteq [\![t_3]\!]_{\text{IO}} \end{array}$$

for stay-rules $q(x_0, y_1, \ldots, y_k) \to t_1$, $q, \mathtt{a}$-rules $q(\mathtt{a}\langle x_1\rangle\, x_2, y_1, \ldots, y_k) \to t_2$ or rules matching the empty forest $q(\epsilon, y_1, \ldots, y_k) \to t_3$, respectively. The evaluation of the right-hand side expressions is defined as follows

$$\begin{aligned} [\![\epsilon]\!]_{\text{IO}} &= \{\epsilon\} \\ [\![y_j]\!]_{\text{IO}} &= \{y_j\} \\ [\![\mathtt{b}\langle s_1\rangle\, s_2]\!]_{\text{IO}} &= \{\mathtt{b}\langle s_1'\rangle\, s_2' \mid s_i' \in [\![s_i]\!]_{\text{IO}}, i = 1,2\} \\ [\![s_1\, s_2]\!]_{\text{IO}} &= \{s_1'\, s_2' \mid s_i' \in [\![s_i]\!]_{\text{IO}}, i = 1,2\} \\ [\![q'(s', t_1, \ldots, t_n)]\!]_{\text{IO}} &= \{t'[t_1'/y_1, \ldots, t_n'/y_n] \mid\ t' \in [\![q']\!]_{\text{IO}}(s'), \\ &\qquad\qquad t_i' \in [\![t_i]\!]_{\text{IO}}, i = 1, \ldots, n\}. \end{aligned}$$

The empty forest is evaluated to the empty forest and an accumulating parameter y_j is mapped to itself. The output of a forest $\mathtt{b}\langle s_1\rangle\, s_2$ is mapped to the set of all potential forests whose first tree is labeled with $\mathtt{b}$ and has as content the evaluation of expression s_1 and whose tail is determined by s_2. The concatenation of two expressions s_1 and s_2 is evaluated to all forests whose first part is an element of the evaluation of s_1 and whose second part accordingly is an element of $[\![s_2]\!]_{\text{IO}}$. The result of a function call $q'(s', t_1, \ldots, t_n)$ is the set of all evaluations t' of the function symbol q' where each occurrence of a parameter $y_i, i = 1, \ldots, k$ is replaced by the forest $t_i' \in [\![t_i]\!]_{\text{IO}}$. This process is a simultaneous substitution of all occurrences.

Outside-In Evaluation

For the contrary evaluation mode, the meaning of a function symbol $q \in Q$ with k accumulating parameters again is a function from input forests to sets of output forests possibly containing variables in $Y = \{y_1, \ldots, y_k\}$:

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

The functions $[\![q]\!]_{\text{OI}}$ are defined as the least fixpoint of the following inclusions

- for stay-rules $q(x_0, y_1, \ldots, y_k) \to t_1$ in R

$$[\![q]\!]_{\mathrm{OI}}(s) \supseteq [\![t_1]\!]_{\mathrm{OI}}\sigma$$

  where $s \in \mathcal{F}_\Sigma$ and $\sigma(x_0) = s$;
- for q, a-rules $q(\mathtt{a}\langle x_1\rangle\, x_2, y_1, \ldots, y_k) \to t_2$ in R with $a \in \Sigma$

$$[\![q]\!]_{\mathrm{OI}}(s) \supseteq [\![t_2]\!]_{\mathrm{OI}}\sigma$$

  where $s = \mathtt{a}\langle s_1\rangle\, s_2$ with $s_1, s_2 \in \mathcal{F}_\Sigma$ and $\sigma(x_i) = s_i, i = 1, 2$;
- for empty forest rules $q(\epsilon, y_1, \ldots, y_k) \to t_3$

$$[\![q]\!]_{\mathrm{OI}}(s) \supseteq [\![t_3]\!]_{\mathrm{OI}}\emptyset$$

  where $s = \epsilon$ and $\emptyset$ is the empty assignment.

The right-hand side expressions are evaluated by means of the function $[\![\cdot]\!]_{\mathrm{OI}}\sigma$ where σ binds input variables to the current input, i.e.,

$$\begin{aligned}
[\![\epsilon]\!]_{\mathrm{OI}}\sigma &= \{\epsilon\} \\
[\![y_j]\!]_{\mathrm{OI}}\sigma &= \{y_j\} \\
[\![\mathtt{b}\langle s_1\rangle\, s_2]\!]_{\mathrm{OI}}\sigma &= \{\mathtt{b}\langle s_1'\rangle\, s_2' \mid s_i' \in [\![s_i]\!]_{\mathrm{OI}}\sigma, i = 1, \ldots, m\} \\
[\![s_1\, s_2]\!]_{\mathrm{OI}}\sigma &= \{s_1'\, s_2' \mid s_i' \in [\![s_i]\!]_{\mathrm{OI}}\sigma, i = 1, 2\} \\
[\![q'(x_i, s_1, \ldots, s_n)]\!]_{\mathrm{OI}}\sigma &= L[y_1 \leftarrow L_1, \ldots, y_n \leftarrow L_n],
\end{aligned}$$

where $L = [\![q']\!]_{\mathrm{OI}}(\sigma(x_i))$ and $L_j = [\![s_j]\!]_{\mathrm{OI}}\sigma$ for each $j = 1, \ldots, n$.

The empty forest is mapped to itself and a variable y_i is replaced by the corresponding content of the i-th actual parameter. The output of a forest $\mathtt{b}\langle s_1\rangle\, s_2$ is mapped to the set of all forests f starting with a tree which is labeled with $\mathtt{b} \in \Sigma$ and whose content is recursively determined by s_1. The tail of f is determined by the evaluation of expression s_2. The concatenation of expressions s_1 and s_2 is mapped to all sequences of trees whose first part is a forest of $[\![s_1]\!]_{\mathrm{OI}}$ and whose second part is a forest of $[\![s_2]\!]_{\mathrm{OI}}$. Finally, a call of q' is mapped to the interpretation of the donation function $[\![q']\!]_{\mathrm{OI}}$ for the content or right context of the current input together with the actual parameters given by the evaluation of the expressions $s_1, \ldots, s_n$. The parameters are again replaced by means of simultaneous OI substitution (cf. Section 2.2).

With the definitions of the semantics of functions we are now able to define the transduction induced by a macro forest transducer.

Definition 4.11 Let $\mu \in \{\mathrm{IO}, \mathrm{OI}\}$ and let $M = (Q, \Sigma, Q_0, R)$ be a stay-mft. The μ-transformation realized by M on a non-empty input forest s, denoted by $\tau_{M,\mu}$, is for both evaluation modes a function from forests to sets of forests $\tau_{M,\mu} : \mathcal{F}_\Sigma \to 2^{\mathcal{F}_\Sigma}$. These functions are induced by the initial functions from Q_0 and they are defined as

$$\tau_{M,\mu}(s) = \bigcup_{q \in Q_0} [\![q]\!]_\mu(s). \triangleleft$$

For a given set $S \subseteq \mathcal{F}_\Sigma$ we denote by $\tau_{M,\mu}(S)$ the set of all outputs which are produced by M on input forests in S, i.e.,

$$\tau_{M,\mu}(S) = \bigcup_{s \in S} \tau_{M,\mu}(s),$$

where μ again denotes the evaluation strategies IO or OI, respectively.

4.4.2 Characterization

In this section we give some characterizations and basic properties of stay macro forest transducers. In particular, we characterize the relation between the height of the input and the output of a stay-mft. Taking advantage of this result we compare stay-mfts with their tree variant and show that the latter ones are a subclass of stay-mfts. Moreover, we present a decomposition of stay macro forest transducers into a stay-mtt and a transduction which performes the concatenation of forests. We close this section by a result stating that recognizable languages are closed under taking pre-images of stay macro forest transductions. The corresponding results for macro forest transducers (without stay rules) can be found in [PS04].

Recall that the binary height equals the height of a forest considered as a binary tree [PS04] — its formal definition is given in Definition 2.6. Our first result relates the binary height of the input with the binary height of the output. Note that stay macro forest transducers can produce forests of arbitrary height by recursively choosing a stay-rule and accumulating an increasing output at every step. Thus, the binary height *bht* of an output forest is either unbounded or, for a finite transduction, it can be exponentially larger then the binary height of the input.

Lemma 4.12 Let Σ be an alphabet. There is a stay-mft M such that for an infinite number of forests $f \in \mathcal{F}_\Sigma$, $bht(s) \geq 2^{2^{bht(f)}}$ for every $s \in \tau_M(f)$.

Proof. Consider the total and deterministic stay-mft $M = (Q, \Sigma, q_0, R)$ where $Q_0 = \{q_0, q\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and R consists of the rules

$$\begin{array}{ll} q_0(a\langle x_1\rangle\ x_2) & \to q(x_2, q(x_2, \mathtt{b}) \\ q_0(\epsilon) & \to \mathtt{b}\,\mathtt{b} \\ q(a\langle x_1\rangle\ x_2, y) & \to q(x_2, q(x_2, y) \\ q(\epsilon, y) & \to y\,y. \end{array}$$

The last rule concatenates parameter y to itself, thus it is doubling the binary height of the accumulated output forest.

We first show by induction on the length $k \geq 0$ of the input forest $s = a^k$ that function q produces a forest y_m with $m = 2^{2^k}$.

For $k = 0$, we have $[\![q]\!](s) = \{y\,y\} = \{y^2\}$, and the assertion follows since $2^{2^0} = 2^1 = 2$.

For $k > 0$, function q evaluates to

$$\{[\![q]\!](a^{k-1})[([\![q]\!](a^{k-1})[y/y])/y]\}$$

which by induction hypothesis applied to both procedure calls, results in a set $\{y^m\}$ with

$$m = 2^{2^{k-1}} \cdot 2^{2^{k-1}} = 2^{2^k}.$$

The lemma follows by induction on the structure of the input forest. For $f = \epsilon$, the initial function evaluates to $\{\mathtt{b}\,\mathtt{b}\} = \{\mathtt{b}^2\}$ and the assertion follows. For $f = a^n$, we have

$$\{[\![q]\!](a^{k-1})[([\![q]\!](a^{k-1})[\mathtt{b}/y])/y]\}.$$

By our first induction, function q evaluates to a forest y^m with $m = 2^{2^k}$. The lemma follows by substituting every occurrence of y by $\mathtt{b}$. □

Due to their ability to concatenate forests, we show in the next theorem that stay-mfts are strictly more expressive than stay-mtts. This result corresponds to Theorem 8 of [PS04], where tree transducers and forest transducers without stay-rules are compared.

Theorem 4.13 s-MTT $\subset$ s-MFT.

Proof. The height property for s-MTT says that for every stay-mtt M, there is a constant $c > 0$ such that for every input f, if $\tau_M(f)$ is finite then $bht(s) \leq c^{bht(f)}$ (cf. [FV98] in combination with the results of [EM03a]). Hence, the translation of the stay-mft M from the last lemma cannot be expressed by any stay macro tree transducer. □

In the next theorem we show that every stay-mft M can be simulated by a stay macro tree transducers followed by an ordinary macro tree transducer (i.e., without stay-rules). The idea is to split the transformation induced by M into two transductions: the first one essentially executes M's transitions, but with the only difference that each concatenation in the output is replaced by an application of a concatenation symbol "@". In the second step, this symbolical lengthening of forests is evaluated, i.e., at every occurrence of the concatenation symbol its right successor is "transported" to the rightmost leaf of its left successor. This step can be performed by the total and deterministic macro tree transducer *App* with the following rules:

$$\begin{aligned}
q_{\text{in}}(x_0) &\to q(x_0, \epsilon)\\
q(\epsilon, y) &\to y\\
q(\mathtt{a}\langle x_1\rangle\, x_2, y) &\to \mathtt{a}\langle q(x_1, \epsilon)\rangle\, q(x_2, y)\\
q(@\langle x_1\rangle\, x_2, y) &\to q(x_1, q(x_2, y))
\end{aligned}$$

for all symbols $\mathtt{a}$ of the underlying alphabet. Since function *App* is defined for every alphabet, let APP denote the class of all transductions evaluating the symbolic concatenation.

Theorem 4.14 s-MFT $\subseteq$ APP $\circ$ s-MTT.

Proof. Let $M = (Q, \Sigma, Q_0, R)$ be a stay-mft. We construct a new stay-mtt $M' = (Q', \Sigma \cup \{@\}, Q_0', R')$ from M. The set of functions is defined as $Q' = \{\overline{q} \mid q \in Q\}$. Accordingly, the set of initial functions Q_0' consists of all $\overline{q_0}$ such that $q_0 \in Q_0$. Every rule $q(\pi, y_1, \ldots, y_k) \to f$ in R is emulated by a rule $\overline{q}(\pi, y_1, \ldots, y_k) \to rew(f)$ where pattern π is copied unchanged and the right-hand sides f are rewritten by means of

$$\begin{array}{ll} rew(\epsilon) & = \epsilon \\ rew(\mathtt{b}\langle f_1\rangle\ f_2) & = \mathtt{b}\langle rew(f_1)\rangle\ rew(f_2) \\ rew(y_j) & = y_j \\ rew(q(x_i, f_1, \ldots, f_k)) & = \overline{q}(x_i, rew(f_1), \ldots, rew(f_k)) \\ rew(f_1\ f_2) & = @\langle rew(f_1)\rangle\ rew(f_2). \end{array}$$

Let τ_{App} denote the transformation which evaluates the concatenation symbols "@". Moreover, we extend this transformation to sets of forests by

$$\tau_{App}(S) = \{\tau_{App}(s) \mid s \in S\}$$

for every set $S \in \mathcal{F}_{\Sigma \cup \{@\}}$.

By structural induction on input forests $f \in \mathcal{F}_\Sigma$, one verifies for all states $q \in Q$ that

$$[\![q]\!](f) = \tau_{App}([\![\overline{q}]\!](f)).$$

This assertion follows from proving for every sub-forest s occurring in a right-hand side of a rule of M that

$$[\![s]\!]_{\mathrm{OI}}\sigma = \tau_{App}([\![rew(s)]\!]_{\mathrm{OI}}\sigma) \quad \text{or} \quad [\![s]\!]_{\mathrm{IO}} = \tau_{App}([\![rew(s)]\!]_{\mathrm{IO}})$$

for OI- and IO-transductions, respectively. Hence,

$$[\![q_0]\!](f) = \tau_{App}([\![\overline{q_0}]\!](f)$$

for all $q_0 \in Q_0$, and therefore, we can conclude that $\tau_M(f) = \tau_{App}(\tau_{M'}(f))$ for every $f \in \mathcal{F}_\Sigma$. $\square$

As an immediate consequence, we obtain a height property for the class of stay macro forest transductions stating that the height of an output forest is either unbounded or it is double exponentially bounded by the height of the input forest.

Corollary 4.15 Let M be a stay-mft. There is a constant $c > 0$ such that for every input forest f, if $\tau_M(f)$ is finite, then $bht(s) \leq 2^{2^{c \cdot bht(f)}}$ for every output forest $s \in \tau_M(f)$.

The decomposition of a stay-mft into two tree transducers is not arbitrary, but for a given alphabet Σ, it always uses the same deterministic macro tree transducer *App* to evaluate the auxiliary concatenation symbols "@". For every such transducer *App*, we observe

Lemma 4.16 For every forest $f \in \mathcal{F}_{\Sigma\cup\{@\}}$ and $\{s\} = \tau_{App}(f)$, $ht(s) \leq ht(f)$.

Proof. Transducer *App* is defined as above, and thus it is total and deterministic by definition. Let s' be a temporary result accumulated in the parameter position of function q. We first show that for each call $q(f, s')$ holds:

$$ht(s) \leq \max\{ht(f), ht(s')\}.$$

We proceed by structural induction on f.

If $f = \epsilon$, then $[\![q(f, s')]\!] = \{s'\}$. It follows that $s = s'$, and thus the claim follows.

If $f = @\langle f_1\rangle\ f_2$, then $[\![q(f, s')]\!] = \{([\![q]\!](f_1))[([\![q]\!](f_2))[s'/y]/y]\}$. By induction hypothesis for f_2, we obtain $ht(s_2) \leq \max\{ht(f_2), ht(s')\}$ for $\{s_2\} = ([\![q]\!](f_2))[s'/y]$. Applying the induction hypothesis to f_1, we obtain

$$\begin{aligned} ht(s) &\leq \max\{ht(f_1), ht(s_2)\} \\ &= \max\{ht(f_1), ht(f_2), ht(s')\} \\ &\leq \max\{ht(f), ht(s')\} \end{aligned}$$

which we wanted to prove.

Finally, for $f = \mathtt{a}\langle f_1\rangle\ f_2$ for some $\mathtt{a} \in \Sigma$, we obtain $[\![q(f, s')]\!] = \{\mathtt{a}\langle s_1\rangle\ s_2\}$, where

$$\begin{aligned} s_1 &\in [\![q]\!](f_1)[\epsilon/y] \\ s_2 &\in [\![q]\!](f_2)[s'/y] \end{aligned}$$

and the claim follows from the induction hypothesis applied to f_1 and f_2.

The lemma follows by the fact that the initial function q_{in} is evaluated to $[\![q(f, \epsilon)]\!]$ for every input forest $f \in \mathcal{F}_\Sigma$. □

Lemma 4.16 gives us a single exponential upper bound on the height of output forests of stay macro forest transducers:

Theorem 4.17 For every stay-mft M there is a constant $c > 0$ such that — if $\tau_M(f)$ is finite — for every $f \in \mathcal{F}_\Sigma$ and $s \in \tau_M(f)$. $ht(s) \leq 2^{c \cdot bht(f)}$.

The height of the output tree of a stay macro tree transducer is either unbounded or it can be exponentially bounded in the height of the input tree [FV98;EM03a]. In fact, there is a stay macro tree transducer which exactly performs the exponential growth of the input tree. It translates monadic trees of height n into monadic trees of height 2^n [FV98]. Iterating this stay-mtt, a translation in s-MTT^2 — denoting the class of all transductions that can be performed by the composition of two stay-mtts — is obtained which, by Theorem 4.17 cannot by realized by a single stay macro forest transducer. This result is the direct generalization of Corollary 12 in [PS04] to transductions with stay-rules.

Theorem 4.18 s-MFT $\subset$ s-MTT^2.

Since this thesis is mainly concerned with macro transductions rather than top-down transductions (i.e., without accumulating parameters) and since the proof is almost literally the same as for the case without stay-rules in [PS04], we here present only as conjecture the relation between stay top-down tree transductions, stay top-down forest transdcuetions, and stay macro tree transductions. Let therefore s-TOP denote the class of all transductions that can be performed by a stay macro tree transducer without parameters, and let s-FTOP denote the class of all transductions which can be realized by a stay macro forest transducer without parameters.

Conjecture 4.19 s-TOP $\subset$ s-FTOP $\subset$ s-MTT.

It is well known in tree transducer theory that the pre-image of a recognizable tree language with respect to a macro tree transducer is again recognizable and can be effectively computed [EV85]. In [EM03a] this result could even be strengthened to the more general result that type checking of compositions of stay-mtts is decidable.

Since stay-mfts can be simulated by the two-fold composition of stay-mtts, we conculde that the pre-image $\tau_M^{-1}(T)$ of the recognizable language T with respect to the transformation τ_M is recognizable as well and can be effectively constructed. The basic proof idea for the next theorem can be found in [EM03a]. It was adapted to macro forest transducers (without stay-rules) in [PS04]. We give an intuition of the construction to illustrate how concatenation and stay-rules can be treated directly.

Before presenting the result, we first have to discuss, how to deal with forests in the output of the transduction. We have to take into account that our finite representation of the output type is compatible with concatenations. Therefore, we use the idea of a forest algebra (cf. the discussion in [BW05]).

Let Σ be a finite alphabet. Then a *finite forest monoid* (ffm) consists of a finite monoid M with a neutral element $e \in M$, a subset $F \subseteq M$ of accepting elements, and finally, a function $up : \Sigma \times M \to M$ mapping a symbol of the alphabet Σ together with a monoid element for its content to a monoid element representing a forest of length 1.

Given a deterministic bottom-up tree automaton $A = (P, \Sigma, \delta, F_A)$, we can construct a finite forest monoid as follows. Let $M = P \to P$ be the monoid of functions from the set of automata states P into itself where the monoid operation is the function composition $\circ : M \times M \to M$ such that for a state $p \in P$ and monoid elements $f, g \in M$, $(f \circ g)(p) = f(g(p))$. In particular, the neutral element of the monoid is the identity function *id*. Moreover, for symbol $\mathtt{a} \in \Sigma$, monoid element $f \in M$, and state $p \in P$, the function *up* is defined as

$$up(\mathtt{a}, f)(p) = \delta_{\mathtt{a}}(p\ f(\delta_\epsilon))$$

where ϵ is the input symbol denoting the empty forest. Finally, the set of accepting elements is given by

$$F = \{f \in M \mid f(\delta_\epsilon) \in F_A\}.$$

This construction shows that every recognizable forest language can be recognized by a finite forest monoid and although the ffm for a bottom-up tree automaton generally can be exponentially larger, this need not always be the case.

Theorem 4.20 Let M be a stay-mft and T a recognizable set. Then

(1) The pre-image $\tau_M^{-1}(T)$ is recognizable.
(2) An automaton accepting $\tau_M^{-1}(T)$ can be constructed in deterministic exponential time.
(3) Deciding emptiness of the pre-image of recognizable languages with respect to stay-mfts is DEXPTIME-hard.

Proof. Let $M = (Q, \Sigma, Q_0, R)$ be a stay-mft. Furthermore, let $A = (P, \Sigma, \delta, F_A)$ be a finite forest monoid accepting T, i.e., $\mathcal{L}(A) = T$.

From A and transducer M we construct an automaton $B = (D, \Sigma, \beta, F_B)$ recognizing $\tau_M^{-1}(T)$, i.e., $\mathcal{L}(B) = \tau_M^{-1}(T)$.

Let *Dom* denote the set of all combinations of a function $q \in Q$ and all suitable tuples of binary relations on states of A

$$Dom = \{(q, S_1, \ldots, S_r) \mid q \in Q_k \text{ and } S_i \subseteq P \times P, i = 1, \ldots, k\}.$$

Then, the set D of states of B is given by

$$D = Dom \to 2^{P \times P}.$$

Intuitively, a relation $S \in P \times P$ describes the set of all possible state transitions of the automaton for T on the forests from a given set of outputs. A state $d \in D$ assigned to an input forest f records for every function q of the stay-mft and every possible sequence $S_1, \ldots, S_k$ of such effect descriptions for the accumulating parameters the set of all state transitions which can be obtained by calling q on f and actual parameters satisfying the S_i.

Given this set of states D, we define for rules transforming the empty forest:

$$d_0(q, S_1, \ldots, S_k) = \bigcup_{i=1}^{m} [f_i]\ \emptyset\ \rho$$

if $q(\epsilon, y_1, \ldots, y_k) \to f_1 | \ldots | f_m$ is a rule in M, $\emptyset$ is the empty assignment, and $\rho(y_i) = S_i$ for all $i = 1, \ldots, k$.

Moreover, we define $\delta_{\mathsf{a}}(d_1 d_2) = d$ where

$$d(q, S_1, \ldots, S_k) = \bigcup_{i=1}^{n} [f_i]\ \sigma\ \rho$$

if $q(\mathtt{a}\langle x_1\rangle\, x_2, y_1, \ldots, y_k) \to f_1 | \ldots | f_n$ is a rule in M, $\sigma(x_j) = d_j$ for $j = 1, 2$, and $\rho(y_i) = S_i$ for all $i = 1, \ldots, k$.

The set F_B of final states of B consists of all mappings d such that for each $q_0 \in Q$, the relations $d(q_0)$ contain a pair (p, p_0) with $p_0 \in F_A$.

The mapping [] simulates the computations of automaton A on the right-hand sides of the transformations rules. Let $1 \subseteq P \times P$ denote the relation $\{(p, p) \mid p \in P\}$ and let '$\circ$' denote the composition of translations. Then [] is formalized with respect to assigments σ for input variables and ρ for parameters as follows

$$
\begin{array}{ll}
[\epsilon]\ \sigma\ \rho & = 1 \\
[y_j]\ \sigma\ \rho & = \rho(y_j) \\
[f_1\ f_2]\ \sigma\ \rho & = ([f_1]\ \sigma\ \rho) \circ ([f_2]\ \sigma\ \rho) \\
[\mathtt{b}\langle f_1\rangle\ f_2]\ \sigma\ \rho & = \{(b_1, b_2) \in P \times P \mid \exists s_1, s_2 \in P : \\
 & \quad (s_1, b_0) \in [f_1]\ \sigma\ \rho, (s_2, b_2) \in [f_2]\ \sigma\ \rho, b_1 = \beta_{\mathtt{b}}(s_1 s_2)\} \\
[q'(x_i, f_1, \ldots, f_k)]\ \sigma\ \rho & = (\sigma(x_i))(q', [f_1]\ \sigma\ \rho, \ldots, [f_1]\ \sigma\ \rho).
\end{array}
$$

The number of states of B is at most

$$|Q \to (2^{P^2})^k \to 2^{P^2}| = (2^{|P|^2})^{(2^{|P|^2})^k \cdot |Q|} \leq 2^{|P|^2 \cdot |Q| \cdot 2^{|P|^2 \cdot k}}$$

This means, the pre-image is double-exponential in the size of the automaton A representing the output type, but only single-exponential in the size of transducer M. Since the construction can be performed in time polynomial in the size of the computed automaton B, the upper complexity bound follows (given that n is bounded by some constant). For the lower bound, we recall that it holds already for the class of top-down transductions [MN02] and thus also holds for s-MFT. □

The main difference to the tree variant is that for forest transducers, relations of states have to be recorded for every parameter position of a stay-mft function. Accordingly, we have the following complexity results for outside-in stay macro forest transdcuers:

OI	**dfta**	**nfta**
dstay-mft	$\lvert P\rvert^{\lvert Q\rvert \cdot \lvert P\rvert^k}$	
stay-mft		$2^{\lvert P\rvert^2 \cdot \lvert Q\rvert \cdot 2^{\lvert P\rvert^2 \cdot k}}$

Note that p stands now for monoid elements. As in the case of tree transductions, the difference of outside-in and inside-out evaluations is that the pre-image automaton has to record only a single state for each parameter position of a function. Thus, the complexities for inside-out stay-mfts are as follows:

IO	**dfta**	**nfta**
dstay-mft	$\lvert P\rvert^{\lvert Q\rvert \cdot \lvert P\rvert^k}$	
stay-mft		$2^{\lvert P\rvert^{2(k+1)} \cdot \lvert Q\rvert}$

4.5 Stay-Mtts with Regular Look-ahead

An interesting extension of our transducer model are *stay macro tree transducers with look-ahead*. Then, the transducer is allowed to inspect the subtrees of a node before processing it. In other words, the new model has an arbitrarily large look-ahead. The idea of a regular look-ahead also occurs in the theory of parsing of context-free languages. It was first combined with a transducer model in [Eng77].

A rule can be applied only if the current node is an element of a recognizable tree language. To be more specific, each rule is additionally equipped with a state p of a bottom-up tree automaton A, thus it is of the form

$$q(\mathtt{a}(x_1,\ldots,x_n),y_1,\ldots,y_k)^p \to t.$$

This rule is applicable to an input tree s, if the root of s is labeled with symbol $\mathtt{a}$ and moreover, if there is a run of automaton A such that state p is assigned to s.

Definition 4.21 (Stay-mtt with Look-ahead) A *stay macro tree transducer with regular look-ahead* is a pair

$$M^A = (M, A)$$

consisting of a bottom-up finite tree automaton $A = (P, \Sigma, F, \delta)$ and a stay macro tree transducer $M = (Q, \Sigma, Q_0, R)$ whose rules are of the form

$$\begin{array}{ll} q(x_0, & y_1,\ldots,y_k)^p \to t \quad \text{or} \\ q(\mathtt{a}(x_1,\ldots,x_n), & y_1,\ldots,y_k)^p \to t \end{array}$$

where $q \in Q$, $\mathtt{a} \in \Sigma$, p is a *look-ahead state* in P, and t is composed by the same grammar as for stay-mtts. ◁

As observed in [EV85], regular look-ahead does not enlarge the class of translations realized by nondeterministic macro tree transducers. This means each transducer with regular look-ahead can be simulated by a transducer without look-ahead. A detailed and exact construction can be found in [EV85]. The proof idea is to combine all right-hand sides of rules for the same function and input symbol into one right-hand side that starts with a "test tree". This tree is organized like nested if-then-else statements such that each original right-hand side is arranged below the corresponding look-ahead. There are additional rules imitating the if-then-else construction for each look-ahead state.

Here, we prefer a more practical simulation of look-ahead. A stay-rule $q(x_0, y_1, \ldots, y_k)^p \to t$ for a look-ahead state $p \in P$ of the finite tree automaton A can be transformed into a rule

$$q(x_0, y_1, \ldots, y_k) \to \mathrm{check}_p(x_0, t)$$

where the test of the look-ahead p is postponed to the new function check_p. It is defined as

$$\mathrm{check}_p(\mathtt{a}(x_1,\ldots,x_n),y) \to \mathrm{check}_{p_1}(x_1,\mathrm{check}_{p_2}(x_2,\ldots\mathrm{check}_{p_n}(x_n,y)\ldots))$$

for all transitions $(p,\mathtt{a},p_1\ldots p_n)$ of automaton A. Analogously, each $q,\mathtt{b}$-rule with look-ahead state p can be simulated by a new rule check_p which checks for both, the input symbol $\mathtt{b}$ and look-ahead state p. The size of the stay-mtt without look-ahead is linear in the sum of the sizes of the original stay-mtt with look-ahead and of the finite tree automaton A.

4.6 Notes and references

Fischer submitted his thesis about "Grammars with Macro-like Productions" in 1968 [Fis68a]. His motivation was that context-free languages inadequately model some features of programming languages. On the other hand, Fischer was convinced that only a formal model could help to describe the programming language to other people or to investigate certain properties of programs. In order to meet his idea of a natural and descriptive formal model for programming languages, for which emptiness is still decidable, Fischer generalized the rules of context-free grammars by borrowing from programming the idea of macros. A macro has a name, a list of (named) arguments, and a body, which may contain occurrences of the argument names, other macros, and terminal symbols. A macro is expanded by replacing its name and any arguments with its corresponding body into which the arguments have been substituted for the argument names. With this intuition in mind, he defines a macro grammar as a rewriting system, where each non-terminal can have a list of arguments, which may be used in the right-hand sides of rewriting rules. In this way, for example, he can define a grammar system as follows:

$$\begin{aligned} S &\to F(\mathrm{a},\mathrm{b},\mathrm{c}) \\ F(x,y,z) &\to F(x\mathrm{a},y\mathrm{b},z\mathrm{c}) \\ F(x,y,z) &\to xyz \end{aligned}$$

Starting with the non-terminal S, the grammar rewrites it to $F(\mathrm{a},\mathrm{b},\mathrm{c})$. Applying the second rule, for example two times, it produces $F(\mathrm{aa},\mathrm{bb},\mathrm{cc})$. The third rule then emits the word "aabbcc". This grammar obviously generates the language $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geq 1\}$ which is not context-free [Fis68a]. Since macro calls can be nested in the right-hand sides, Fischer considers both evaluation orders: inside-out and outside-in, because they generate different classes of languages.

Engelfriet takes up these ideas and combines them with the features of top-down tree transducers [Eng80]. Here the point of view has changed: while macro grammars generate some output starting with an initial non-terminal, now the choice of the rules is triggered by some input tree. This

is the time of birth of the *macro tree transducer*. The first extensive investigation can be found in [EV85]. Deterministic macro tree transducers are investigated in detail in [FV98], where they are compared with other formal models of syntax-directed semantics. "*Syntax-directed semantics is based on the idea that the semantics of a string of a context-free language is defined in terms of the semantics of the syntactic substructures of the string*" [FV98]. This means, if the semantics is specified in a syntax-directed way, it is necessary that each string is already parsed before the computation of its semantic value can begin. The authors introduce macro tree transducers as a formal model to specify syntax-directed semantics.

Fülöp and Vogler additionally define in [FV98] an *environment* E so they do not need to fix the rank of the initial state to 1. They define a deterministic mtt M as the tuple

$$(Q, \Sigma, \Delta, q_0, R, E)$$

where Q is the set of states, Σ and Δ are the input and output alphabet, respectively, $q_0 \in Q$ is a designated state called the initial state, and R is a set of rules of the form

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

with $q \in Q$, $\mathtt{a} \in \Sigma$, and t is defined as in Definition 4.1 but contains only symbols from the output alphabet Δ. Finally, $E = (t_1, \ldots, t_r)$ where $t_1, \ldots, t_r \in \mathcal{T}_\Delta$ and $r + 1 = \mathit{rank}(q_0)$. Let $M = (Q, \Sigma, \Delta, q_0, R, E)$ be such an mtt with $E = (t_1, \ldots, t_r)$. Then we can construct a stay-mtt $N = (Q_N, \Sigma_N, Q_0, R_N)$ with the same behaviour as M. Therefore, we define $Q_N = Q \cup q'$ for a new state q', $\Sigma_N = \Sigma \cup \Delta$, $Q_0 = \{q'\}$, and $R_N = R \cup \{r\}$ where the initial rule r of the stay-mtt N is defined as

$$q'(x_0) \to q_0(x_0, t_1, \ldots, t_r).$$

The complete input tree is passed on to the initial function q_0 of M; at the same time the accumulating parameters are set to the trees t_i of the environment E. This means the new rule r only calls function q_0 on the input together with the parameters defined in the environment without changing the transformational behaviour.

Stay macro tree transducers were first mentioned in [EM03a]. There, k-pebble tree transducers are realized by an $(k + 1)$-fold composition of macro tree transducers. Conversely, every macro tree transducer can be simulated by a composition of pebble tree transducers. Therefore, the mtt must be extended by the ability to remain at a node, instead of strictly moving down in each transformation step. The idea is to encode the behaviour of each single pebble into one (somehow simple) transducer and to realize the transformation core by a nondeterministic macro tree transducer. Each of the first k transducers implements the same total and deterministic function that adds information about the position of a pebble to the input tree. Engelfriet and Maneth define the stay macro tree transducer to have rules of the form

$$\langle q, \mathtt{a}, b, j\rangle(y_1, \ldots, y_k) \to \zeta,$$

where q denotes a state of the transducer, $\mathtt{a}$ is a symbol of the input tree, and j is the child number of the current node (j for the j-th child; 0 for the root node). Right-hand sides ζ are trees over output symbols, state-instruction pairs and parameters. The state-instruction pairs can have the two forms

$$\langle q, \text{stay}\rangle \quad \text{or} \quad \langle q, \text{down}_i\rangle$$

where i denotes the child to which state q should be applied. This means, there is no explicit distinction between stay-rules and q, $\mathtt{a}$-rules as in our definition. In a deterministic setting stay operations are superfluous [EM03a], and thus, both transducers models are equivalent. In the nondeterministic case, both models can be transformed into each other, because whenever a stay-instruction occurrs in a right-hand side of Engelfriet and Maneth's transducers, a stay-rule of the transducers as defined in Definition 4.1 can be chosen and vice versa.

Note that in [MBPS05], stay-mtts are defined in a slightly different way. The rules there are of the form

$$q(x_0 \text{ as } \mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t.$$

For an input tree $s = \mathtt{a}(s_1, \ldots, s_n)$, variable x_0 is bound to the current input s while for all $i = 1, \ldots, n$, variable x_i is bound to the according subtree s_i. Variable 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.

An interesting result about macro tree transducers (without stay-rules) is that every composition computing a function of *linear size increase* can be realized by just one macro tree transducer [Man03]. Linear size increase means that the size of the output is linear bounded in the size of the input. In [EM03b] it is proved that for a total deterministic mtt with look-ahead, it is decidable whether or not it is of linear size increase. The proof puts the term of linear size increase down to the term of finite copying. In this context, finite copying means that every node of the input tree is transformed only by a bounded number of states (functions), and that each parameter is also copied a bounded number of times only. Since this property is decidable for mtts, one has an algorithm to decide for a composition of mtts whether it is of linear size increase, and if so to effectively construct a single, transduction-equivalent mtt.

Voigtländer considers macro tree transducers from a different point of view [VK04; Voi04]. Although modularization is a key advantage of functional languages, it is, on the other hand, one of the sources for unefficient programs. Solving a problem by composing functions that solve subproblems, means that each time the next function is called on the way to the complete solution, intermediate results have to be created. Viewing each macro tree transducer as a special kind of functional program, Voigtländer investigates certain syntactic restrictions of the composed transducers, such that it

is possible to eliminate intermediate results by constructing a single macro tree transducer that implements the composition of the original ones [VK04].

5

Type Checking TL Programs

As mentioned in the introduction, XML is a very popular data exchange format. Usually, XML documents are often generated automatically by some application, and are then processed by a another application or program. For a correct exchange of information it has to be guaranteed that for correct inputs only correct outputs are produced.

Consider, for example, an XML file of bibliographic entries functioning as database (e.g. [Ley02]). Users can search this database via a Web interface, which starts a query (or transformation) returning the search results. The query should always return a valid XHTML (or HTML) document, otherwise the result cannot be correctly displayed in the Web interface. Consequently, the query is correct only if it returns those parts of the document matching the user's search *and* the queried parts are formatted as an XML document of type XHTML.

Thus, the wish for *type-safe* XML transformations becomes the center of interest, where type-safe means that for every correct input document a transformation produces results, which meet the structural requirements of the output. In this context, static type checking XML transformations is of great benefit. On the one hand it guarantees that the transformation computes the transformation actually intended by the programmer, and on the other hand each output document conformes to a specified output type that is possibly input for a successive transformation.

In Chapter 3 we have presented the transformation language TL which subsumes the so-called tree transformation core of most of the popular transformation languages like XSLT [XSL99] or XQuery [BCF+06]. In order to illustrate the idea of type checking, we start with an example transformation.

The given document lists all incoming e-mails in the `Inbox`-element. Element `Trash` is meant to collect all deleted e-mails. Besides the normal `Mail`-elements, `Inbox` also contains e-mails inside a `Spam`-element. This indicates that these mails have been identified as spam, e.g., by some automated mail-filter.

Example 5.1

```
⟨MailDoc⟩
  ⟨Inbox⟩
    ⟨Mail⟩
      ⟨From⟩ hamlet@denmark.com ⟨/From⟩
      ⟨To⟩ horatio@denmark.com ⟨/To⟩
      ⟨Subject⟩⟨/Subject⟩
      ⟨Data⟩ ... ⟨/Data⟩⟨/Mail⟩
    ⟨Spam⟩
      ⟨Mail⟩
        ... ⟨Subject⟩ V.I.A.G.R.A. ⟨/Subject⟩
        ⟨Data⟩ ... ⟨/Data⟩
      ⟨/Mail⟩⟨/Spam⟩
  ⟨/Inbox⟩
  ⟨Trash⟩ ... ⟨/Trash⟩
⟨/MailDoc⟩
```

In order to "clean up" the `Inbox`-element, all e-mails marked as spam should be moved into the `Trash`-element, while all other elements in the `Inbox` should be left untouched. Thus, a transformation should return:

Example 5.1 (continued)

```
⟨MailDoc⟩
  ⟨Inbox⟩
    ⟨Mail⟩
      ⟨From⟩ hamlet@denmark.com ⟨/From⟩
      ⟨To⟩ horatio@denmark.com ⟨/To⟩
      ⟨Subject⟩⟨/Subject⟩
      ⟨Data⟩ ... ⟨/Data⟩⟨/Mail⟩
  ⟨/Inbox⟩
  ⟨Trash⟩
    ⟨Spam⟩
      ⟨Mail⟩
        ... ⟨Subject⟩ V.I.A.G.R.A. ⟨/Subject⟩
        ⟨Data⟩ ... ⟨/Data⟩
      ⟨/Mail⟩⟨/Spam⟩
  ... ⟨/Trash⟩
⟨/MailDoc⟩
```

The transformation explained above can be implemented with the following TL program

```
1  q0(label_MailDoc(x1)) ⟶ ⟨MailDoc⟩
2                             q0(label_Inbox(x2))
3                             q0(label_Trash(x2))
4                           ⟨/MailDoc⟩
5  q0(label_Inbox(x1))   ⟶ ⟨Inbox⟩
6                             q2(x1/x2 ∧ label_Mail(x2))
7                           ⟨/Inbox⟩
8  q0(label_Trash(x1))   ⟶ ⟨Trash⟩
9                             q2(∃ z. z; x1 ∧ z/x2 ∧ label_Spam(x2))
10                            q2(x1/x2)
11                          ⟨/Trash⟩
12 q2(x1)                ⟶ x1
```

The first rule for q_0 (cf. line 1) is applicable to the `MailDoc`-element of the input. Its action specifies to produce an `MailDoc`-element in the output whose content is obtained by recursively applying q_0 to the nodes specified by the select patterns $\mathsf{label}_{\texttt{Inbox}}(x_2)$ and $\mathsf{label}_{\texttt{Trash}}(x_2)$, respectively. Since these patterns do not contain variable x_1, the nodes to which they refer are determined *absolutely* in the input tree, i.e., from the root node. The second rule of q_0 is applicable if the current node is an `Inbox`-element. Its action specifies to generate a new `Inbox` in the output. The `Inbox`'s content is obtained by applying q_2 to the nodes selected by the pattern $x_1/x_2 \wedge \mathsf{label}_{\texttt{Mail}}(x_2)$ (line 6), i.e., the children x_2 of the current node x_1 which are labeled with `Mail`. Since the current node is the `Inbox`-element of the input, this rule copies all e-mails from the folder for incoming mails into the `Inbox`-element of the output — except the `Spam`-elements. The third q_0-rule transformes the `Trash`-element. The first call to function q_2 is meant to copy all `Spam`-elements from `Inbox` into `Trash`. The necessary select pattern is given by the formula $\exists\, z.\, z; x_1 \wedge z/x_2 \wedge \mathsf{label}_{\texttt{Spam}}(x_2)$. This formula selects all nodes x_2 labeled with `Spam` whose ancestor z is a left sibling of the current node x_1 (line 9). For completing our transformation, we finally have to copy all elements x_2 of the `Trash` folder x_1 into the `Trash`-element of the output.

Although the attentive and gentle reader might be convinced that this TL program realizes the "simple" transformation we explained above, we want to have a *general facility* testing whether a given program implements the intended transformation. Consider, for example, an XML document containing information about the clients of a company. The different departments of the company need different pieces of information in order to transact business. The software develpors, for example, need the specifications of the ordered programs; the accounting department needs only a detailed statement of costs. All these information pieces are automatically extracted from the same XML document. This means, it is necessary to guarantee that the automatic extraction returns the correct data.

A prerequisite for such a test is an exact specification of the transformation. One possibility to define the intended transformation is to specify the input and output type.

Our problem can then be reduced to the question whether the program is correct with respect to the given types. This process is called *type checking*. Recall that for a given transformation F and an input and output type, T_{in} and T_{out}, respectively, *F type checks* with respect to T_{in} and T_{out} implies, that

$$\forall d \in T_{\text{in}} : \quad F(d) \in T_{\text{out}}.$$

This means, for all documents d conforming to the input type, the transformation result $F(d)$ is valid with respect to the output type T_{out}.

In this chapter we describe how to type check TL programs. The key idea is to convert an arbitrary TL program into a model for which type checking is decidable. This model is the stay macro tree transducer because first, type checking of compositions of stay-mtts is decidable, and second, since each stay-mtt can be viewed as a restricted form of a TL program, the conversion is tractable and has to concentrate only on those features that go beyond the stay-mtt.

Let $Prog = (R, A_0)$ a TL program as defined in Chapter 3.1 (cf. Definition 3.2). The decomposition of *Prog* into stay-mtts, which realize the transformation induced by *Prog*, is carried out in several steps.

(1) In a first step (Section 5.1) the input is decorated so that the simulating transducer can decide at each node whether a pattern matches or not. For this decoration, we use a bottom-up followed by a top-down relabeling.
(2) Then we simulate program *Prog* by a tree-walking transducer T_{walk} (Section 5.2). The key point of this step is to compile the global movements, inherent in the select patterns, into local movements. T_{walk} works as follows: if a TL expression $q(\psi, S_1, \ldots, S_k)$ is evaluated for a node v, T_{walk} successively proceeds from v to the root node of the input, where it returns a (representation of a) list of the results obtained by recursively applying the function q to all nodes u where the pair (v, u) satisfies the pattern ψ.
(3) In the last step we have to eliminate all up-moves in the input tree and obtain a composition of stay macro tree transducers simulating *Prog*'s transformation (Section 5.3).

5.1 Annotating the Input

In order to annotate the input with informations about the matching patterns, we first compile all match and select patterns of the TL program into a single automaton. Since we want to decompose TL programs into stay macro tree transducers, we have to take into account that stay-mtts work on ranked

trees. Thus, the input forest has to be represented as binary tree (cf. Section 2.1).

Let Φ and Ψ be the finite sets of MSO match and select patterns occurring in our TL program *Prog*, 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.

Using the well-known translation of MSO formulas over finite ranked trees by Thatcher and Wright [TW68], we can construct a nondeterministic tree automaton

$$Match = (P, \Sigma, F, \delta).$$

Together with this automaton, we define sets $U_\phi \subseteq P, \phi \in \Phi$, and relations $B_\psi \subseteq P^2, \psi \in \Psi$, such that for every document forest f the following holds:

(1) For every $\phi \in \Phi$, $(v) \models_f \phi$ if and only if there exists an accepting run *run* of *Match* on the binary encoding of f such that $run(v) \in U_\phi$, and
(2) for every $\psi \in \Psi$, $(v, u) \models_f \psi$ if and only if there exists an accepting run *run* of *Match* on the binary encoding of f such that $(run(v), run(u)) \in B_\psi$.

While the sets U_ϕ contain those states which indicate a match of a match pattern ϕ in a left-hand side of a TL rule, the relations B_ψ contain all pairs of states indicating a match of a select pattern ψ occurring in a recursive function call in a rule.

For the binary tree representation t of a forest, let $S(t) \subseteq P$ denote the set of states p in which the match automaton $Match = (P, \Sigma, F, \delta)$ (defined on page 67) may possibly reach the root node of t. The set $S(t)$ is recursively defined by

$$\begin{aligned} S(\epsilon) &= \{p \in P \mid (p, \epsilon) \in \delta\} \\ S(\mathtt{a}(t_1, t_2)) &= \{p \in P \mid \exists p_i \in S(t_i) : (p, \mathtt{a}, p_1 p_2) \in \delta\}. \end{aligned}$$

(1) First, each node $v \in \mathcal{N}(t)$ of the input tree t labeled with a symbol $\mathtt{a} \in \Sigma$ is relabeled by

$$\langle \mathtt{a}, S_1, S_2 \rangle$$

where S_1, S_2 is the set of states reachable at the subtree rooted at the first and second child of v, respectively. According to the recursive definition of the sets $S()$, this relabeling can be done by means of a single deterministic post-order traversal over t. This means it can be done by a total deterministic bottom-up relabeling

$$Bot = (B_{Bot}, \Sigma, F_{Bot}, R_{Bot}),$$

where $B_{Bot} = 2^S$, Σ is the alphabet of the match automaton *Match*, and F_{Bot} is the set of final states which coincides with the set of accepting states of *Match*. The rules in R_{Bot} are

$$\begin{aligned} \epsilon &\to b_{S'}(\langle \epsilon \rangle) \\ \mathtt{a}(b_{S_1}(x_1), b_{S_2}(x_2)) &\to b_{S''}(\langle \mathtt{a}, S_1, S_2 \rangle(x_1, x_2)), \end{aligned}$$

where the sets S' and S'' are defined as follows

$$\begin{aligned} S' &= S(\epsilon) \\ S'' &= \{s \in S \mid (s, \mathtt{a}, s_1 s_2) \in \delta, s_i \in S_i, i = 1, 2\}. \end{aligned}$$

A formal definition of bottom-up relabelings and a comparison with top-down finite state tree transformations can be found in [Eng75].

(2) Additionally, each node $v \in \mathcal{N}(t)$ receives the value 0 if v is the root node 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

$$T(v) = \{run(v) \mid run \text{ accepting run of } Match \text{ for } t\},$$

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 $\mathtt{a}$ have labels

$$\langle \mathtt{a}, j, T(v), S(v_1), S(v_2) \rangle,$$

whereas leaf nodes u have labels

$$\langle \mathtt{a}, j, T(v) \rangle.$$

After having equipped every node of t with the according sets of reachable states as described above, this second relabeling can be implemented by means of a single deterministic pre-order traversal over the annotated input tree t.

This relabeling can be done by a total deterministic top-down relabeling which corresponds to a top-down tree transducer whose rules are of the general form $q(\mathtt{a}(x_1, \ldots, x_k)) \to \mathtt{b}(q(x_1), \ldots, q(x_k))$ [EM99]. In particular, the relabeling is implemented by the transducer

$$Top = (B_{Top}, \Sigma', b_{0,F}, R_{Top}),$$

where the function set $B_{Top} = \{0, 1, 2\} \times 2^S$ contains all pairs consisting of a child number and a subset of states. Σ' denotes the modified labels of the form $\langle \mathtt{a}, S_1, S_2 \rangle$ for all symbols $\mathtt{a} \in \Sigma$ and sets $S_1, S_2 \in S$. The initial function is $b_{0,F}$ and the rules in R_{Top} are

$$b_{j,T_0}(\langle \mathtt{a}, S_1, S_2 \rangle(x_1, x_2)) \to \langle \mathtt{a}, j, T, S_1, S_2 \rangle(b_{1,T_1}(x_1), b_{2,T_2}(x_2)),$$

where the sets T and $T_i, i = 1, 2$, are defined as

$$\begin{aligned} T &= \{p \in T_0 \mid (p, \mathtt{a}, p_1 p_2) \in \delta \text{ and } p_i \in S_i\} \\ T_i &= \{p_i \in S_i \mid (p, \mathtt{a}, p_1 p_2) \in \delta \text{ and } p \in T,\ p_{3-i} \in S_{3-i}\}. \end{aligned}$$

Moreover, for leaf nodes the following rules are in R_{Top}

$$b_{j,T_0}(\epsilon) \to \langle \epsilon, j, T \rangle,$$

where $T = \{p \in T_0 \mid (p, \epsilon) \in \delta\}$.

Note that our bottom-up relabeling *Bot* followed by the top-down relabeling *Top* can jointly be considered as top-down relabeling with regular look-ahead, because the set of deterministic bottom-up relabelings is a subset of the class of deterministic top-down transducers with look-ahead (Theorem 3.2 of [Eng77]) and this class is closed under composition with deterministic top-down relabelings (Lemma 2.10(2) of [Eng77]). Our complete relabeling is also equivalent to an MSO definable relabeling [EM99; BE00]. Let REL denote the class of all MSO definable relabelings.

5.2 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 idea is to compile the global selection steps into local movements. This means, instead of defining a binary relation between nodes of the input with an MSO formula, we now formalize a collection of functions traversing the input node by node and collecting the nodes which have to be processed next.

As an intermediate model, we define a generalization of macro tree transducers (defined on page 34) called *macro tree-walking transducer*.

This model generalizes the stay-mtts by allowing to continue the transformation at a child or the father of the current node.

Definition 5.2 (Macro tree-walking transducer) A *macro tree-walking transducer* (2-mtt) is a pair (R, A_0) where A_0 is a set of initial actions and R is a finite set of rules of the form

$$q(\mathsf{label}_{\mathtt{a}}(x_0), y_1, \ldots, y_k) \longrightarrow A,$$

where q is a function name and a is a label of a node in the input. Possible actions (including the initial actions from A_0) are described by the grammar:

$$A ::= \mathtt{b}(A) \mid x_0 \mid y_j \mid q'(\psi, A_1, \ldots, A_m),$$

where b is the label of a node in the output, y_j is one of the accumulating parameters, q' is a function name, and ψ is a pattern which is either $x_0 = x_1$, or $x_1 = \mathsf{father}(x_0)$, or $x_1 = \mathsf{child}_i(x_0)$.

Intuitively, a 2-mtt is similar to a TL program, but operates on ranked trees instead of forests. Moreover, match patterns in the left-hand side of the rules are restricted to the form $\mathsf{label}_{\mathtt{a}}(x_0)$, i.e., they are only allowed to check the label of the current input node x_0. The select patterns in recursive function calls may only be of one of the following forms:

- $x_0 = x_1$, i.e., the current node x_0 itself is selected;
- $x_1 = \mathsf{father}(x_0)$ selects the father of x_0;

- $x_1 = \mathsf{child}_i(x_0)$, which selects the i-th child of the current node x_0 – in our binary tree representation of forests, this pattern replaces the ";" and "/" predicats.

Note that each of these patterns selects at most one next node v', given any binding of variable x_0 to a node v of the input tree.

If we restrict the select patterns to $x_0 = x_1$ and $x_1 = \mathsf{child}_i(x_0)$ and furthermore do not allow the copy action x_0 in right-hand sides, then we get the concept of stay macro tree transducers (cf. Definition 4.1 on page 34).

The key idea of the simulation of the TL program *Prog* by a macro tree-walking transducer 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 with the binary select pattern ψ. This means function q has to be applied to all nodes $v_1 < \ldots < v_n$ where the sequence of pairs $(v, v_1), \ldots, (v, v_n)$ represents all matches in document order of the pattern ψ in f. In order to collect the forests produced by

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

for all nodes $v_i, i = 1, \ldots, n$, our simulating 2-mtt successivly proceeds from the node v to the root node r of the annotated input where it returns a representation of the list

$$[\![q]\!]_f(v_1, [\![A_1]\!]_f, \ldots, [\![A_k]\!]_f) \cdots [\![q]\!]_f(v_n, [\![A_1]\!]_f, \ldots, [\![A_k]\!]_f).$$

These applications to the nodes v_i are collected on the path from v to the root node by traversing the prevailing left or right siblings, depending on the direction from where the simulation has moved up.

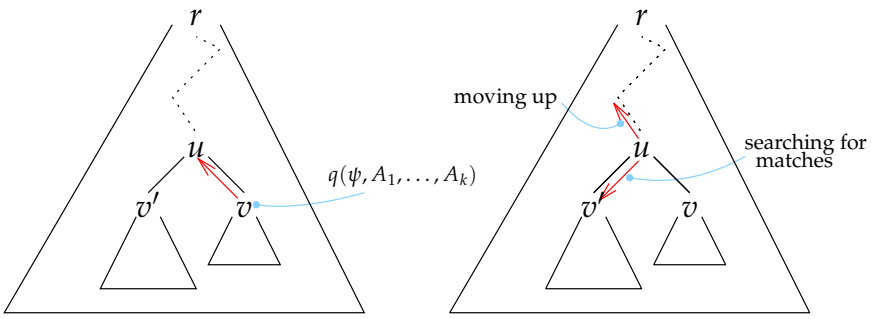

Fig. 5.1. Simulating global selection. In order to simulate the call of TL function q at node v, one traverses the subtree of v, moves up to the father u of v, and searches from u the left sibling v' of v for nodes matching ψ, and so on until reaching the root node.

Figure 5.1 illustrates this behaviour. At node v function $q(\psi, A_1, \ldots, A_k)$ is called. In order to find all nodes $v_1, \ldots, v_n$ building together with v a match

for pattern ψ, we systematically have to traverse the complete input tree. Therefore, we first scan the subtree rooted at node v for possible matches. Then we move up to the father u of v and have to search sibling v' of v for possible matches and at the same time we have to test whether u itself is a possible match. From node u we continue this traversal until reaching the root r of the input tree and have searched the complete input for matching nodes.

For the up movements, we therefore introduce the family of functions up, whereas for the collection of matches to the left and right of the current path to the root, we introduce the functions down.

In particular, we construct for the TL program $Prog = (R, A_0)$ the rule set of the macro tree-walking transducer T_{walk} as follows. Each TL rule

$$q(\psi, y_1, \ldots, y_k) \longrightarrow A$$

is simulated by the rules

$$\begin{aligned} q(\mathsf{label}_{\langle \mathtt{a}, j, T, S_1, S_2 \rangle}(x_1), y_1, \ldots, y_k) &\to [A]_j \\ q(\mathsf{label}_{\langle \epsilon, j, T \rangle}(x_1), y_1, \ldots, y_k) &\to [A]_j \end{aligned}$$

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

$$\begin{aligned} [\epsilon]_j &= \epsilon \\ [y_i]_j &= y_i \\ [x_1]_j &= x_1 \\ [\langle \mathtt{b} \rangle A \langle / \mathtt{b} \rangle]_j &= \mathtt{b}([A]_j, \epsilon) \\ [A_1\ A_2]_j &= [A_1]_j @ [A_2]_j. \end{aligned}$$

Since 2-mtts operate on ranked trees, they do not support concatenation of forests as a base operation. Therefore, we express concatenation symbolically by means of the new binary constructor "@"[1]. The occurrences of this symbolic operator will be evaluated in a subsequent transformation.

In order to simulate recursive function calls $q'(\psi, A_1, \ldots, A_k)$ occurring in 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 the pair (r, r) is a match for select pattern ψ, i.e., r is selected by ψ, then the 2-mtt transforms r with function q and concatenates the results of calling down for the child nodes of r,

$$\begin{aligned} [q'(\psi, A_1, \ldots, A_k)]_0 = \; & q'(x_1 = x_2, [A_1]_0, \ldots, [A_k]_0) @ \\ & \mathsf{down}^{q'}_{\psi, g_1}(x_2 = \mathsf{child}_1(x_1), [A_1]_0, \ldots, [A_k]_0) @ \\ & \mathsf{down}^{q'}_{\psi, g_2}(x_2 = \mathsf{child}_2(x_1), [A_1]_0, \ldots, [A_k]_0). \end{aligned}$$

[1] For better readability this constructor is written infix, i.e., $t_1 @ t_2$ stands for the binary tree $@(t_1, t_2)$ and denotes the concatenation of the trees t_1 and t_2.

Note that the function down records the name q' of the called function together with the select pattern ψ and a relation $g_i, i = 1, 2$, on the nodes which is explained below. Note also that the current node is selected by the pattern ψ if and only if T contains a state p such that $(p, p) \in B_\psi$. If the current node is *not* selected by ψ, then we omit the transformation $q'(x_1 = x_2, [A_1]_0, \ldots, [A_k]_0)$ of the current node, thus we simulate the call by

$$\begin{aligned}[q'(\psi, A_1, \ldots, A_k)]_0 = \; & \mathsf{down}^{q'}_{\psi,g_1}(x_2 = \mathsf{child}_1(x_1), [A_1]_0, \ldots, [A_k]_0)@ \\ & \mathsf{down}^{q'}_{\psi,g_2}(x_2 = \mathsf{child}_2(x_1), [A_1]_0, \ldots, [A_k]_0).\end{aligned}$$

If on the other hand the function call $q'(\psi, A_1, \ldots, A_k)$ occurs at a node v different from the root node of the input, i.e., $j \neq 0$, we have to search the current subtree for possible matches and we have to continue our search at the father of v. If v is selected by pattern ψ, then the subsequent call for q' is simulated by

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

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 and have

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

Analogue to the case where q' is called at the root node, we compute the list of outputs for the matches located in the subtree rooted at the current node at that node. Now, however, we pass a tree representation of this forest in an accumulating parameter to the function up. This function is meant to proceed upwards to the root to collect also the transformations of the remaining nodes selected by pattern ψ. Similar to down, function 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 a mapping h.

The extra information h and g_i ($i = 1, 2$) for the functions up and down, respectively, are relations on the states. They 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 *Match* (cf. page 67)

$$\begin{aligned}h &= \{(p, p) \mid p \in T\} \\ g_i &= \{(p, p_i) \in T \times S_i \mid \exists p_{3-i} \in S_{3-i} : (p, \mathsf{a}, p_1 p_2) \in \delta\}.\end{aligned}$$

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 function

$$\mathsf{down}^{q'}_{\psi,g}$$

is meant to traverse the subtree at the current node w and to return (a representation of) the list of results obtained by applying the function q' to all nodes $v_1, \ldots, v_m$ in document order which are selected by pattern ψ. In other words, down concatenates the values of q' for all selected nodes in the subtree rooted at w. Thus, it can recursively be defined by the rules

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

with

$$g'_i = \{(p, p_i) \in P \times S_i \mid \exists (p, p') \in h, p_{3-i} \in S_{3-i} : (p', \mathsf{a}, p_1 p_2 \in \delta\}.$$

These rules deal with the case where the current node is a match of ψ, 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.

For the up moving function on the other hand, a call

$$\mathsf{up}^{q,j}_{\psi,h}$$

denotes the function up scanning the input on its way to the root for matches of the select pattern ψ, and each match is transformed with function q. The number $j \in \{1, 2\}$ indicates that the current node w was reached coming from its j-th child. The function up moves from the current node to its father, and then, depending whether it came from the left or right subtree (recorded in the number j) it calls function down on the right or left subtree, respectively. For the case that the current node is selected by the pattern ψ, i.e., $h_j \cap B_\psi \neq \emptyset$ if up is called from the j-th child, we define for the case that the root node of the input is reached from the right child

$$\begin{aligned}
&\mathsf{up}^{q',1}_{\psi,h}(\mathsf{label}_{\langle \mathsf{a},0,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k, y_{k+1}) \quad \rightarrow \\
&\qquad q'(x_1 = x_2, y_1, \ldots, y_k)@ \\
&\qquad y_{k+1}@ \\
&\qquad \mathsf{down}^{q'}_{\psi,g_1}(x_2 = \mathsf{child}_2(x_1), y_1, \ldots, y_k)
\end{aligned}$$

and for all other nodes reached from the right child

$$\begin{aligned}
&\mathsf{up}^{q',1}_{\psi,h}(\mathsf{label}_{\langle \mathtt{a},j,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k, y_{k+1}) \quad \rightarrow \\
&\qquad \mathsf{up}^{q',j}_{\psi,h_1}(\ x_2 = \mathsf{father}(x_1), y_1, \ldots, y_k, \\
&\qquad\qquad q'(x_1 = x_2, y_1, \ldots, y_k)@ \\
&\qquad\qquad y_{k+1}@ \\
&\qquad\qquad \mathsf{down}^{q'}_{\psi,g_1}(x_2 = \mathsf{child}_2(x_1), y_1, \ldots, y_k)).
\end{aligned}$$

The definitions for reaching a node from the left child are defined accordingly

$$\begin{aligned}
&\mathsf{up}^{q',2}_{\psi,h}(\mathsf{label}_{\langle \mathtt{a},0,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k, y_{k+1}) \quad \rightarrow \\
&\qquad q'(x_1 = x_2, y_1, \ldots, y_k)@ \\
&\qquad \mathsf{down}^{q'}_{\psi,g_2}(x_2 = \mathsf{child}_1(x_1), y_1, \ldots, y_k)@ \\
&\qquad y_{k+1} \\
&\mathsf{up}^{q',2}_{\psi,h}(\mathsf{label}_{\langle \mathtt{a},j,T,S_1,S_2\rangle}(x_1), y_1, \ldots, y_k, y_{k+1}) \quad \rightarrow \\
&\qquad \mathsf{up}^{q',j}_{\psi,h_1}(\ x_2 = \mathsf{father}(x_1), y_1, \ldots, y_k, \\
&\qquad\qquad q'(x_1 = x_2, y_1, \ldots, y_k)@ \\
&\qquad\qquad \mathsf{down}^{q'}_{\psi,g_2}(x_2 = \mathsf{child}_1(x_1), y_1, \ldots, y_k)@ \\
&\qquad\qquad y_{k+1}).
\end{aligned}$$

Observe that the recursive calls to up receive modified values h_j. Moreover, we have to generate new relations g_j for the subsequent calls of down traversing the up to now *un*-scanned subtrees. The new values h_i and g_i $(i = 1, 2)$ are defined by

$$\begin{aligned}
h_i &= \{(p, p') \in P \times T \mid \exists (p, p_i) \in h, p_{3-i} \in S_{3-i} : (p', \mathtt{a}, p_1 p_2) \in \delta\} \\
g_i &= \{(p, p_{3-i}) \in P \times S_{3-i} \mid \exists p' \in T, (p, p_i) \in h : (p', \mathtt{a}, p_1 p_2) \in \delta\}.
\end{aligned}$$

In this section we compiled the TL program *Prog* into a 2-mtt T_{walk} operating on suitable annotated tree representations of forests. The 2-mtt, however, does not compute the (tree representations of the) output forests of *Prog* directly. Instead, it generates occurrences of the symbolic concatenation operator @ whenever needed. Summing up, we have the following result:

Theorem 5.3 For every TL program P there is a composition

$$A \circ \tau_{P'} \circ R$$

which computes the binary tree encoding of the transformation realized by P. $A \in \mathsf{APP}$ is the mapping which evaluates all occurrences of @ in binary trees, P' is a 2-mtt, and R is a relabeling in REL. If P is deterministic, then the 2-mtt P' can be chosen to be deterministic, too.

5.3 Removing Up Moves

In the previous section we constructed from a TL program P a macro tree-walking transducer. In this section we want to change this constructed transducer in such a way that the new transducer processes the input in a strict *top-down* fashion. This means, we aim at simulating a TL program with stay macro tree transducers (cf. Chapter 4).

Since stay-mtts cannot move upwards in their input, we need an approporiate method to translate all moves to ancestors performed by the 2-mtt. The key idea is, instead of moving up to a node u, we generate all possible function calls at u and pass them in the parameter positions to any further calls. In this way, a call of function q for the father node can be simulated by selecting the corresponding parameter y_q. Clearly, this simulation idea increases the number of parameters dramatically, because each possible function call is simulated in an extra parameter, i.e., for functions $q_1, \ldots, q_n$ of the 2-mtt we now need n functions each with n accumulating parameters.

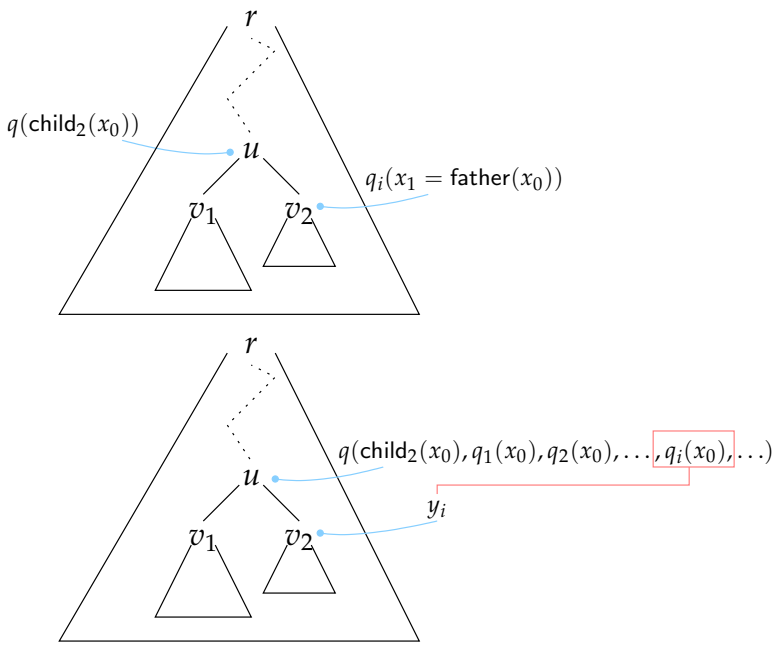

Fig. 5.2. Removing up moves. The upper tree shows the situation for the 2-mtt: at node u function q is applied to node v_2, and at v_2 function q_i is applied to its father u. The second tree shows the simulation idea: Now, function q records in the parameters each possible transformation result of the current node x_0. Then, the transformation of the father node can be replaced by looking up the appropriate parameter.

This simulation, however, is not able to deal properly with accumulating parameters of the 2-mtt which are modified during up moves. Therefore, we first make use of the following decomposition idea: we decompose the 2-mtt P into a 2-mtt P' *without* accumulating parameters followed by some special transformation Y (which we explain later). The 2-mtt P' executes only the calling behaviour of P, while performing parameter passing symbolically by means of formal substitution symbols. A symbol α_i represents the usage of formal parameter y_i and a symbol σ_m represents the substitution of m actual parameters. Thus, the following small 2-mtt with parameters

$$\begin{aligned}
&q(\mathsf{label}_{\mathtt{a}}(x_0), y_1, y_2) \to q(x_1 = \mathsf{child}_1(x_0), \mathtt{b}(y_1), y_2)\\
&q(\mathsf{label}_{\mathtt{r}}(x_0), y_1, y_2) \to y_1\\
&q(\mathsf{label}_{\mathtt{1}}(x_0), y_1, y_2) \to y_2
\end{aligned}$$

would be rewritten as

$$\begin{aligned}
&q(\mathsf{label}_{\mathtt{a}}(x_0)) \to \sigma_2(q(x_1 = \mathsf{child}_1(x_0)), \mathtt{b}(\alpha_1), \alpha_2)\\
&q(\mathsf{label}_{\mathtt{r}}(x_0)) \to \alpha_1\\
&q(\mathsf{label}_{\mathtt{1}}(x_0)) \to \alpha_2
\end{aligned}$$

Such a transducer without parameters now produces trees containing the auxiliary occurrences of the symbols α_i and σ_m.

The symbolic substitution can be evaluated by a *yield* transformation. For an extended alphabet $\Sigma' = \Sigma \cup \{\alpha_i \mid 1 \le i \le m\} \cup \{\sigma_i \mid 1 \le i \le m\}$, this transformation can be implemented by a deterministic macro tree transducer $\mathit{Yield} = (Q, \Sigma', \mathit{yield}_0, R_{\mathit{Yield}})$ with

$$Q = \{\mathit{yield}_i \mid 1 \le i \le m\}.$$

For $k \ge 0$ function yield_k has k extra arguments and is defined as

$$\begin{aligned}
&\mathit{yield}_k(\alpha_j, y_1, \ldots, y_k) && \to y_j, \quad \text{if } j \le k\\
&\mathit{yield}_k(\sigma_n(x_1, \ldots, x_{n+1}), y_1, \ldots, y_k) && \to \mathit{yield}_n(x_1, \mathit{yield}_k(x_2, y_1, \ldots, y_k), \ldots,\\
&&& \qquad \mathit{yield}_k(x_{n+1}, y_1, \ldots, y_k))\\
&\mathit{yield}_k(\mathtt{b}, y_1, \ldots, y_k) && \to \mathtt{b}\\
&\mathit{yield}_k(\mathtt{a}(x_1, x_2), y_1, \ldots, y_k) && \to \mathtt{a}(\mathit{yield}_k(x_1, y_1, \ldots, y_k),\\
&&& \qquad \mathit{yield}_k(x_2, y_1, \ldots, y_k)),
\end{aligned}$$

where a and b are symbols from Σ. Let YIELD denote the class of all transductions realizing the symbolic substitution and parameter passing.

In the following results we consider without loss of generality only 2-mtts working on binary tree representations of forests, i.e., the working alphabet has one symbol "ϵ" of rank 0 denoting the empty forest, and all other symbols are of rank 2.

We first consider deterministic tree-walking transducers and show that they can be decomposed into a deterministic macro tree transducer followed by a yield-function for evaluating the symbolic substitution. This decomposition is performed in two steps:

(1) The 2-mtt is decomposed into another 2-mtt without accumulating parameters followed by a yield-function.
(2) The 2-mtt without accumulating parameters then can be translated into a deterministic macro tree transducer.

Lemma 5.4 For every deterministic 2-mtt P there exists a 2-mtt P' without parameters such that

$$\tau_P = Y \circ \tau_{P'},$$

where Y is a transformation of the class YIELD and P' again is deterministic.

Proof. Let $P = (\Sigma, R, A_0)$ be a deterministic 2-mtt. Moreover, let m be the maximal number of accumulating parameters of a function of P. Then we construct a deterministic 2-mtt $P' = (\Sigma', R', A_0')$. The new alphabet is defined as

$$\Sigma' = \Sigma \cup \{\alpha_i \mid 1 \leq i \leq m\} \cup \{\sigma_i \mid 1 \leq i \leq m\} \cup \{\bot\}$$

where $\bot$ is a new symbol denoting *undefined*. The initial rule is $A_0' = S(A_0)$ and the new rule set is obtained from R by rewriting each rule

$$q(\mathsf{label}_\mathtt{a}(x_0), y_1, \ldots, y_k) \to A \quad \text{as} \quad q(\mathsf{label}_\mathtt{a}(x_0)) \to S(A)$$

where the new actions are obtained by

$$\begin{array}{ll} S(y_i) & = \alpha_i \\ S(\mathtt{b}(A_1, \ldots, A_k)) & = \mathtt{b}(S(A_1), \ldots, S(A_k)) \\ S(q'(\psi, A_1, \ldots, A_l)) & = \sigma_l(q'(\psi), S(A_1), \ldots, S(A_l)). \end{array}$$

Furthermore, we add for all pairs $q, \mathtt{b}$ for which P does not have a rule with left-hand side $q(\mathsf{label}_\mathtt{b}(x_0), \ldots)$ the new rule

$$q(\mathsf{label}_\mathtt{b}(x_0)) \to \bot\ .$$

These rules avoid that P' reaches a node v with a function q for which no rule is applicable – or even simpler, the rules avoid that P' gets "stuck".

The transformation Y is implemented by an yield transducer as described above. Note that none of the functions $yield_k$ is defined for the input symbol $\bot$. □

The following result has already been proved in [EV86]. We repeat the construction here, because the provided decomposition is necessary for understanding the complete simulation of TL programs by at most three stay macro tree transducers.

Lemma 5.5 The transformation of a deterministic 2-mtt P can be effectively decomposed into

$$Y \circ \tau_M,$$

where M is a total and deterministic mtt, Y is a transformation of the class YIELD.

Proof. We first decompose P according to Lemma 5.4 into a 2-mtt P' without parameters and a macro tree transducer *Yield*. P' can now be transformed into a total and deterministic mtt $M = (Q, \Sigma, q_0, R)$.

Our construction depends on the information whether a node is the root node or not. Since mtts are closed under relabelings, we relabel the input and keep in mind that this additional transfromation step can be absorbed into M [EV85]. It is sufficient to use a deterministic mtt with the initial function b_0 and rules of the form

$$b_j(\mathtt{a}(x_1, x_2)) \to \langle \mathtt{a}, j\rangle(b_1(x_1), b_2(x_2))$$

for all occurring symbols a.

Assume that P' has functions $q_1, \ldots, q_n$. For each of these functions, M has two variants. One without parameters which is used if the current node is the root node, because no up moves can be performed and thus no parameters are necessary. The second variant is used at every other node and has n accumulating parameters to store all possible function values for the current node. This means

$$Q = \{\overline{q_\nu} \mid rank(\overline{q_\nu}) = 0, \nu = 1, \ldots, n\} \cup \{q_\nu \mid rank(q_\nu) = n+1, \nu = 1, \ldots, n\}.$$

Based on this set of functions, we replace each rule

$$q_\nu(\mathsf{label}_d(x_0)) \to t$$

with $d = \langle \mathtt{a}, j\rangle$ a symbol of the annotated input, by the new rules

$$\begin{aligned} &\overline{q_\nu}(d(x_1, x_2)) = U_0(t), \text{ if } j = 0\\ &q_\nu(d(x_1, x_2)) = U(t), \quad \text{for } j \neq 0. \end{aligned}$$

The right-hand side t is rewritten with

$$\begin{aligned} U(d'(A_1, \ldots, A_k)) &= d'(U(A_1), \ldots, U(A_k))\\ U(q_\mu(x_1 = \mathsf{father}(x_0))) &= y_\mu\\ U(q_\mu(x_1 = x_0)) &= U(rhs(q_\mu, \mathsf{label}_d(x_0)))\\ U(q_\mu(x_1 = \mathsf{child}_j(x_0))) &= q_\mu(z_j, U(rhs(q_1, \mathsf{label}_d(x_0))), \ldots,\\ &\qquad U(rhs(q_n, \mathsf{label}_d(x_0)))), \end{aligned}$$

where $j \in \{1, 2\}$ and $rhs(q_i, \mathsf{label}_d(x_0)), i = 1, \ldots, n$, is the right-hand side of the q_i rule of P' with match pattern $\mathsf{label}_d(x_0)$.

The rules for leaf nodes with labels $\langle \epsilon, j\rangle$ are rewritten analogously, with the only difference that no accumulating parameters occur in the corresponding right-hand sides.

When transforming a single right-hand side with $U(.)$, it may recursively start to evaluate other right-hand sides for parameter positions. Since 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 nonterminating computations, we use the auxiliary symbol $\perp$ denoting *undefined*. Thus, we replace the $(n+1)$-th recursive application of $U(.)$ by $\perp$. $\square$

Switching to nondeterministic tree-walking transducers, we carefully have to distinguish between outside-in and inside-out evaluation. Recall that the two evaluation modes differ in the order of evaluating nested function calls. Outside-in transducers compute the outermost function call first, thus parameters may contain unevaluated functions. Inside-out transducers, on the other hand, evaluate the innermost function call first (cf. [ES77; ES78; EV85]).

Therefore, we first explain the decomposition for outside-in 2-mtts and afterwards for inside-out 2-mtts. The idea is the same as for deterministic transformations: the tree-walking transducer is simulated by a 2-mtt without accumulating parameters and a yield function for evaluating the symbolic substitution. Then the constructed 2-mtt without parameters is translated into a stay macro tree transducer.

Lemma 5.6 The transformation of a nondeterministic OI 2-mtt P can be effectively decomposed into

$$Y \circ \tau_M,$$

where M is a nondeterministic OI stay-mtt and Y is a transfromation of the class YIELD_c.

Proof. As in Lemma 5.4 for deterministic 2-mtts, we first decompose P into a 2-mtt P' without accumulating parameters followed by a function for evaluation of symbolic substitution. Afterwards we simulate P' with an OI stay-mtt M.

In order to construct P', we use a decomposition idea which has already been provided by Engelfriet and Vogler in [EV86]. The idea is as follows. First, we force P to be deterministic by collecting all rules

$$q(\mathsf{label}_a(x_0), y_1, \ldots, y_k) \to t_i, \quad \text{for } i = 1, \ldots, r$$

for function q with the same left-hand side, and combining all right-hand sides into a single rule

$$q(\mathsf{label}_a(x_0), y_1, \ldots, y_k) \to \$(t_1, \$(\ldots, \$(t_{r-1}, t_r)\ldots)),$$

where "\$" is a new binary operator symbol denoting binary choice. The resulting transducer is obviously deterministic. We then apply the transformation $S(.)$ from Lemma 5.4 to the initial action as well as to the new rules — if there were more than one initial rules, they would have been combined in the same way. Additionally and in order to allow termination of the function call at any moment, we add for every pair q, a

$$q(\mathsf{label}_a(x_0)) \to \perp$$

as a second rule. Intuitively, the resulting transducer P' does not only delay substitution into formal variables but also the choice between different alternatives for a given function call. The substitution and choice operators are evaluated by a transformation Y which can be implemented by an stay-mtt *Yield* similar to that of Lemma 5.4 but with additional rules

$$\begin{aligned} &yield_k(\$(x_1, x_2), y_1, \ldots, y_k) \to yield_k(x_1, y_1, \ldots, y_k) \\ &yield_k(\$(x_1, x_2), y_1, \ldots, y_k) \to yield_k(x_2, y_1, \ldots, y_k) \end{aligned}$$

for the binary choice symbol "$".

The stay macro tree transducer is then constructed similar to the deterministic variant but with the modified definition of the transformation $U(.)$ of right-hand sides:

$$\begin{aligned} U(\mathtt{b}'(A_1, \ldots, A_k)) &= \mathtt{b}'(U(A_1), \ldots, U(A_k)) \\ U(q_\mu(x_1 = \mathsf{father}(x_0))) &= y_\mu \\ U(q_\mu(x_1 = x_0)) &= q_\mu(z_0) \\ U(q_\mu(x_1 = \mathsf{child}_j(x_0))) &= q_\mu(z_j, q_1(z_0, y_1, \ldots, y_n), \ldots, q_n(z_0, y_1, \ldots, y_n)). \end{aligned}$$

The transformation is now simpler, because we are able to use the stay rules for expressing a repeated transformation of the current node. □

Lemma 5.7 The transformation of a nondeterministic IO 2-mtt P can be effectively decomposed into

$$Y \circ \tau_M,$$

where M is a nondeterministic IO stay-mtt and Y is a transfromation of the class YIELD_c.

Proof. Again, we first decompose P into a 2-mtt P' without accumulating parameters. Therefore we rewrite each rule

$$q(\mathsf{label}_a(x_0), y_1, \ldots, y_k) \to A \quad \text{as} \quad q(\mathsf{label}_a(x_0)) \to S(A),$$

where $S(.)$ is defined as in Lemma 5.5. We also assume that the input is relabeled as in Lemma 5.5. Recall that this relabeling can be performed in one step together with the simulating mtt M [EV85].

The resulting transducer can now be transformed into an IO stay-mtt M. We assume that P' consists of functions $q_1, \ldots, q_n$. For each of these n functions, the stay macro tree transducer M has two variants:

(1) $\overline{q_\nu}$ is used if the current node is the root node, i.e., no up-moves can be performed and therefore no parameters are necessary;
(2) q_ν is used for every other node and has n accumulating parameters for capturing all possible next actions.

On the basis of this set, we replace each rule

$$q_\nu(\mathsf{label}_d(x_0)) \to t$$

with $d = \langle \mathtt{a}, j \rangle$ by the new stay-rules

$$\begin{array}{ll} \overline{q_\nu}(z_0) & = U_0(t), \text{ if } j = 0 \\ q_\nu(z_0, y_1, \ldots, y_n) & = U_d(t), \text{ for } j \neq 0. \end{array}$$

The new mtt M operates on input variables z_0 for stay-rules and z_1, z_2 for binary symbols.

The right-hand sides t are converted with the rewrite function $U_d(.)$ which is recursively defined over the structure of right-hand sides:

$$\begin{array}{ll} U_d(\mathtt{b}'(A_1, \ldots, A_k)) & = \mathtt{b}'(U_d(A_1), \ldots, U_d(A_k)) \\ U_d(q_\mu(x_1 = \mathsf{father}(x_0))) & = y_\mu \\ U_d(q_\mu(x_1 = x_0)) & = q_\mu(z_0, y_1, \ldots, y_n) \\ U_d(q_\mu(x_1 = \mathsf{child}_j(x_0))) & = q'_{i,j}(z_0, \; iter_{1,d}(z_0, y_1, \ldots, y_n), \ldots, \\ & \qquad iter_{n,d}(z_0, y_1, \ldots, y_n)). \end{array}$$

Since our mtt model allows only rules which either contain input variable z_0 representing the current node (the stay-rules) or input variables $z_1, z_2, \ldots$ representing the first, second,... son of the current node, we have to postpone the transformation of the j-th son in the last equation by introducing functions $q'_{\mu,j}$. They are defined for all binary labels d as follows:

$$\begin{array}{l} q'_{\mu,1}(d(z_1, z_2), y_1, \ldots, y_n) \to q_\mu(z_1, y_1, \ldots, y_n) \\ q'_{\mu,2}(d(z_1, z_2), y_1, \ldots, y_n) \to q_\mu(z_2, y_1, \ldots, y_n). \end{array}$$

Function $iter_{i,d}$ iterates nondeterministically over all possible definitions of function q_i applied to a node labeled with d. Thus, it allows to simulate the IO evaluation of nested function calls. It either returns *undefined* ("$\perp$") or the choice over all appropriate right-hand sides of function q_i. Given that

$$q_i(\mathsf{label}_d(x_0)) \to t_1 \mid \ldots \mid t_k,$$

it is defined as

$$\begin{array}{l} iter_{i,d}(z_0, y_1, \ldots, y_n) \to \perp \\ iter_{i,d}(z_0, y_1, \ldots, y_n) \to \$(U_d(t_1), \$(\ldots, \$(U_d(t_{k-1}), U_d(t_k)) \ldots)). \end{array}$$

The new output symbol "$" represents the choice between its left and right child. Function $iter_{i,d}$ again calls for each right-hand side $t_1, \ldots, t_k$ the above defined transformation $U_d(.)$.

If the current node is the root node, i.e., the label is of the form $d = \langle \mathtt{a}, 0 \rangle$, then the right-hand sides are transformed by function $U_0(.)$:

$$\begin{aligned} U_0(\mathfrak{b}'(A_1,\ldots,A_k)) &= \mathfrak{b}'(U_0(A_1),\ldots,U_0(A_k)) \\ U_0(q_\mu(x_1 = x_0)) &= \overline{q_\mu}(z_0) \\ U_0(q_\mu(x_1 = \mathsf{child}_j(x_0))) &= q'_{\mu,j}(z_0, \overline{iter}_{1,d}(z_0),\ldots,\overline{iter}_{n,d}(z_0)). \end{aligned}$$

Accordingly, the iterating function $\overline{iter}$ has also no parameters and uses only the parameterless version of functions $q_1,\ldots,q_n$. Its definition is nearly equal to that of *iter* above and therefore omitted. □

Theorem 5.3 illustrates that every TL program can be decomposed into a macro tree-walking transducer followed by the evaluation of the symbolic concatenation given that the input is annotated with information about the used patterns. Together with Lemma 5.5, we obtain our main decomposition result for deterministic TL programs.

Theorem 5.8 There effectively exists a total and deterministic mtt M for every determinitic TL program P such that

$$\tau_P = A \circ Y \circ \tau_M,$$

where the transformations A and Y only depend on the used alphabet.

Proof. According to our constructions so far, we have decomposed the transformation realized by a TL program P into a composition

$$A \circ Y \circ \tau_M \circ R.$$

This follows from Theorem 5.3 and Lemma 5.5. The theorem follows from the fact that macro tree transducers are effectively closed under relabelings of the input from REL, by Theorem 4.21 and Corollary 4.10 of [EV85]. □

With Lemmata 5.6 and 5.7 we obtain a corresponding result for nondeterministic TL programs.

Theorem 5.9 The transfromation of every nondeterministic TL program P can be decomposed into the composition

$$A \circ Y \circ \tau_M,$$

where M is a stay-mtt which can be effectively constructed from P.

Proof. By Theorem 5.3 and Lemmata 5.6 or 5.7, we have decomposed a TL program P into a composition

$$A \circ Y \circ \tau_M \circ R.$$

The theorem follows from the fact that IO as well as OI macro tree transducers are effectively closed under relabelings of the input from REL [EV85]. □

These two theorems substantiate our main result: recognizability of sets of forests is effectively preserved by taking pre-images of TL transformations. Every TL program can be effectively compiled into a composition of three (stay) macro tree transducers, and since their introduction it is known that taking pre-images of compositions of macro tree transductions effectively preserves recognizability (Theorem 7.4 of [EV85]). The same also holds for stay macro tree transducers (Corollary 44 of [EM03a]).

Theorem 5.10 Let Σ be an alphabet. For every TL program P and recognizable set $R \in \mathcal{F}_\Sigma$ of output forests, the pre-image

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

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

As a corollary this result gives a solution for the type checking problem for TL programs. Assume we are given recognizable sets $T_{\text{in}}, T_{\text{out}} \in \mathcal{F}_\Sigma$ of the admissable input and output types, respectively. In order to check that the set of outputs produced by a TL program *Prog* for every $f \in T_{\text{in}}$ always conformes to T_{out}, we perform *inverse type inference* [MSV00]. Using Theorem 5.10, type checking can be done as follows. First, determine a finite automaton for the set

$$U = \tau_P^{-1}(\mathcal{F}_\Sigma T_{\text{out}})$$

which is possible by Theorem 5.10 and because of the fact that recognizable sets are effectively closed under complement. The set U represents the set of all inputs for which the transformation τ_P returns results which do not conform to T_{out}. In the second step, it hence remains to verify whether or not $T_{\text{in}} \cap U = \emptyset$. In particular, no wrong results are produced by program P if and only if the intersection is empty. Since intersection emptiness is decidable for finite automata, we obtain the following result.

Corollary 5.11 Type checking for TL programs is decidable.

5.4 The Decomposition by Example

In oder to illustrate the key ideas of the decomposition of a TL program into at most three macro tree transducer, we here show the steps by means of our example TL program.

Recall that the program moves all mails which are marked as spam from the inbox into the trash folder.

$$
\begin{array}{lll}
1 & q_0(\mathsf{label}_{\texttt{MailDoc}}(x_1)) & \longrightarrow \langle\texttt{MailDoc}\rangle \\
2 & & \quad q_0(\mathsf{label}_{\texttt{Inbox}}(x_2)) \\
3 & & \quad q_0(\mathsf{label}_{\texttt{Trash}}(x_2)) \\
4 & & \langle\texttt{/MailDoc}\rangle \\
5 & q_0(\mathsf{label}_{\texttt{Inbox}}(x_1)) & \longrightarrow \langle\texttt{Inbox}\rangle \\
6 & & \quad q_2(x_1/x_2 \wedge \mathsf{label}_{\texttt{Mail}}(x_2)) \\
7 & & \langle\texttt{/Inbox}\rangle \\
8 & q_0(\mathsf{label}_{\texttt{Trash}}(x_1)) & \longrightarrow \langle\texttt{Trash}\rangle \\
9 & & \quad q_2(\exists\, z.\, z; x_1 \wedge z/x_2 \wedge \mathsf{label}_{\texttt{Spam}}(x_2)) \\
10 & & \quad q_2(x_1/x_2) \\
11 & & \langle\texttt{/Trash}\rangle \\
12 & q_2(x_1) & \longrightarrow x_1
\end{array}
$$

In the first step of the decomposition, the TL program is translated into a macro tree-walking transducer. This translation compiles the global selections specified by the occurring select patterns into local movements.

Consider, for example, the rule transforming the `Inbox`-element (line 5). It selects in its right-hand side each child which is labeled with `Mail` and copies it into the output. This rule is simulated by the following 2-mtt rule:

$$
\begin{aligned}
q_0(\mathsf{label}_{\langle\texttt{Inbox},\ldots\rangle}(x_1)) \longrightarrow \texttt{Inbox}(\, & \mathsf{up}^{q_2,1}_{\psi,h}(\; x_2 = \mathsf{father}(x_1), \\
& \qquad \mathsf{down}^{q_2}_{\psi,g_1}(x_2 = \mathsf{child}_1(x_1))@ \\
& \qquad \mathsf{down}^{q_2}_{\psi,g_2}(x_2 = \mathsf{child}_2(x_1))), \epsilon),
\end{aligned}
$$

where ψ denotes the pattern $x_1/x_2 \wedge \mathsf{label}_{\texttt{Mail}}(x_2)$.

In the next step, we reformulate the macro tree-walking transducer into one without accumulating parameters. Therefore, the passing of parameters and substitutions are expressed by symbolic operators α_i and σ_i. Thus, the rule is rewritten as:

$$
\begin{aligned}
q_0(\mathsf{label}_{\langle\texttt{Inbox},\ldots\rangle}(x_1)) \longrightarrow & \\
\texttt{Inbox}(\, & \sigma_1(\mathsf{up}^{q_2,1}_{\psi,h}(\; x_2 = \mathsf{father}(x_1)), \\
& \qquad \sigma_0(\mathsf{down}^{q_2}_{\psi,g_1}(x_2 = \mathsf{child}_1(x_1)))@ \\
& \qquad \sigma_0(\mathsf{down}^{q_2}_{\psi,g_2}(x_2 = \mathsf{child}_2(x_1))))), \epsilon),
\end{aligned}
$$

Since the function moving up in the input has one parameter, it is grouped into a σ_1 operator, while the down functions are grouped into σ_0.

In the last step, we have to create from this macro tree-walking transducer an ordinary stay macro tree transducer, by eliminating each up move. Assuming that the result of $\mathsf{up}^{q_2,1}_{\psi,h}$ applied to the father of the current node is passed in the k-th parameter, the rule is rewritten as:

$$
\begin{aligned}
q_0(\langle\texttt{Inbox},\ldots\rangle(z_1,z_2), y_1,\ldots,y_k,\ldots,y_n) \longrightarrow & \\
\texttt{Inbox}(\, & \sigma_1(y_k, \sigma_0(\mathsf{down}^{q_2}_{\psi,g_1}(z_1, s_1,\ldots,s_n))@ \\
& \qquad \sigma_0(\mathsf{down}^{q_2}_{\psi,g_2}(z_2, s_1,\ldots,s_n)))), \epsilon),
\end{aligned}
$$

where s_i are the results of applying each function to the current node. By these three steps, every TL program can be transformed into a stay macro tree transducer with symbolic substitution and symbolic concatenation.

5.5 Notes and References

The decomposition of TL programs into at most three stay macro tree transducers have been published in [MBPS05]. There, we invastigated only the decomposition of *outside-in* transformations, meaning that, in the presence of nested function calls, we fixed order of evaluation to be outside-in.

The *macro tree-walking transducer* — the intermediate model for compiling global selections into local movements — equals the CFT(Tree-walk) transducer model defined in [EV86], because they combine the features of context-free tree grammars with a tree-walk facility [EV86;EM03a]. In [EV86] the so-called *grammar with storage* is introduced as a transducer model. Basically, "*a storage type is specified by a set of configurations which can be tested by predicates and transformed by instructions*" [EV86]. The nonterminals of the grammar are viewed as the functions of the transducer. If each occurrence of a nonterminal N then is associated with a configuration c, the grammar is enabled to act on a storage type. Since a grammar rule can only be applied to a nonterminal-configuration pair $N(c)$ if a test (specified in the rule) holds for the configuration c, the "*derivations of the grammar are controlled by the storage configurations*" [EM03a]. The new configuration-nonterminal pairs are obtained from the right-hand side of the rule and a specified transformation of c. The 2-mtt is also equal to the 0-pebble macro tree transducer introduced in [EM03a], and thus is in one-to-one correspondence with the macro attributed tree transducer of [KV94;FV98].

XQuery [BCF$^+$06] treats XML data as immutable trees. Moreover, the nodes of the input also have a physical identity, which means that parts may be either identical or equal as labeled trees. Type checking is performed via *forward* type inference, i.e., starting out from the given input type, the processor infers for each function its output type and checks whether these inferred types are consistent in themselves and with respect to a given output type. This means, type checking is only approximative and type errors might occur at run-time. Since XQuery is able to generate new element and attribute names, the corresponding type rule must be pessimistically assume that the result type can be of any kind [MS05].

A large class of transformations which was specifically designed in order to model most of the existing XML transformation languages, is the k-pebble tree transducer [MSV00]. One of its most important conceptional characteristics is that type checking is decidable with respect to regular tree languages. Milo et al. present one of the first techniques for exact type checking and familiarize the XML community with the method of inverse type inference.

This means, instead of inferring the possible outputs of a given transformation F, they compute the pre-image

$$U = \{d \mid F(d) \notin T_{\text{out}}\}$$

of all inputs d that result in outputs which do not conform to the given output type T_{out}. This set gives a solution for the type checking problem, because for a given input type T_{in}, one simply has to check whether the intersection of U and T_{in} is empty or not. If it is empty, for all inputs conforming to T_{in} only outputs of type T_{out} are produced.

In principle, the k-pebble tree transducer is a finite state transducer marking nodes of the input tree with up to k different pebbles. When modeling an XML transformation, one extra pebble is used for each node that appears in the pattern of the transformation [MSV00]. This means that the number k depends on the complexity of the patterns that are used to realize the transformation. Unfortunately, this number k influences the time complexity of type checking a k-pebble tree transducer. The best known time complexity is a tower of exponents whose height grows with the number of used pebbles [EM03a].

In order to obtain more efficient type checking algorithms, one can restrict the expressive power of the type and transformation formalism. Restrictions under which type checking for a fragment of top-down XSLT can be done in polynomial time were investigated in [MN02; MN04]. Since type checking quickly turns undecidable for query languages that are able to test equality of data values, Martens and Neven focus on simple top-down recursive (structural) transformations motivated by XSLT and recursion on trees [MN05]. For deleting and unbounded copying transformations, they show that type checking is hard for EXPTIME. Restricting to non-deleting transductions, the complexity of type checking lowers to PSPACE, when the output type is given by DTDs [MN05]. Moreover, they show that type checking becomes tractable, i.e. is in PTIME, for restricted deletion and copying. The most interesting fact of this restricted class of transformations is that arbitrary deletion is allowed when no copying occurs, and bounded copying is permitted for all transformation rules that delete only in a bounded fashion [MN04]. Here, the ability of deleting is measured in the number of states occurring in the top-level of a right-hand side, while the ability of copying is bounded by a number k if there are at most k occurrences of states in every sequence of siblings in the right-hand sides of the transducer. This definition is motivated by the fact that a function occurring in a right-hand side of a rule is applied to every subtree of the current node. This means, that a rule $(q\mathtt{a}) \to \mathtt{b}(q_1)$, transforms every subtree of `a` with function q_1 and puts the resulting forest under the new output symbol `b`. This corresponds to the following macro forest transducer:

$$\begin{aligned} q(\mathtt{a}\langle f_1\rangle\, f_2) &\to \mathtt{b}\langle \mathit{repeat}_{q_1}(f_1)\rangle \\ \mathit{repeat}_{q_1}(\mathtt{c}\langle f_1\rangle\, \mathit{rest}) &\to q_1(f_1)\, \mathit{repeat}_{q_1}(\mathit{rest}). \end{aligned}$$

Pankowski shows that this approach is both descriptive and expressive, and illustrates how to use these unranked top-down tree transducers to specify and perform transformations of XML documents [Pan03a; Pan03b].

XDuce and CDuce [HP01; BCF03] support a local form of type inference where types are explicitly specified for the function arguments and their return values, while they are inferred for pattern matching. The type system is based on the notion of regular expression types, which correspond to the class of regular tree languages. The most essential part of the type system is the subtyping relation, which is defined by inclusion of the values represented by the types [MS05]. XDuce's type checker is conservative in the sense that a program that passes type checking is guaranteed to transform valid input into valid output. On the other hand, there exist programs that are correct, i.e., type-safe, but where appropriate type annotations are not expressible. Hence, type checking is approximative, and possibly correct programs are rejected.

Another functional approach to XML processing is XMλ [MS99]. Here, the type information is mapped onto extended Haskell types. As in XQuery and XDuce, type inference is approximate.

In [AMF$^+$01; AMF$^+$03] type checking is considered from a different point of view. Milo et al. investigate the type checking problem fro transformations relational data into tree data. The problem consists of statically verifying that the output of every transformation belogns to a predefined output tree languages. This means, the transformations do not map trees onto trees, but database entries to trees. Therefore, they introduce the transformation language *TreeQL*. A program is a tree, where each node is labeled with a symbol of the output tree language and a formula extracting some entries from the database. Additionally, the free variables of a formula at a descendant node in the program must be a superset of the free variables at its ancestors [AMF$^+$03]. For this class of transductions it is shown that type checking is undecidable in its most general formulation, but can be done in polynomial time, when the formulas in the TreeQL programs are restricted to conjunctive queries with a bounded number of variables and the output type specifies only cardinality constraints on the tags of the children of a node, but does not restrict their order [AMF$^+$01].

6

Type Checking by Forward Type Inference

In general, the type checking problem for XML transformations is undecidable. Hence, every solution which deals with all possible transformation features has to be approximative. However, this seem to work well for practical XSLT transformations. Another approach is to restrict the types and transformations in such a way that type checking becomes decidable.

In the previous chapter we restricted the transformations to be expressible in the language TL. For this transformation language, we have provided an exact type checking algorithm by decomposing every TL program into at most three stay macro tree transducers — independent of the match and select patterns used by the transformation.

Even though the class of translations for which type checking is decidable is surprisingly large, the price to be paid for exactness is also extremely large: the complexity of the known algorithms for compositions of stay-mtts is a tower of exponents whose height grows with the number of transducers in the composition. For practical considerations, however, one is interested in useful subclasses of transformations for which type checking is tractable.

In this chapter, we report on another approach to type checking XML transformations: 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

(1) a *linear* stay macro tree transducer which never translates the same node more than once at the same time, or
(2) a *b-bounded* stay macro tree transducer which translates every node only a bounded number of times.

Note that no restriction is put on the copying that the stay-mtt applies to its accumulating parameters, i.e., parameters can freely be copied. Note further, that we focus on nondeterministic tree transducers with call-by-value semantics, i.e., which evaluate nested function calls in an inside-out mode.

Opposed to the techniques in the previous chapter, this approach is based on *forward type inference* which means that we compute for a given transformation the possible set of outputs with respect to a given input type, and then test whether or not this computed set is contained in the desired output type.

First, we generalize the well-known triple construction for context-free grammars to provide a general construction for stay-mtts which produce only output trees of the language accepted by some deterministic finite automaton (Section 6.1). In Section 6.2 we use stay moves to cut down the numbers of function calls in right-hand sides which crucially affect the complexity of the construction. We also present a formulation by 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 stay-mtts but automatically meets the improved time bounds on the provably efficient sub-classes (Sections 6.3 and 6.4). We close the chapter by generalizing the techniques from stay-mtts to stay macro forest transducers which additionally provide built-in support for the concatenation of forests.

In the following, we will not mention given input types in our discussions and theorems explicitly. Instead, we implicitly assume that this type has been encoded into the stay-mtt. This can be done as follows. Assume that the input type S is given by a possibly nondeterministic finite tree automaton $A = (P, \Sigma, F, \delta)$ (cf. Section 2.3). From a stay-mtt $M = (Q, \Sigma, Q_0, R)$, we then build a new stay-mtt $M_A = (Q', \Sigma, Q_0', R')$ whose function symbols are pairs consisting of a function symbol of M and an automaton state of A, i.e.,

$$Q' = \{\langle q, p\rangle \mid q \in Q \text{ and } p \in P\} \cup \{\langle q, \top\rangle \mid q \in Q\}$$

with $\top \cap P = \emptyset$. For each rule $q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t$ with $\mathtt{a} \in \Sigma$ we create new rules

$$\langle q, p\rangle(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to [t]\ \rho,$$

where for all $1 \leq j \leq n$, $\rho(x_j) = p_j$ if $(p, \mathtt{a}, p_1 \cdots p_n) \in \delta$. $[t]\ \rho$ describes the rewriting of the right-hand sides and is inductively defined over their structure

$$\begin{aligned}
[y_i]\ \rho &= y_i \\
[\mathtt{b}(x_1, \ldots, x_{m_1})]\ \rho &= \mathtt{b}([t_1]\ \rho, \ldots, [t_{m_1}]\ \rho) \\
[q'(x_j, t_1, \ldots, t_{m_2})]\ \rho &= \langle q', \rho(x_j)\rangle(x_j, [t_1]\ \rho, \ldots, [t_{m_2}]\ \rho).
\end{aligned}$$

This means, a predecessor state p_j corresponds to the input variable x_j and therefore occurs in the new right-hand sides as the second component in recursive calls on x_j.

Since stay-rules only change the state of the transducer independent from the input, it is also not necessary to consider a change of the tree automata states. Thus, a stay-rule $q(x_0, y_1, \ldots, y_k) \to t$ is rewritten as

$$\langle q, \top \rangle(x_0, y_1, \ldots, y_k) \to [t]\, \emptyset$$

where $\top$ signalizes that this rule is valid for all states of M and $\emptyset$ is the empty assignment.

In order to deal with variables x_j not occurring in the right-hand side, we introduce extra functions $\mathsf{check}_{p'}$ for every state $p' \in P$ such that $\mathsf{check}_{p'}(s, y_1)$ produces y_1 if and only if there is a run of A on the tree s which results in a state p'.

Then the new set of initial states is the set of all pairs consisting of an initial state of M and an accepting state of A, thus

$$Q_0' = \{\langle q, p \rangle \mid q \in Q_0 \text{ and } p \in F\}.$$

In particular, the new stay-mtt M_A is of size $\mathcal{O}(|M| \cdot |A|)$, where $|M|$ is the size of the transducer and $|A|$ is the size of the tree automaton representing the input type. Since the construction of M_A does not add new function calls in rules of M, M_A is linear in the input variables $x_0, x_1, \ldots$ if M is.

6.1 Intersection with Regular Languages

We show that for a stay macro tree transducer and a given output type, it is possible to construct a new stay macro tree transducer which produces only outputs of the desired type. In fact, if the output type is given as finite tree automaton, the corresponding construction is a rather straight-forward generalization of the triple construction known for context-free languages. In case of stay-mtts, the construction is even simpler if we additionally assume that the recognizable tree language is given by a deterministic finite tree automaton. Remember that these are finite tree automata where the transition relation defines a function as explained on page 15.

Theorem 6.1 Let Σ be an alphabet. For a stay-mtt M and a deterministic finite tree automaton A there is a stay-mtt M_A with

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

for all trees $t \in \mathcal{T}_\Sigma$.

The stay-mtt 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. For a given alphabet Σ let $M = (Q, \Sigma, Q_0, R)$ be a stay-mtt and let $A = (P, \Sigma, \delta, P_f)$ be the deterministic bottom-up finite state tree automaton where P is the set of states, $P_f \subseteq P$ is the set of final states, and $\delta : \Sigma \times P^k \to P$ is the transition function.

The set of function symbols of the new stay-mtt $M_A = (Q_A, \Sigma, Q_{0,A}, R_A)$ consists of all pairs whose first component is a function symbol of rank $k+1$ of M and whose second component is a sequence of length k of states of A, i.e.,

$$Q_A = \{\langle q, p_0 \dots p_k\rangle \mid q \in Q_{k+1},\ p_0, \dots, p_k \in P\}.$$

One of these new function symbols $\langle q, p_0 \dots p_k\rangle$ is meant to generate all trees t containing variables from $y_1, \dots, y_k$ for which there is a run of automaton A starting at the leaves y_i ($i = 1, \dots, k$) with states p_i and reaching the root of t in state p_0.

The rule set R_A of the intersection stay-mtt M_A consists of all rules

$$\langle q, p_0 \dots p_k\rangle(\pi, y_1, \dots, y_k) \to t'$$

for each rule $q(\pi, y_1, \dots, y_k) \to t$ of M, where either $\pi = x_0$ or $\pi = \mathtt{b}(x_1, \dots, x_l)$, and a tree $t' \in T^{p_0 \cdots p_k}[t]$, where the sets $T^{p_0 \cdots p_k}[t]$ are inductively defined over the structure of right-hand sides

$$\begin{aligned}
T^{p_i p_1 \cdots p_k}[y_i] &= \{y_i\} \\
T^{p_0 p_1 \cdots p_k}[\mathtt{a}(t_1, \dots, t_m)] &= \{\mathtt{a}(t'_1, \dots, t'_m) \mid \delta_{\mathtt{a}}(p'_1, \dots, p'_m) = p_0 \text{ and} \\
&\qquad \forall i \in \{1, \dots, m\} : t'_i \in T^{p'_i p_1 \cdots p_k}[t_i]\} \\
T^{p_0 p_1 \cdots p_k}[q'(, t_1, \dots, t_n)] &= \{\langle q', p_0 p'_1 \dots p'_m\rangle(t'_1, \dots, t'_n) \mid \\
&\qquad \forall i \in \{1, \dots, n\} : t'_i \in T^{p'_i p_1 \cdots p_k}[t_i]\}.
\end{aligned}$$

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

$$[\![\langle q, p_0 \dots p_k\rangle]\!](s) = [\![q]\!](s) \cup \{t \in \mathcal{T}_\Sigma(Y) \mid \delta^*(t, p_1 \dots p_k) = p_0\} \tag{6.1}$$

$Y = \{y_1, \dots, y_k\}$ and δ^* is the extension of the transition function of automaton A to trees containing variables from Y, namely

$$\begin{aligned}
\delta^*(y_i, p_1 \dots p_k) &= p_i \\
\delta^*(\mathtt{a}(t_1, \dots, t_m), p_1 \dots p_k) &= \delta_{\mathtt{a}}(\delta^*(t_1, p_1 \dots p_k), \dots, \delta^*(t_m, p_1 \dots p_k))
\end{aligned}$$

for all symbols $\mathtt{a} \in \Sigma$. The set of new initial function symbols then consists of all $\langle q, p_f\rangle$, where $q_0 \in Q_0$ and $p_f \in P_f$ is an accepting state of A. Then the correctness of the construction follows from equation (6.1).

Let the stay-mtt M be of size N with at most k parameters and at most d occurrences of function calls in the right-hand sides, and let n be the number of states of the deterministic tree automaton. Since there can be (in the worst case) n^{k+1} copies of a rule of M; and for every non-terminal occurring in the right-hand sides we may choose arbitrary output states, the intersection grammar is of size $\mathcal{O}(N \cdot n^{k+1+d})$. This completes the proof. □

6.2 Shortening of Right-hand Sides

In general, the number of occurrences of states in a right-hand side of a rule can be arbitrarily large. Stay macro tree transducers, however, allow a construction which cuts down the depth of right-hand sides to at most 2.

The key idea of Lemma 6.2 is to split the right-hand sides into their subterms and to organize the execution or rather the data-flow 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 new 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 a rule in M is a state with m parameters, then each new state also has rank m. Accordingly, m parameters are passed in each of the new rules, which explains the size increase of at most k^2.

Lemma 6.2 For every stay-mtt M, a stay-mtt M' can be constructed with the following properties:

(1) $\tau_M = \tau_{M'}$;
(2) whenever a right-hand side of M' is not contained in $\mathcal{T}_\Sigma(Y)$, 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 $k+1$; and
(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.

Proof. For alphabet Σ, let $M = (Q, \Sigma, Q_0, R)$, and let k denote the maximum of the maximal rank of output symbols and the maximal number of accumulating parameters of a state.

In the first step, we show that from the stay-mtt M we can construct a stay-mtt M_1 of size $\mathcal{O}(|M|)$ which is equivalent to M and whose only rules of depth exceeding 2 either do not contain input variables x_i or are stay-rules. The idea is to replace $q, \mathtt{a}$-rules

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

of M where t contains variable x_j and is of depth exceeding 2, with the rules

$$\begin{array}{rcl} q(x_0, \quad y_1, \ldots, y_k) & \to & t' \\ \langle p, \mathtt{a}, j\rangle(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_m) & \to & p(x_j, y_1, \ldots, y_m). \end{array}$$

The new right-hand side t' is obtained from t by replacing each call $p(x_j, \ldots)$ with a call of the new function $\langle p, \mathtt{a}, j\rangle$ applied to the current node x_0, i.e.,

$$p(x_j, \ldots) \quad \text{is replaced by} \quad \langle p, \mathtt{a}, j\rangle(x_0, \ldots).$$

Thus, we postpone pattern matching against the current input subtree until the recursive call $p(x_j, \ldots)$.

In the next step, we simulate each stay-rule

$$q(x_0, y_1, \ldots, y_k) \to t$$

of M_1, where t is too deep, by rules of depth at most 2. Therefore, we introduce separate states for every proper subterm of t. Thus, a rule

$$q(x_0, y_1, \ldots, y_k) \to \sigma(t_1, \ldots, t_i, \ldots, t_n)$$

where σ is either a function or an output symbol, and subterm t_i is deeper than 1, is simulated by

$$\begin{array}{rcl} q(x_0, y_1, \ldots, y_k) & \to & \sigma(t_1, \ldots, q_{t_i}(x_0, y_1, \ldots, y_k), \ldots, t_n) \\ q_{t_i}(x_0, y_1, \ldots, y_k) & \to & t_i \end{array}$$

for a new auxiliary state q_{t_i}. Since every subterm of depth at most 2 can only contain $k+1$ states, the assertion follows. □

Note that this reduction of right-hand sides undoubtedly influences the execution of a stay-mtt. Assume therefore that both M and M' work on a given 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.

6.3 Linear Stay-Mtts

We first consider stay macro tree transducers, where each input variable x_i occurs at most once in every right-hand side of the stay-mtt. stay-mtts as well as ordinary mtts satisfying this restriction are called *linear* (cf. Definition 4.2 or [EV85]). Note that there is no restriction on the use of parameters. Linearity means, that no input node is transformed more than once in a single transformation step.

Linearity for a stay-mtt in particular implies that the number of function calls in right-hand sides is bounded by the maximal rank of the input symbols. We observe for linear stay-mtts that their output languages can be described by means of rules where the input arguments of all occurring function symbols are simply omitted.

Accordingly, the resulting rules no longer specify a transformation from input trees into output trees of a special shape. Instead, the rule system now *generates* output trees of that shape. In this way, we obtain a set of rules which constitutes a *context-free tree grammar* (cftg).

Context-free tree grammars were invented in the late 60s [Rou70a;Rou70b], gaining the structural intuition about macro grammars [Fis68b]. Regular tree

grammars are natural generalizations of regular grammars from strings to trees [Bra69]. As a consequence, nonterminals can only occur as leaves in the right-hand sides of the productions of regular tree grammars. This restriction was lifted by Rounds by allowing for arbitrary nodes in the right-hand sides to be labeled with a nonterminal [Rou70b]. In this way Rounds generalized context-free grammars to work on trees. See [ES77; ES78] for a comprehensive study of the basic properties of context-free tree grammars and an exact and detailed differantiation between inside-out and outside-in context-free tree languages.

Formally, a context-free tree grammar G can be represented by a tuple

$$(N, \Sigma, P, N_0),$$

where N is a finite ranked set of nonterminals, $N_0 \subseteq N$ is a set of initial symbols of rank 0, Σ is the finite ranked alphabet of terminal symbols, and P is a finite set of rules of the form

$$q(y_1, \ldots, y_k) \to t$$

where $q \in N_k$ is a nonterminal of rank $k \geq 0$. The right-hand side t is a tree built up from the variables $y_1, \ldots, y_k$ by means of applying nonterminal and terminal symbols.

Analogue to stay-mtts, inside-out (IO) and outside-in (OI) evaluation order for nonterminal symbols must be carefully distinguished [ES77; ES78]. In the context of this chapter, we define the evaluation mode to be *IO* or *call-by-value* evaluation order in the presence of nested function calls.

Intuitively, the generation induced by a grammar G starts with one of the initial nonterminals. If there occurs a nonterminal $q(t_1, \ldots, t_k)$ for trees $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$ in an already generated tree t, we nondeterministically choose one of the productions for q, then replace in its right-hand side each occurrence of a variable y_i with the corresponding (fully evaluated) tree t_i for all $i = 1, \ldots, k$, and replace in t the nonterminal node by the obtained tree. The generating process ends if the resulting tree does not contain any further nonterminal symbols.

The least fixpoint semantics for the context-free tree grammar G over Σ and with nonterminals N is obtained straightforwardly along the lines for stay-mtts — simply by removing the corresponding input components. In particular, this semantics assigns to every nonterminal $q \in N_k$ of rank $k \geq 0$, a set $[\![q]\!] \subseteq \mathcal{T}_\Sigma(Y)$ for a set of variables $Y = \{y_1, \ldots, y_k\}$. The language generated by grammar G is

$$\mathcal{L}(G) = \bigcup \{t \in \mathcal{T}_\Sigma \mid t \in [\![q_0]\!],\ q_0 \in N_0\},$$

i.e., the language consists of all trees that can be generated starting from an initial nonterminal.

By Corollary 5.7 of [EV85] follows that the output language of a linear IO stay-mtt M can be characterized by an IO context-free tree grammar G_M

which can be constructed from M in linear time. The following theorem repeats this result.

Theorem 6.3 The output language of a linear stay-mtt M can be characterized by a context-free tree grammar G_M which can be constructed from M in linear time.

Proof. Given a linear stay-mtt $M = (Q, \Sigma, Q_0, R)$, we construct

$$G_M = (N, \Sigma, P, N_0),$$

where each function of M with k accumulating paramters is represented by a nonterminal of rank k, i.e., $N = \{s_q | q \in Q_{k+1}, rank(s_q) = k\}$ and $N_0 = \{s_q \in E | q \in Q_0\}$. For every rule

$$q(\pi, y_1, \ldots, y_k) \to t$$

of R, where pattern π is either x_0 or $\mathtt{a}(x_1, \ldots, x_n)$ for $\mathtt{a} \in \Sigma_n$, there is exactly one rule in P

$$s_q(y_1, \ldots, y_k) \to rpl(t),$$

where the right-hand side of the production is obtained from t by the rewriting function rpl which is defined as

$$\begin{array}{ll} rpl(y_j) & = y_j \\ rpl(\mathtt{a}(t_1, \ldots, t_n)) & = \mathtt{a}(rpl(t_1), \ldots, rpl(t_n)) \\ rpl(q'(x_i, t_1, \ldots, t_m)) & = s_{q'}(rpl(t_1), \ldots, rpl(t_m)). \end{array}$$

A formal proof that G_M indeed characterizes the output language of M can be found, e.g., in [EV85]. □

The characterization of the output language of a particular stay-mtt by a context-free tree grammar is useful because emptiness for (IO)-cftgs is decidable in linear time. The emptiness check can be done with a similar algorithm as the one for context-free word grammars, see for example [Sal73].

In the word case, the algorithm works as follows. Mark each terminal symbol in every rule of the grammar. Then search in the rule set for a production rule $A \to \alpha$, in which α only consists of marked symbols and nonterminal A is unmarked. If such a production exists, mark A in every rule and repeat this process. If the start symbol of the grammar is unmarked, then the language generated by the grammar is empty, otherwise it is non-empty.

Accordingly, this algorithm can be formalized for context-free tree grammars as follows. Let $G = (N, \Sigma, N_0, P)$ be a cftg over alphabet Σ and let $Y = \{y_1, \ldots, y_m\}$ be a set of variables. Let $U_0 \subseteq U_1 \subseteq U_2 \subseteq \ldots$ be a sequence of subsets of N with

$$\begin{array}{l} U_0 = \emptyset \\ U_{k+1} = U_k \cup \{q \in N \mid \exists q(y_1, \ldots, y_k) \to \alpha \text{ in } P \text{ such that } \alpha \in \mathcal{T}_{\Sigma \cup U_k}(Y)\}. \end{array}$$

Since the set of nonterminals N is finite and $U_k \subseteq U_{k+1} \subseteq N$ for all k follows that there exists a number $n \geq 1$ with $U_n = U_{n+1}$, and for all $k > n$, $U_k = U_n$. The language $\mathcal{L}(G)$ generated by G is obviously empty if $N_0 \notin U_n$ [Fis68a].

The next theorem summarizes some known results about cftgs which are of interest in the context of this work (cf. [GS84; GS97]):

Theorem 6.4 Let G be a context-free tree grammar.

(1) It can be decided in linear time whether or not $\mathcal{L}(G)$ is empty.
(2) For every deterministic finite tree automaton A, a cftg G_A can be constructed such that

$$\mathcal{L}(G_A) = \mathcal{L}(G) \cap \mathcal{L}(A).$$

The grammar G_A can be constructed in time $\mathcal{O}(N \cdot n^{k+1+d})$ where N is the size of G, k is the maximal rank of the nonterminals of G, d is the maximal number of occurrences of nonterminals in right-hand sides of G, and n is the number of states of the finite tree automaton.

The complexity bound provided for the construction of Theorem 6.4 is a worst-case estimation. Instead, we want to point out that the triple construction for the context-free tree grammar G_A can be organized in such a way that only "useful" nonterminals are constructed. Here useful means, that each nonterminal is used in some derivation step.

Therefore, we introduce for every nonterminal q of grammar G with rank k, a predicate q with arity $k+1$, written in the standard Datalog terminology as $q/k+1$ [AHV95]. Every production $q(y_1, \ldots, y_k) \rightarrow t$ of G induces the Datalog implication

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

where the body of the clause is determined by $goals[t]_{Y_0}$. For a variable X it is recursively defined over the structure of t

$$\begin{aligned}
goals[y_j]_X &= X = Y_j \\
goals[\mathtt{a}(t_1, \ldots, t_m)]_X &= \delta(X, \mathtt{a}, X_1, \ldots, X_m) \wedge goals[t_1]_{X_1} \wedge \ldots \wedge goals[t_m]_{X_m} \\
goals[q'(t_1, \ldots, t_n)]_X &= q'(X, X_1, \ldots, X_n) \wedge goals[t_1]_{X_1} \wedge \ldots \wedge goals[t_n]_{X_n}
\end{aligned}$$

where the variables X_i in the last two rows are "fresh" (for all i of $1, \ldots, m$ and $1, \ldots, n$, respectively). For subsets X and $X_1, \ldots, X_k$ of the set of states of automaton A, the predicate $\delta(X, \mathtt{a}, X_1, \ldots, X_k)$ denotes that $(p, \mathtt{a}, p_1, \ldots, p_k)$ for all $p \in X$ and $p_i \in X_i, i = 1, \ldots, k$.

If we add the transitions of the finite tree automaton A as facts, for every nonterminal q of rank k, a bottom-up evaluation of the resulting Datalog program computes the set of all tuples $(p_0, \ldots, p_k)$ such that the evaluation $[\![\langle q, p_0, \ldots, p_k \rangle]\!]$ is non-empty. If we additionally want to restrict these relations only to those tuples which may contribute to a terminal derivation of the initial nonterminal $\langle q_0, p_f \rangle$, we simply may top-down solve the program with the query $\Leftarrow q_0(p_f)$.

Practically, top-down solving organizes the construction in such a way that only "useful" nonterminals of the intersection grammar G_A are considered [AHV95]. Using this approach, the number of newly constructed nonterminals will often be much smaller than the bounds stated in Theorem 6.4.

In general, we are interested in type checking tree transformations which are implemented as stay-mtts. As we have seen in the introduction to this chapter, it is possible to code the input type specification into the stay-mtt M. Moreover, assume that we have given a deterministic bottom-up tree automaton A which describes the output type. Constructing the corresponding complement automaton A^c, type checking for M means to test whether any output of M is accepted by A^c. If transducer M is linear, then the corresponding intersection stay-mtt M_{A^c} is again linear. Thus, its range can be characterized by a context-free tree grammar generating all "illegal outputs" of M with respect to A^c. Therefore, Theorem 6.4 gives us the following type checking result together with a time estimation for linear stay macro tree transducers.

Theorem 6.5 (Type checking linear stay-mtts) Let M be a linear stay-mtt. Then M can be type checked with respect to a given output type.

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

Proof. Following Theorem 6.3, we first construct the context-free tree grammar G_M for stay-mtt M which characterizes precisely the output language of M. Then, following Theorem 6.4, we construct the intersection grammar between G_M and the complement automaton of the deterministic finite tree automaton which represents the specified output type. Finally, we check nonemptiness of the intersection grammar which can be done in linear time, cf. Theorem 6.4. □

If we apply the algorithm in the proof of Theorem 6.5 to *non-linear* stay-mtts, then the constructed context-free tree grammar does no longer precisely characterize the output language of the transformation. This is because dependencies on the input subtrees, viz several function calls on the same input variable x_i, have been lost in the grammar. Rather, the cftg generates a superset of the possible outputs and hence provides a conservative over-approximation.

Theorem 6.6 Let M be a stay-mtt and G_M the context-free tree grammar constructed for M. Then $\tau_M(\mathcal{T}_\Sigma) \subseteq \mathcal{L}(G_M)$.

Since the constructed 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. This means if our type checking routine answers positively, then the program definitly type checks with respect to the given input and output types. On the other hand, a correct program might be rejected as not type safe due to our conservative approximation.

Note that when we approximate the output languages of general stay-mtts with context-free tree grammars, then we no longer assume that the maximal number d of occurrences of nonterminals in a right-hand side of this grammar is bounded by a small constant. If d turns out to be "unacceptably large", we still can apply Lemma 6.2 to limit the maximal number of occurrences of nonterminals in each right-hand side to a number k which is the maximum for the maximal rank of the input symbols and the maximal number of parameters.

From a practical point of view, one does not necessarily need to first construct the context-free tree grammar depicting the output language of the transducer and then test the emptiness of the intersection grammar. Assume, we are given a stay-mtt M and a deterministic tree automaton A which describes the output type. Then, it is possible to directly construct the intersection stay-mtt M_A and test it for emptiness. Therefore, we assemble for each rule

$$q(\pi, y_1, \ldots, y_k) \to t$$

of M, where pattern π either is $\mathtt{a}(x_1, \ldots, x_n)$ or x_0, a Datalog implication

$$q(Y_0, \ldots, Y_k) \Leftarrow \mathit{goals}[t]_{Y_0},$$

where $q/(k+1)$ is a predicate with arity $k+1$ and the body $\mathit{goals}[t]_{Y_0}$ is computed as for grammars shown on page 97. In order to test the emptiness of the intersection stay-mtt M_A, it suffices to query the Datalog program with $\Leftarrow q_0(p)$ for all initial functions q_0 of M and final states p of A.

This gives us the following type checking algorithm for linear stay macro tree transducers:

---

(1) Compute Datalog program $\mathcal{D}(M)$
(2) Compute the intersection stay-mtt M_A by solving $\mathcal{D}(M)$
(3) **if** no accepting states of A are found for the predicates $q_0/1$
(4) **then return True**
(5) **else return False**

6.4 b-bounded Stay-Mtts

In this section we investigate in how far the exact techniques from the last section can be extended to more general classes of stay macro tree transducers. Again our ambition is to find precise and even more important tractable

characterizations of the output language. If the stay-mtt is no longer linear, we must take into account that distinct function calls could refer to the same input node and therefore must be "glued together". This means we pay particular attention to those function calls transforming the same input subtree. Therefore, we first formalize the notion of b-bounded stay-mtts and then extend our type checking methods to this class of transducers.

In general, an arbitrary number of function calls may be applied to the same sub-document of the input. On the other hand, typical transformations consult every part of the input only a small number of times. Therefore, we consider the subclass of stay-mtts processing every subtree of the input at most b times. Thus in principle, b-bounded copying is a semantic property (cf. [MN04] and [EM99]).

Instead of dealing with a semantic definition, we find it more convenient to consider syntactic b-bounded copying only. In order to define this property, let $M = (Q, \Sigma, Q_0, R)$ be a stay macro tree transducer. Assume w.l.o.g. that every state of M is *syntactically productive*, i.e., can produce at least one tree with respect to the cftg approximating M. For all states q of M, we define the maximal copy numbers $\beta[q]$ 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:

$$\beta[q] \;\geq\; \beta[q_1] + \ldots + \beta[q_m],$$

where $q(\mathtt{a}(x_1, \ldots, x_l), y_1, \ldots, y_k) \to t$ is a rule of M and, for some $i, 1 \leq i \leq l$, $q_1, \ldots, q_m$ is the sequence of occurrences of calls $q_j(x_i, \ldots), j = 1, \ldots, m$, 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 stay-mtt M is *syntactically b-bounded copying* (or, a b-stay-mtt for short) if and only if $[q] \leq b$ for all states of M.

The least solutions of such constraint systems over the natural numbers can be determined in linear time [Sei94b]. In fact, [Sei94b] also provides a simple criterion which precisely characterizes whether or not all copy numbers are finite. It amounts to checking that for every constraint $\beta[q] \;\geq\; \beta[q_1] + \ldots + \beta[q_m]$, whenever q and q_j are in the same strong component of the variable dependence graph of the constraint system then the constraint is of the simple form: $\beta[q] \geq \beta[q_j]$ only. Thus, the next propostion follows from the definitions and [Sei94b].

Proposition 6.7 Assume that M is a stay-mtt where every state is syntactically productive.

1. It can be decided in linear time whether the stay-mtt M is syntactically b-bounded for some b.

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

Depart from fundamental considerations, we here are interested in stay-mtts where every input node is visited only a *small* number of times. Although difficult to substantiate, we start out from the idea that most transformations do not process the same input subtree arbitrarily often but only a small number of times. We observe for b-bounded stay-mtts:

Theorem 6.8 For every syntactically b-bounded stay-mtt M the following holds:

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

Proof. Let $M = (Q, \Sigma, Q_0, R)$ a b-bounded stay-mtt. For the first assertion, we claim that for every state $q \in Q$ of M with k parameters,

$$\beta[q] \geq \beta[\langle q, p_0 \dots p_k \rangle]$$

for every sequence $p_0, \dots, p_k$ of automata states where $\beta[\langle q, p_0 \dots p_k \rangle]$ is syntactically productive. This claim is easily verified by fixpoint induction with respect to the corresponding constraint systems characterizing the copy numbers $\beta[q]$ and $\beta[\langle q, p_0 \dots p_k \rangle]$, respectively.

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

Theorem 6.1 provides us with the technical background to prove our main result for syntactically b-bounded stay macro tree transducers.

Theorem 6.9 (Type checking b-bounded stay-mtts) Type checking for a b-bounded stay macro tree transducer M can be done in time $\mathcal{O}(N^b \cdot n^{b \cdot (k+1+d)})$ where N is the size of the stay-mtt M, k is the maximal number of accumulating parameters, d is the maximal rank of an input symbol, and n is the size of a deterministic finite tree automaton for the output type.

Instead of first testing b-boundedness and then running a specialized algorithm, we prefer to formulate a general purpose algorithm which returns correct results for all stay-mtts, but will additionally meet the better complexity bounds on the exhibited sub-classes. Given a stay-mtt M and a deterministic bottom-up tree automaton A, our algorithm works as follows:

(1) For M, compute equivalent stay-mtt M' where the numbers of occurrences of functions in right-hand sides are bounded
(2) Compute Datalog program $\mathcal{D}(M')$
(3) Compute a safe superset of the states of the intersection stay-mtt M'_A by solving $\mathcal{D}(M')$
(4) **if** no accepting states of A are found for the predicates $q_0/1$
(5) **then return True**
(6) **else** compute Horn clauses $\mathcal{H}(M'_A)$
(7) **return** $solve(\mathcal{H}(M'_A))$

In the worst case, this algorithm will be exponential in the number of states of M and double exponential in the number of parameters. Due to the lower bounds for translation emptiness, there is only little hope to get better performances. If on the other hand, the stay-mtt is linear or syntactically b-bounded, the algorithm's complexity will achieve the upper bounds given in Theorems 6.5 and 6.9, respectively. Even more, due to the demand-driven algorithms in steps 3 and 7, we obtain satisfying run-times even for practical relevant transformations, like the one presented in the introduction to Chapter 5.

6.5 Stay Macro Forest Transducers

In the previous sections, we took a closer look at linear and syntactically b-bounded stay macro tree transducers. Since they are defined to work only on ranked trees, they have the disadvantage that stay-mtts are not able to work on forests directly but refer to representations of forests by means of ranked trees. In this context we understand a forest as an arbitrary long sequence of trees. As a consequence, there is no intuitive stay-mtt solution expressing the "simple" operation of lengthening a forest by another sequence of trees. As shown in [PS04], this limitation, though, can be lifted. There, we have proposed macro forest transducers which operate on forests directly and generalize stay-mtts by providing concatenation of forests as additional operation on output forests. This extra feature implies that some macro forest translations cannot be realized by a single stay-mtt alone but only by the composition of a stay-mtt with the transformation *App*. In this section we want to discuss in how far the results for linear and b-bounded macro tree transducers can be generalized to macro forest transducers.

We first show that the emptiness problem is DEXPTIME-complete also for stay-mfts. Furthermore, we present the counterpart of Theorem 6.1 for the forest case, viz. that it is possible to construct for a given stay-mft M and an output type, a stay-mft M' that produces only outputs of the desired type.

Theorem 6.10 Deciding whether the transduction τ_M for a stay macro forest transducer M is not empty is DEXPTIME-complete.

Proof. The lower bound follows since every stay-mtt is also a stay-mft, i.e., s-MTT $\subset$ s-MFT. The algorithm for the upper bound almost literally also works for stay-mfts (cf. Theorem 4.9). □

In order to formalize the result about restricting the range of a stay macro forest transducer, we additionally have to take into account that our deterministic finite state representation of the output type is compatible with concatenations.

Therefore, we fall back upon the known idea of a forest algebra. We propose to use finite forest monoids (cf. the discussion in [BW05]). Let Σ be a finite alphabet. Then a *finite forest monoid* (ffm) consists of a finite monoid M with a neutral element $e \in M$, a subset $F \subseteq M$ of accepting elements, and finally, a function $up : \Sigma \times M \to M$, which maps a symbol of the alphabet Σ together with a monoid element for its content to a monoid element representing a forest of length 1.

Given a deterministic bottom-up tree automaton $A = (P, \Sigma, \delta, F_A)$, we can construct a finite forest monoid as follows. Let $M = P \to P$ be the monoid of functions from the set of automata states P into itself, where the monoid operation is the function composition $\circ : M \times M \to M$ such that for a state $p \in P$ and monoid elements $f, g \in M$, $(f \circ g)(p) = f(g(p))$. In particular, the neutral element of the monoid is the identity function *id*. Moreover, for symbol $\mathtt{a} \in \Sigma$, monoid element $f \in M$, and state $p \in P$, the function *up* is defined as

$$up(\mathtt{a}, f)(p) = \delta_{\mathtt{a}}(p\ f(\delta_\epsilon)),$$

where ϵ is the input symbol denoting the empty forest. Finally, the set of accepting elements is given by

$$F = \{f \in M \mid f(\delta_\epsilon) \in F_A\}.$$

This construction shows that every recognizable forest language can be recognized by a finite forest monoid and although the ffm for a bottom-up tree automaton generally can be exponentially larger, this does not always need to be the case.

Theorem 6.11 Let Σ be an alphabet. For a stay-mft M and a finite forest monoid A there is a stay-mft M_A with

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

for all trees $t \in \mathcal{T}_\Sigma$.

The stay-mft 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 forest monoid A.

Note that the complexity bound postulated for the construction of Theorem 6.11 is not worse than for macro transducers which work on trees. In order to cut down the number d in the exponent of the complexity estimation, we proceed as in Lemma 6.2. For the forest case, however, it is not sufficient to cut down only the depths of the right-hand sides. We additionally have to ensure that all occurring forests also contain at most two trees. This is possible by introducing appropriate states for sub-forests which contain more than two trees. Accordingly, we obtain

Lemma 6.12 For every stay-mft M, a stay-mft M' can be constructed with the following properties:

(1) $\tau_M = \tau_{M'}$;
(2) whenever a right-hand side of M' is not contained in $\mathcal{T}_\Sigma(Y)$, then the depth of t is bounded by 2;
(3) the maximal number of states in a right-hand side of stay-mft M' is at most $\max(k+1, 2)$; and
(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 concepts for delineating or approximating the output languages of stay-mtts given in Sections 6.3 and 6.4 naturally can be extended to stay macro forest transducers as well. We only have to take care that the grammar notions are appropriately generalized to deal with the ability of stay-mfts to concatenate forests.

We introduce the concept of a *context-free forest grammar* (cffg) G as a tuple

$$(E, \Sigma, P, E_0),$$

where E is a finite ranked set of non-terminals, $E_0 \subseteq E$ is a set of initial non-terminals with $rank(p_0) = 0$ for all non-terminals $p_0 \in E_0$, Σ is the finite ranked alphabet of terminal symbols, and P is a finite set of production rules of the form

$$p(y_1, \ldots, y_k) \to f$$

where $p \in E$ is a non-terminal with rank $k \geq 0$. The right-hand sides f are built up from the empty forest and variables $y_1, \ldots, y_k$ by means of concatenation, application of non-terminal and terminal symbols. Note that this new grammar formalism can be considered as a generalization of Fischer's macro grammars [Fis68a] from trees to forests. As in the rest of this chapter, we concentrate on the inside-out mode of evaluating nested non-terminal occurrences.

Since the notion of linearity for stay macro forest transducers is completely analogously defined as linearity for stay-mtts, we characterize the

output language of a stay-mftas a context-free forest grammar. The construction idea again can be viewed as ignoring each occurring input pattern and variable.

Theorem 6.13 For each stay-mft M, a context-free forest grammar G_M can be constructed in linear time such that $\tau_M(\mathcal{F}_\Sigma) \subseteq \mathcal{L}(G_M)$. If M is linear, then even holds $\tau_M(\mathcal{F}_\Sigma) = \mathcal{L}(G_M)$.

Proof. Given a stay-mft $M = (Q, \Sigma, Q_0, R)$ we construct

$$G_M = (N, \Sigma, P, N_0)$$

where each function of M with k accumulating paramters is represented by a non-terminal of rank k, i.e., $N = \{s_q | q \in Q_{k+1}, rank(s_q) = k\}$ and $N_0 = \{s_q \in E | q \in Q_0\}$. For every rule

$$q(\pi, y_1, \ldots, y_k) \to t$$

of R where pattern π is either x_0 or ϵ or $\mathtt{a}\langle x_1\rangle\ x_2$ for $\mathtt{a} \in \Sigma_2$, there is exactly one rule in P

$$s_q(y_1, \ldots, y_k) \to rpl(t)$$

where the right-hand side of the production is obtained from t by the rewriting function rpl which is defined as

$$\begin{array}{ll} rpl(y_j) & = y_j \\ rpl(\mathtt{a}\langle t_1\rangle\ t_2) & = \mathtt{a}\langle rpl(t_1)\rangle\ rpl(t_n) \\ rpl(q'(x_i, t_1, \ldots, t_m)) & = s_{q'}(rpl(t_1), \ldots, rpl(t_m)). \end{array}$$

The proof that G_M indeed characterizes the output language of M is a straight-forward generalization of the proof in [EV85]. □

Due to the analogous definitions of tree and forest grammars, a counterpart to Theorem 6.4 also exists for the forest case and summarizes the observations as follows.

Theorem 6.14 Let G be a context-free forest grammar.

(1) It can be decided in linear time whether or not $\mathcal{L}(G)$ is empty.
(2) For every finite forest monoid M, a cftg G_M can be constructed such that

$$\mathcal{L}(G_M) = \mathcal{L}(G) \cap \mathcal{L}(M).$$

The grammar G_M can be constructed in time $\mathcal{O}(N \cdot n^{k+1+d})$, where N is the size of G, k is the maximal rank of the nonterminals of G, d is the maximal number of occurrences of nonterminals in right-hand sides of G, and n is the number of states of the finite forest monoid.

The latter Theorem 6.14 immediatly gives us a first precise type checking result for linear stay-mfts.

Theorem 6.15 (Type checking linear stay-mfts) Type checking for a linear stay-mft M can be done in time $\mathcal{O}(N \cdot n^{k+3})$ where N is the size of the stay-mft M, k is the maximal number of accumulating parameters, and n is the size of a finite forest monoid for the output type.

Again, we also obtain from this result a method for approximative type checking general transducers. This means we compute a safe superset of the possible outputs and type check the transducer on the basis of this set.

Literally rendering the notion of syntactically b-bounded transducers from the tree case (cf. 100), we can extend the methodes from linear stay-mfts to stay macro forest transducers which inspect their input several but only a bounded number of times.

By analogy with the tree case, we obtain the following result.

Theorem 6.16 For every syntactically b-bounded stay-mft M the following holds:

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

As an immediate consequence, we can state our main theorem for type checking b-bounded stay macro forest transducers. The interesting observation about this result is that type checking for the more expressive model of forest transductions can be solved with the same expenditure of time as for tree transducers.

Theorem 6.17 (Type checking b-bounded stay-mfts) Type checking for a b-bounded stay macro forest transducer M can be done in time $\mathcal{O}(N^b \cdot n^{b\cdot(k+3)})$, where N is the size of the stay-mtt M, k is the maximal number of accumulating parameters, n is the size of a finite forest monoid for the output type.

6.6 Forward Type Checking by Example

Our running example transformation of a mail file (cf. 3) can also be implemented as a stay macro tree transducer working on a binary tree represention of the input. In order to keep the example code small, we omit the content of the `Mail`- and `Spam`-elements.

1 $q(\texttt{MailDoc}(x_1, x_2)) \quad \to \texttt{MailDoc}(q(x_1), \epsilon)$
2 $q(\texttt{Inbox}(x_1, x_2)) \quad \to \texttt{Inbox}(q_{\text{mail}}(x_1), q_{\text{trash}}(x_2, q_{\text{spam}}(x_1)))$
3 $q_{\text{mail}}(\texttt{Mail}(x_1, x_2)) \quad \to \texttt{Mail}(\epsilon, q_{\text{mail}}(x_2))$
4 $q_{\text{mail}}(\texttt{Spam}(x_1, x_2)) \quad \to q_{\text{mail}}(x_2)$
5 $q_{\text{mail}}(\epsilon) \quad \to \epsilon$
6 $q_{\text{spam}}(\texttt{Mail}(x_1, x_2)) \quad \to q_{\text{spam}}(x_2)$
7 $q_{\text{spam}}(\texttt{Spam}(x_1, x_2)) \quad \to \texttt{Spam}(\epsilon, q_{\text{spam}}(x_2))$
8 $q_{\text{spam}}(\epsilon) \quad \to \epsilon$
9 $q_{\text{trash}}(\texttt{Trash}(x_1, x_2), y) \to \texttt{Trash}(q_{\text{trash}}(x_1, y), \epsilon)$
10 $q_{\text{trash}}(\texttt{Spam}(x_1, x_2), y) \to \texttt{Spam}(\epsilon, q_{\text{trash}}(x_1, y))$
11 $q_{\text{trash}}(\texttt{Mail}(x_1, x_2), y) \to \texttt{Mail}(\epsilon, q_{\text{trash}}(x_1, y))$
12 $q_{\text{trash}}(\epsilon, y) \quad \to y$

The first line starts with processing the `MailDoc`-element. Line 2 transformes the `Inbox`-element by producing a new `Inbox`-element in the output and calling function q_{mail} on the left son and function q_{trash} on the right son. The result of trasnforming the content of the `Inbox`-element with function q_{spam} is passed as parameter to q_{trash}. The functions q_{mail} and q_{spam} specify contrary transformations of the content of the `Inbox`-element. While q_{mail} collects all elements labeled with `Mail`, function q_{spam} collects all elements labeled with `Spam`. Function q_{trash} copies its input into the output and passes parameter y until a leaf is reached, and outputs the parameter.

This stay macro tree transducer is *not* linear, because the rule in line 2 calls two functions for the second input variable x_2.

The forward type inference algorithm characteritzes the output language of a b-bounded stay macro tree transducer by means of a context-free tree grammar. Recall, that the tree grammar is obtained from the transducer by omitting the input arguments. Thus, it looks for our example stay-mtt as follows:

1 $q \quad \to \texttt{MailDoc}(q, \epsilon)$
2 $q \quad \to \texttt{Inbox}(q_{\text{mail}}, q_{\text{trash}}(q_{\text{spam}}))$
3 $q_{\text{mail}} \quad \to \texttt{Mail}(\epsilon, q_{\text{mail}})$
4 $q_{\text{mail}} \quad \to q_{\text{mail}}$
5 $q_{\text{mail}} \quad \to \epsilon$
6 $q_{\text{spam}} \quad \to q_{\text{spam}}$
7 $q_{\text{spam}} \quad \to \texttt{Spam}(\epsilon, q_{\text{spam}})$
8 $q_{\text{spam}} \quad \to \epsilon$
9 $q_{\text{trash}}(y) \to \texttt{Trash}(q_{\text{trash}}(y), \epsilon)$
10 $q_{\text{trash}}(y) \to \texttt{Spam}(\epsilon, q_{\text{trash}}(y))$
11 $q_{\text{trash}}(y) \to \texttt{Mail}(\epsilon, q_{\text{trash}}(y))$
12 $q_{\text{trash}}(y) \to y$

In line 2, one can see that only function (nonterminal) q_{trash} keeps its parameter as argument. All other functions are nonterminals without arguments.

In order to check whether the intersection of this grammar and a specified output type is empty or not, a Datalog program of the following form is

constructed. It can by solved by any Datalog solver for a query "q(p_f)", where p_f is a finite state of an automaton describing the predefined output language.

```
q_spam(X34) :-
q_spam(X34).
q_spam(X31) :-
delta( X31, Spam, X32, X33 ),
delta( X32, e ),
q_spam( X33 ).
q_spam(X30) :-
delta( X30, e ).
q_trash(X25, X26) :-
delta( X25, Trash, X27, X28 ),
q_trash( X27, X29 ),
X26=X29,
delta( X28, e ).
q_trash(X20, X21) :-
delta( X20, Spam, X22, X23 ),
delta( X22, e ),
q_trash( X23, X24 ),
X21=X24.
q_trash(X15, X16) :-
delta( X15, Mail, X17, X18 ),
delta( X17, e ),
q_trash( X18, X19 ),
X16=X19.
q_trash(X13, X14) :-
X14=X13.
q_mail(X10) :-
delta( X10, Mail, X11, X12 ),
delta( X11, e ),
q_mail( X12 ).
q_mail(X9) :-
q_mail( X9 ).
q_mail(X8) :-
delta( X8, e ).
q(X5) :-
delta( X5, MailDoc, X6, X7 ),
q( X6 ),
delta( X7, e ).
q(X1) :-
delta( X1, Inbox, X2, X3 ),
q_mail( X2 ),
q_trash( X3, X4 ),
```

```
q_spam( X4 ).
```

6.7 Notes and References

The results presented in this chapter are previously published in [MPS06].

In 6.3 we used Datalog to model the intersection of a context-free tree grammar and a regular tree language. Datalog is regarded as a "toy language" for studying deductive databases; it extends the conjunctive queries with recursion. Although closely related to logic-programming, there are some main differences between Datalog and logic-programming. Datalog only has relation symbols, whereas logic-programming also uses function symbols. Due to the absence of function symbols, Datalog programs always have finite models. Moreover, the expressive power of Datalog lies within PTIME.

Technically, a Datalog program consists of rules of the form

$$R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n),$$

where $R_1, \ldots, R_n$ with $n \geq 1$ are relation names and $u_1, \ldots, u_n$ are free tuples of appropriate arities. Each variable occurring in u_1 must occur in at least one tuple of the right-hand side. The semantics of such a program is the minimal model satisfying all rules [AHV95].

Evaluation techniques are usually devided into two classes: the top-down and bottom-up evaluation. The advantage of top-down solving is that selections which are a part of the initial query can be propagated into the rules as they are expanded. Top-down techniques are therefore more efficient than bottom-up solutions [AHV95].

Discussing the results for stay macro tree transducers with bounded copying, we defined this property as *syntactic b-bounded copying*. This means, it can be tested by inspecting the transducer rules. An alternative syntactic restriction, which implies our restriction of b-bounded copying, is the notion of *single use restriction*. It was originally invented in the context of attribute coupled grammars, but later generalized to stay-mtts in [EM99]. There it is shown for a restricted class of stay-mtts, that *semantic bounded copying* (called finite-copying) implies single use restriction, and hence syntactic bounded copying.

In [MOS05] Møller et al. propose a sound type checking algorithm based on an XSLT flow analysis which determines the possible outcomes of pattern matching operations. For the benefit of better performance, the algorithm deals with regular approximations of possible outputs.

Xact is a so-called *embedded* XML system with Java as host language [KMS04; KCM04]. XML data is modeled with *templates*, which are XML tree fragments with named gaps appearing in element contents or attributes. Values can be filled into these gaps in any order and at any time, and conversely, subtrees can be replaced by gaps in order to replace or even remove

data [MS05]. The static guarantees are obtained by a dataflow analysis based on the concept of *summary graphs* which approximatively track the operations on templates in the program. The type checker is conservative which means a program which passes the analysis cannot produce invalid XML documents at runtime.

Another embedded XML transformation language is XOBE [KL03]. This system uses *regular hedge expressions* as typing formalism, which (more or less) correspond to the class of regular tree languages. Based on explicit type annotations on every XML variable declaration, XOBE checks the subtype relationship for assignment statements. The type checker is not sound, i.e., in some cases programs that are not type-safe are classified as correct because the check routine makes somehow generous assumptions [MS05].

7

Implementation of a Type Checker

A common task in document processing, especially in XML processing, is to transform a document from one format into another format. A publication list stored as XML document, for example, has to be transformed into XHTML so that it can be viewed in a Web browser (cf. [Ley02]). Or a view of an XML database has to be computed by omitting some information from each entry or by restricting the set of returned entries to those specified in the view.

In general, such transformation are realized by an application written in a domain specific transformation language like XSLT [XSL99], XQuery [BCF+06], or fxt [Ber05], or it is implemented via a general purpose programming language with a more or less comfortable interface for XML processing like C [KR88], C♯ [ECM06], or Java [GJS96]. The computation, however, is similar for every implementation: traversing the input, the program collects all relevant parts and generates from these selected parts together with some new elements the output.

Since the produced output might be the input for a subsequent application, it is necessary to guarantee some pre-defined structural properties. Provided that the input is correct, the generation of XHTML documents from an XML bibliography should always result in a *valid* XHTML file, because otherwise the Web browser does not present the intended information. Guaranteeing structural properties might even be security relevant. Consider an XML file, which contains information about students including their names, marks, and the visited courses. For data protection reasons, the transformation results must not contain information so that marks can be mapped to a student's name. Even these "simple" examples show that structural guarantees for the output are an important task in XML processing.

In the previous chapters we have shown that transformations, which take only structural information into account, can be expressed by macro tree transducers and thus can be type checked. This means, it is possible to check whether each output document is of a specified structure or type. In order

to test our theoretical results for practical examples, we have developed a macro tree transducer suite which provides basic type checking algorithms.

7.1 A Type Checker Tutorial

We introduce the functionality of the type checker by means of our example macro tree transducer for moving mail marked as spam from the inbox into the trash folder (cf. Chapter 5). Recall that macro tree transducers work on ranked trees. Accordingly, we define the transducer to work on a binary tree representation of the mail file.

To keep the rule set small, we assume that `mail`-elements only contain the designated leaf label `e` representing the empty forest. Thus, we do not need rules for transforming `Body`-, `From`- or `To`-elements etc. Then the transformation is given by the following macro tree transducer:

```
1  q_init(Doc(l,r)) -> Doc(q_inbx(l),e())
2  q_inbx(Inbox(l,r)) -> Inbox(q_mail(l),q_trsh1(r,q_spam(l)))
3  q_trsh1(Trash(l,r),y) -> Trash(q_trsh2(l,y),e())
4  q_mail(Mail(l,r)) -> Mail(e(),q_mail(r))
5  q_mail(Spam(l,r)) -> q_mail(r)
6  q_mail(e()) -> e()
7  q_spam(Mail(l,r)) -> q_spam(r)
8  q_spam(Spam(l,r)) -> Spam(e(),q_spam(r))
9  q_spam(e()) -> e()
10 q_trsh2(Mail(l,r),y) -> Mail(e(),q_trsh2(r,y))
11 q_trsh2(Spam(l,r),y) -> Spam(e(),q_trsh2(r,y))
12 q_trsh2(e(),y) -> y
```

The transducer is defined in its normal syntax except that the right-arrow is replaced by " -> ". Since the gentle reader is already familiar with this transducer, we explain only those details, which illustrate the features of our implementation.

Line 1 defines a rule for function *q_init*. Function names are strings starting with a letter, followed by a sequence consisting of letters, numbers, and the underscore "_". Additionally, function names can end with "'". Legal function names are *q_inbx*, *Init*, *q_trsh*2, or q'.

The patterns are defined in the usual way, like `Inbox`(l,r) in line 2. Input symbols, here `Inbox`, also have to start with a letter and can consist of letters, numbers, and the underscore. The programmer does not need to define the used alphabet. It is automatically generated from all symbols occurring in the left- and right-hand sides. The rank of a symbol is also derived from the given pattern. Line 2, for example, defines symbol `Inbox` to be of rank 2, whereas `e`() in line 1 indicates that this symbol is of rank 0. The two forms `e`() and `e` for leaf labels are equivalent.

In order to make the rules more readable, input variables for identifying the subtrees of an input symbol can be named like functions. These variable names are only valid for one rule. This means, `Inbox`(l, r) (line 2) identifies the left son with l and the right son with r. Both names can be used in a function call in the right-hand side of that rule.

A function call is an expression like $q\_trsh1(r, q\_spam(l))$ in line 2. Function q_trsh1 is applied to input variable r. The expression $q\_spam(l)$ sets the accumulating parameter of q_trsh1 to the result of applying function q_spam to input variable l. Recall that we have to collect all mails marked as spam, i.e., mails inside a `Spam`-element, at this point of the transformation because later the mtt cannot go back to this node. If a function has more than one parameter, we have a comma-separated list of expressions, where the first is passed to the first parameter, the second expression to the second parameter and so on.

Line 3 gives an example, how parameters are used. Left of " -> "the parameter is named y, where again the name can freely be chosen as for function names. In the right-hand side, this name can be used to refer to the content of the according parameter. If a function has more than one parameter, each parameter has to be declared in a comma-separated list in the left-hand side. As mentioned above, parameters are passed by their order to the function. Consequently, the content of the first parameter is identified by the first declared name, the second parameter by the second name and so on. In the right-hand side of the rule in line 3, parameter y occurs in the parameter position of the function call $q\_trsh2(l, y)$. Parameter names can occur anywhere in a right-hand side, except as input argument for a function call.

To save the overhead of specifying initial functions explicitly, the type checker uses the first function rule to determine the initial function. In our example mtt, function q_init is the initial function.

In order to type check this macro tree transducer, we need a facility to define types. As mentioned in the preliminaries, we abstract from concrete type definition languages like DTD, XML Schema or Relax NG [BPSM$^+$04; FW04; Jam01], and prefer to specify input and output types as recognizable tree languages. Due to their one-to-one correspondence with regular tree automata, types are given as bottom-up automata (see Section 2.3) description in the following way:

```
1  p_Mail, Mail, p_Leaf, p_Leaf;
2  p_Mail, Mail, p_Leaf, p_Mail;
3  Error, Mail, p_Leaf, p_Spam;
4  Error, Mail, p_Leaf, Error;
5  p_Spam, Spam, p_Leaf, p_Leaf;
6  p_Spam, Spam, p_Leaf, p_Mail;
7  p_Spam, Spam, p_Leaf, p_Spam;
8  p_Spam, Spam, p_Leaf, Error;
9  p_Inbx, Inbox, p_Leaf, p_Trsh;
10 p_Inbx, Inbox, p_Mail, p_Trsh;
11 p_Trsh, Trash, p_Leaf, p_Leaf;
12 p_Trsh, Trash, p_Mail, p_Leaf;
13 p_Trsh, Trash, p_Spam, p_Leaf;
14 Error, Inbox, p_Spam, p_Trsh;
15 Error, Inbox, Error, p_Trsh;
16 p_Doc, Doc, p_Inbx, p_Leaf;
17 Error, Doc, Error, p_Leaf;
18 p_Leaf, e;.
19 Error
```

This description is partitioned into two parts: first, a rule set and second, the accepting states. The rule set is given by a ";" separated list of transition rules. The rule in line 1, for example:

```
p_Mail, Mail, p_Leaf, p_Mail;
```

has to be read as follows. The automaton is in state *p_Leaf* for the left son of the current node and in state *p_Mail* for the right son. If the label of the current node equals `Mail`, then the automaton assigns state *p_Mail* to the tree rooted at `Mail`. Rules for leafs, for example in line 15, have an empty list of states for the subtrees. The rule set is closed with a period. Then the comma-separated list of final or accepting states follows. In our example this list consists only of state *Error*.

The example automaton describes all incorrect outputs of our mailbox transformation. If a `Spam`-element occurs in the list of `Mail`-elements, then the automaton changes into state *Error* (line 3). Once the *Error*-state is reached, it is propagated through the list of `Mail`-elements (line 4) to the `Inbox`-element (line 15) and further to the root node (line 17). The rule in line 8 specifies that state *p_Spam* is assigned to a `Spam`-element, even if it is above a subtree whose state is *Error*. With this rule we avoid that the automaton assigns state *Error* to the content of the `Trash`-element because there `Mail`- and `Spam`-elements can be mixed. The accepting state is defined as *Error* because this exactly describes the tree structure we want to avoid with our transformation.

Assuming that we have saved the macro tree transducer in file "`mail.mtt`" and the automaton describing the erroneous outputs in file "`output.fta`", we can check our mtt with the following command:

```
check -top -mtt mail.mtt -out output.dfa
```

This command determines the output language of the macro tree transducer and checks whether the intersection between the computed output language and the given output type is empty (cf. Section 6.3). For our example mtt and the specification of the erroneous output documents, the checker returns that the intersection is empty. This means in particular that the transducer does not produce any document of the given type, thus we can conclude that the transformation is correctly designed.

Changing the output type so that it describes exactly all documents that can be produced by our macro tree transducer, then the checker returns that the intersection is not empty. This answer, however, says only that the transducer is able to generate a document conforming to the predefined type. The check against the set of incorrect outputs, on the other hand, says that the mtt by no means produces a document of the wrong type.

To type check our example macro tree transducer by computing the pre-image as described in Chapter 4, one can use the command:

```
check -pre -mtt mail.mtt -out output.dfa
```

Unfortunately, the automaton accepting the pre-image is so large that the runtime of the checker becomes inacceptable. The mtt has 6 functions:

$$q\_init, q\_inbx, q\_trsh1, q\_trsh2, q\_mail, \text{ and } q\_spam$$

with maximal one accumulating parameter. Let Q denote this set of functions. The tree automaton has a state set P consisting of 7 states:

$$p\_Mail, p\_Spam, p\_Inbx, p\_Trsh, p\_Doc, p\_Leaf, \text{ and } Error.$$

The constructed automaton accepting the pre-image has as states mappings d such that for every function q with k parameters, $d(q)$ is a mapping from P^k to P (see Section 4.3). Thus, the number of states of the pre-image automaton is $|Q \to P^k \to P|$ which equals $(|P|^{|P|^k})^{|Q|} = |P|^{|Q||P|^k}$. Since $|Q| = 6$ in our example and $|P| = 7$, we have $7^{7 \cdot 6} (= 7^42)$ states for the pre-image automaton. In order to compute the automaton, the checker has to determine for each of the 7^{42} states the according transitions. Due to the "non-elementary" size of the pre-image automaton, this type checking method works only for very small transducers.

7.2 The Implementation Details

In this section we briefly highlight some important implementation issues and solutions specific to our approach. We have developed an OCaml suite, which implements basic concepts for storing and manipulating *macro tree transducers*. Moreover, the suite provides prototype implementations of the two type checking methods:

(1) computing the pre-image of an mtt, and
(2) characterizing the output language of an mtt.

For our implementation of the macro tree transducer suite, we have chosen OCaml for several practical reasons. As a *functional* language OCaml is a natural choice for processing tree structured data. Compared to imperative approaches, the declarative programming technique of tree processing via recursive functions in functional languages is much clearer. This is an advantage, since the right-hand sides of the rules form trees.

A second reason is, that OCaml is a *strongly typed language*, which means that the compiler can ensure that each program executes without any type errors. It also helps to find programming errors as early as possible — which was also a motivation for our research in the field of XML transformations.

Moreover, OCaml provides the concepts of *modules*. A module or *structure* is a collection of data types and functions. A *signature* specifies which components of a structure are accessible from the outside, and with which type. It can be used to hide some components or the concrete implementation by restricting the types. Signatures are the interfaces for structures. *Functors* can be understand as functions from structures to structures. They are used to express parameterized structures. A structure A parameterized by a structure B is a functor F with a formal parameter B which returns the actual structure A itself. The expected type of parameter B is specified via a signature. The functor F can then be applied to one or several implementations $B_1, \ldots, B_n$ of B, yielding the corresponding structures $A_1, \ldots, A_n$. The concept of modules reuse and maintain the implemented functions.

Another advantage of OCaml is that the programs usually exhibit satisfactory performance. The compiler can produce both bytecode and native code, i.e., the programmer has the choice whether the compiled program has to be interpreted or is a standalone executable.

In the following, we first introduce the data structure for storing macro tree transducers. Then, we present the two type checking methods, and close with the implementation of the emptiness check and some considerations how to reduce the size of macro tree transducers.

7.2.1 Representation of Macro Tree Transducers

A macro tree transducer obviously can be represented as a collection of mutual recursive OCaml functions. The transformation induced by the following macro tree transducer

$$
\begin{array}{ll}
1 & q_0(\mathtt{R}(x_1,x_2)) \quad\;\; \to \mathtt{M}(q_1(x_1,\mathtt{E}'),q_2(x_2,E)) \\
2 & q_1(\mathtt{A}(x_1,x_2),\ y) \to \mathtt{A}'(q_1(x_1,\mathtt{D}(y)),q_2(x_2,\mathtt{D}(y))) \\
3 & q_1(\mathtt{B}(x_1,x_2),\ y) \to \mathtt{B}'(q_1(x_1,\mathtt{D}(y)),q_2(x_2,\mathtt{D}(y))) \\
4 & q_1(\mathtt{E},\qquad\ y) \to y \\
5 & q_2(\mathtt{A}(x_1,x_2),\ y) \to \mathtt{A}'(q_1(x_1,\mathtt{D}(y)),q_2(x_2,\mathtt{D}(y))) \\
6 & q_2(\mathtt{B}(x_1,x_2),\ y) \to \mathtt{B}'(q_1(x_1,\mathtt{D}(y)),q_2(x_2,\mathtt{D}(y))) \\
7 & q_2(\mathtt{E},\qquad\ y) \to E'
\end{array}
$$

which replaces each left leaf by a sequence of D's that is of the same length as the path from the root to that leaf, can be realized by the OCaml program

```
1  type t  = E | A of t * t | B of t * t
2  type r  = R of t * t
3  type t' = E' | A' of t' * t' | B' of t' * t' | D of t'
4  let rec q0 s = match s with
5     | R(x1,x2) -> M( q1 x1 E', q2 x2 E')
6  and q1 s y = match s with
7     | A(x1,x2) -> N( q1 x1 (D(y)), q2 x2 (D(y)) )
8     | B(x1,x2) -> O( q1 x1 (D(y)), q2 x2 (D(y)) )
9     | E        -> y
10 and q2 s y = match s with
11    | A(x1,x2) -> A'( q1 x1 (D(y)), q2 x2 (D(y)) )
12    | B(x1,x2) -> B'( q1 x1 (D(y)), q2 x2 (D(y)) )
13    | E        -> E'
```

The lines 1, 2 and 3 of the OCaml program define the tree structure on which the transducer is defined. Line 2 restricts the root label to R, which is done by the q_0-rule of the mtt. Each mtt function corresponds to an OCaml function. The functions are defined as mutual recursive functions (defined by the `let rec` – `and` expression) because the q_1- and q_2-rules call each other. The case expression of the form

$$
\begin{array}{l}
\mathtt{match}\ \mathit{expr} \\
\quad \mathtt{with}\ \mathit{pattern}_1\ \mathtt{->}\ \mathit{expr}_1 \\
\qquad\ \mid \mathit{pattern}_2\ \mathtt{->}\ \mathit{expr}_2 \\
\qquad\qquad\quad \vdots \\
\qquad\ \mid \mathit{pattern}_n\ \mathtt{->}\ \mathit{expr}_n
\end{array}
$$

matches the value of *expr* against patterns $\mathit{pattern}_1$ to $\mathit{pattern}_n$. If the matching against $\mathit{pattern}_i$ succeeds, the associated expression expr_i is evaluated, and its value becomes the value of the whole `match` expression. If several patterns match the value of *expr*, the one that occurs first in the `match` expression is selected [LDJ+04]. This means that in our example program the input tree `s` is matched against the patterns and the according action is performed.

Although this is a very elegant way to represent macro tree transducers and to study their outputs, we here prefer an abstract representation of mtts,

because we want to analyse some syntactic properties and construct other formats of the mtt rules. Therefore, we store each macro tree transducer in the following data structure:

```
1  type symbol = int
2  type state = int

3  type rhs = (symbol, state) RightHandSide.rhs
4  type action = Undefined | Det of rhs | Non_det of rhs list
5  type domain = None | OneOf of symbol list

6  type rule = symbol -> action
7  type rule_set = (state,(domain * rule)) Hashtbl.t
8  type parameter_table = (state,int) Hashtbl.t

9  type ('a,'b,'c) mtt = {
10    mutable q : 'a symbol_table;
11    mutable pars : parameter_table;
12    mutable sigma : 'b symbol_table;
13    mutable delta : 'c symbol_table;
14    mutable init : state list;
15    mutable rules : rule_set;
16    mutable sigma_arity : (symbol,int) table }
```

Symbols as well as functions (or states) are represented as integers, because then we can abstract from the concrete function or symbol types.

In order to be as flexible as possible, the type for macro tree trasducer is a *polymorphic* record where the type variable `'a` denotes the type of the function names, type variable `'b` stands for the type of the input symbols and type variable `'c` denotes the type of the output symbols. Each field of the record is declared to be `mutable` so that its value can be "physically" changed [LDJ+04].

Function names `q` (cf. line 10), input symbols `sigma` (line 12), and ouput symbols `delta` (line 13) are stored in a symbol table of the according type. A symbol table maps integers to the represented values and vice versa. Additionally, the number of parameters of each occurring function is stored in the field `pars` and the rank (or arity) of input symbols is stored in the field `sigma_arity`. The initial functions are stored as list in the field `init` (line 14).

The rules of the mtt are stored in a hashtable (line 15). The hashkeys are the function names of the mtt. The hashed entries are pairs consisting of the symbols for which this function is defined and a function mapping an input symbol to an action (cf. lines 6 and 7). An action is encoded as variant type and can either be undefined, or exactly one right-hand side expression, or — in the case of a nondeterministic mtt — a list of right-hand side expressions. Each right-hand side expression is also represented by a variant type, which consists of the following constructors:

```
1  type ('symbol,'state) rhs =
2     | Empty_rhs
3     | Parameter of int
4     | Symbol of 'symbol * int * ('symbol,'state) rhs list
5     | Call of 'state * int * int * ('symbol,'state) rhs list
```

A right-hand side can either be empty (line 2), or the j-th parameter (line 3), an output symbol followed by the list of its successors, or a function call together with the list of parameters. The integer type `int` in the `Symbol` case (line 4) stores the length of the successor list. The integers in the case of `Call` specify the input variable x_i and the number of parameters (i.e., the length of the parameter list).

A macro tree transducer, which is given by the command-line argument `-mtt` is automatically translated into this internal data representation.

7.2.2 Pre-Image Computation

In the context of type checking, one of the main properties of macro tree transducers is that taking pre-images effectively preserves recognizability. In other words, for a recognizable set T and an mtt M the pre-image $\tau_M^{-1}(T)$ is again recognizable (see Theorem 4.6). As we have seen, this justifies the following algorithm to check whether a macro tree transducer M computes a type-safe transduction with respect to an input type T_{in} and an outpt type T_{out}:

---

(1) Compute the complement $T = \overline{T_{\text{out}}}$ of T_{out};
(2) Compute $U = \tau_M^{-1}(T)$;
(3) **if** $U \cap T_{\text{in}} = \emptyset$
(4) **then return** "type-safe"
(5) **else return** "there are type errors"

The main point of this algorithm is the computation of the pre-image of the macro tree transducer. The automaton accepting it has to keep track of all possible inputs whose transformation results in a document of type T_{out}.

We concentrate here on deterministic macro tree transducers with one initial state. Let $M = (Q, \Sigma, \{q_0\}, R)$ be such an mtt, and let $A = (P, \Sigma, \delta, F_A)$ be a bottom-up tree automaton such that $\mathcal{L}(A) = T$, where T denotes the complement of the output type.

From M and A we construct an automaton $B = (D, \Sigma, \beta, F_B)$ recognizing the pre-image U. The state set D consists of all mappings d such that for every $q \in Q$ with k parameters, $d(q)$ is a mapping from P^k to P.

A naive solution determines for every symbol $\mathtt{a} \in \Sigma$ of rank $n \geq 0$, and every k-tuple $d_1, \ldots, d_n \in D^n$, a transition $\delta_{\mathtt{a}}(d_1 \ldots d_n) = d$ of automaton B.

State d of D is defined as follows: For every function $q \in Q_k$, and $p_1, \ldots, p_k \in P$, and mtt rule $q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t$ of M, let

$$d(q)(p_1, \ldots, p_m) = [t]\ \sigma\ \rho,$$

where

$$\sigma(x_j) = d_j \ \text{ for } j = 1, \ldots, n$$
$$\rho(y_i) = p_i \ \text{ for } i = 1, \ldots, k.$$

The definition of $[t]\ \sigma\ \rho$ can be found in the proof of Theorem 4.6. For this approach, the state set D has to be computed a priori and the determination of each transition is very time consuming. The number of states of D is

$$|Q \to P^k \to P| = |P|^{|Q| \cdot |P|^k}.$$

Then, for every symbol $\mathtt{a}$ of Σ with rank n one has to find for every n-tuple of states of D, the correct result state.

In order to improve this computation, it is better to construct the transitions of the pre-image automaton in a *demand-driven* way. Demand-driven means, the algorithm starts with a minimal initial set, and computes all other states and their transitions *by need* or *on demand*.

We begin the demand-driven computation of β with states for the initial function q_0. An accepting run of automaton A assigns to the root node of a possible output tree a state $p_0 \in F_A$. The root node of any output tree is produced by processing the initial function of M. Therefore, we start the construction of β with $d_0 : \{q_0 \mapsto p_0\}$. In order to obtain the transitions that have as result d_0, we have to inspect the rules of the initial function q_0. Instead of presenting the complex algorithm, which obtains from already computed states and transitions the next states together with their transitions, we try to give an intuition of the construction by maens of some characteristic examples.

For a transition $\delta_{\mathtt{b}}(p_1, p_2, p_3) = p_0$ of the output automaton, and a transformation rule

$$q_0(\mathtt{a}(x_1, x_2)) \to \mathtt{b}(q_1(x_1), q_2(x_2), q_3(x_1)),$$

we construct the following transition of β:

$$\beta_{\mathtt{a}}(\{q_1 \mapsto p_1 \wedge q_3 \mapsto p_3\}\{q_2 \mapsto p_2\}) = d_0,$$

where $\{q_1 \mapsto p_1 \wedge q_3 \mapsto p_3\} \in D$ is the state of the left son of the input symbol $\mathtt{a}$, and $\{q_2 \mapsto p_2\} \in D$ the state of the right son of $\mathtt{a}$. The state $\{q_1 \mapsto p_1 \wedge q_3 \mapsto p_3\}$ is obtained from all function calls in the right-hand side which are applied to x_1. Due to the transition $\delta_{\mathtt{a}}$, function call $q_1(x_1)$ must result in the state p_1 because its return value determines the first son of $\mathtt{b}$. Accordingly, q_2 is connected with the state p_2, and from function q_3 has to be obtained the state p_3. Each of the mappings from mtt functions to automata states, give rise to the next construction step.

For an mtt rule of the form

$$q(\mathtt{a}(x_1, x_2), y) \to \mathtt{c}(q_1(x_1, q_2(x_1, y))),$$

and a transition $\delta_{\mathtt{a}}(p_1) = p_2$, we obtain for all states $p_3 \in P$ the following transitions of the pre-image automaton:

$$\beta_{\mathtt{a}}(\{(q_1, p_3) \mapsto p_1 \wedge (q_2, p) \mapsto p_3\}) = \{(q, p) \mapsto p_1\}.$$

Here, the return value of the function q_1 depends on the return value of the function q_2, because the call $q_2(x_1, y)$ is in the parameter position of the call of function q_1. Therefore, the result state p_3 of q_2 is an input state for q_1.

For a transformation rule, which return only the value of its parameter

$$q(\mathtt{a}(x_1, x_2), y) \to y,$$

We obtain all transitions $\beta_{\mathtt{a}}(\_, \_) = \{(q, p) \mapsto p\}$ for all states $p \in P$, because the output is independent from the subtrees of the current input, but it depends on the content of the accumulating parameter y. Here, $\_$ stands for all states $d \in D$. Accordingly, an mtt rule of the form

$$q(\mathtt{a}(x_1, x_2), y) \to \mathtt{c}(y),$$

gives rise to the transition $\beta_{\mathtt{a}}(\_, \_) = \{(q, p) \mapsto p_1\}$ for all transitions $\delta_{\mathtt{c}}(p_2) = p_1$ of the output automaton.

During this demand-driven computation, we have to keep track of already computed states d and their transitions, in order to avoid that any state or transition is computed twice.

7.2.3 Implementation of Forward Type Inference

Forward type inference is a method to solve the type checking problem. For a given transformation F, the output language with respect to a predefined input type T_{in} is inferred:

$$F(T_{\text{in}}) = \{F(t) \mid t \in T_{\text{in}}\}.$$

To check whether F generates for correct inputs of T_{in} only correct outputs of a specified output type T_{out}, the intersection of the inferred output type and the complement of T_{out} is determined. For the complement set $\overline{T_{\text{out}}}$, we have

$$F(T_{\text{in}}) \cap \overline{T_{\text{out}}} = \emptyset \quad \text{if and only if} \quad \forall t \in T_{\text{in}} : F(t) \in T_{\text{out}}.$$

This means that we have to implement an efficient algorithm to infer the set of all outputs and to test whether the intersection of this inferred set and a predefined set of illegal outputs is empty or not.

In Section 6.3 we have shown that the output languages of linear stay macro tree transducers can be exactly characterized by context-free tree

grammars. However, instead of first constructing a context-free grammar, then determine the intersection grammar, and finally, check whether it is empty, we organize this method in two steps. Let $M = (Q, \Sigma, \{q_0\}, R)$ be a macro tree transducer and let $R = (P, \Sigma, F, \delta)$ be a finite tree automaton.

First, we construct from mtt M and automaton R a Datalog program P. The transition rules of R become the facts of P. For each rule

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

of the automaton, we construct a Datalog implication

$$q(\mathtt{Y}_0, \ldots, \mathtt{Y}_k) \texttt{ :- } \mathit{goals}[t]_{\mathtt{Y}_0}.$$

where $q/(k+1)$ is a predicate of arity $k+1$ (see page 99). The body $\mathit{goals}[t]_{\mathtt{Y}_0}$ is recursively defined over the structure of t

$$\begin{array}{ll} \mathit{goals}[y_j]_{\mathtt{Y}_0} & = \mathtt{X} = \mathtt{Y}_j \\ \mathit{goals}[\mathtt{a}(t_1, \ldots, t_m)]_{\mathtt{Y}_0} & = \delta(\mathtt{Y}_0, \mathtt{a}, \mathtt{X}_1, \ldots, \mathtt{X}_m)\,,\ \mathit{goals}[t_1]_{\mathtt{X}_1}\,, \ldots, \mathit{goals}[t_m]_{\mathtt{X}_m} \\ \mathit{goals}[q'(t_1, \ldots, t_n)]_{\mathtt{Y}_0} & = q'(Y_0, \mathtt{X}_1, \ldots, \mathtt{X}_n)\,, \mathit{goals}[t_1]_{\mathtt{X}_1}\,, \ldots, \mathit{goals}[t_n]_{\mathtt{X}_n} \end{array}$$

where the variables $\mathtt{X}_i$ in the last two rows are fresh (for all i of $1, \ldots, m$ and $1, \ldots, n$, respectively).

Second, we start to solve for each initial function q_0 of the mtt and each accepting state p_f of the automaton the query

$$\texttt{?- } q_0(p_f).$$

This query asks whether there exists a model for all clauses of program P such that this fact can be derived. In particular, if such a model exists, then the intersection of the output language and the given output type is not empty.

The query is solved by the so-called *SuccinctSolver* developed by Nielson, Nielson and Seidl [NS01; NNS+04]. This solver suite provides a state-of-the-art constraint solver. It computes teh least model of the alternation-free fragment of *Least Fixpoint Logic* in clausel form. Although this logic is more expressive than Datalog, it still allows for polynomial model-checking routines [NNS+04]. On the Datalog fragment, the solver achieves the best known theoretical bound.

The solver operates by recursively processing the original clause and by propagating an environment which collects bindings of the instantiated variables. Environments map variables to atoms but they are constructed in a lazy fashion meaning that variables may not have been given their values when introduced by quantifiers. Hence, environments are partial mappings consisting of pairs $x \mapsto a$, where x is a variable and a is either an atom or *unbound*. Whenever during the propagation of a partial environment, a quantified subexpression is encountered introducing variable x, the environment is extended by the binding $x \mapsto \mathit{unbound}$. New bindings for variables are obtained at a query as the most general unifiers. A detailed explanation of the solving algorithm can be found in [NS01; NSN02; NNS+04].

The OCaml suite provides an implementation of Datalog. Besides the common functionality it supports *explicit unification* and *explicit dis-unification* by means of operators "`=`" and "`\=`", respectively. The implementation provides integer arithmetics, where the builtin infix predicate "`is/2`" assigns the result of evaluating the expression on its right-hand side to a variable on its left-hand side. This means, "`X is 2*Y`" bounds the value of "`2*Y`" to variable `X`. Furthermore, the *SuccinctSolver* supports untyped lists by means of the builtin "`::/2`".

The *SuccinctSolver* solves queries like "`?-` $q_0(p_f)$`.`" in a top-down fashion. This solving strategy propagates selections that are made in the initial query into the rules as they are expanded. This makes the solver very elegant and fast [NSN02; NNS+04].

Compared with the approach presented in the previous section, forward type inference shows efficient performances on all tested transformations, due to the elegant solving techniques. The mtt implementation of the transformation, which moves spam mails into a trash folder (see Chapter 5), is checked in a few seconds, while computing the pre-image is very time-consuming and uses the complete memory of OCaml so that the checker stops with an out-of-memory exception.

7.2.4 Realization of Emptiness Check

As described in Section 4.3, it is decidable whether the transduction of a given stay-mtt or mtt is empty or not.

We briefly repeat the algorithm. Let therefore $M = (Q, \Sigma, Q_0, R)$ be a mtt. For every subset $S \subseteq Q$, we introduce a propositional variable $[S]$, where $[S] = \mathit{true}$ denotes the fact that every function $q \in S$ produces some output for an input tree, i.e.,

$$\exists t \in \mathcal{T}_\Sigma \forall q \in S : [\![q]\!](t) \neq \emptyset.$$

In particular, for the empty set $\{\}$ we have the implication $[\{\}] \Leftarrow \mathit{true}$. For every subset $S \subseteq Q$, we consider all implications

$$[S] \Leftarrow [S_1] \wedge \ldots \wedge [S_n]$$

for all selections of rules $q(\mathtt{a}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to t'_q$, $q \in S$, for the same symbol $\mathtt{a} \in \Sigma$. The sets S_i are defined as follows: $S_i = \{p \in Q \mid \exists q \in S : p(x_i, \ldots) \text{ occurs in } t'_q\}$.

Since the size of the constraint system is exponential in the size of the macro tree transducer M, we use a local constraint solver with dynamic tracking of variable dependencies [Sei06]. The solver is an OCaml functor

```
module SolverCon (X: Item) (D: Lattice)
```

where structure `X` specifies the variables of the constraint system, and structure `D` defines the lattice. Since we are interested only in the question whether

a function of the transducer produces some output or not, it suffices to use an implementation of the boolean lattice. Structure `X` implements the variables as lists of transducer states together with a compare function.

The advantage of this solver implementation is that it is not necessary to generate the complete constraint system a priori. Instead, the solver is started with an initial list of variables together with a function, which returns for every variable its dependencies. Whenever a new variable is introduced in a right-hand side of a constraint, it is added to the list of variables whose values have to be computed. Additionally, the variable of the left-hand side is added to the influence list of the new variable. Then, if a new value is obtained for that variable, every value of variables in its influence list is updated. In this way, the solver computes a model for the constraint system by dynamically adding new variable dependencies and updating the values of all variables, when the value of an influencing variable is changed.

7.2.5 Tuning the Macro Tree Transducer

As we have already seen, the number of functions of the macro tree transducer and the number of parameters has a great influence on the complexity of the pre-image automaton. Recall that the automaton accepting the pre-image has

$$|Q \to P^k \to P| = |P|^{|Q|\cdot|P|^k},$$

where Q denotes the set of mtt functions, P denotes the set of states of the output automaton, and k is the maximal number of parameters occurring in a function.

Therefore, it is of great benefit to keep the number of mtt functions and the number of parameters as small as possible. In this section, we present a method to determine *unused* functions, i.e., functions that are never called, and an technique to reduce the number of parameters.

Find Superfluous mtt Functions

Each macro tree transducer $M = (Q, \Sigma, Q_0, R)$ induces a graph which elucidates the calling behaviour of each function. The vertices of this graph G are the function names in Q. The edges are obtained as follows: Graph G contains an edge from a function q to a function q' only if q' occurs in a right-hand side of a q rule. Then, a function is superfluous, if it is not reachable from an initial state q_0 in the graph G. Since these functins are never called, they can be deleted together with all their rules. This method works for both deterministic and nondeterministic (stay) macro tree transducers.

Removing Superfluous Parameters

The key problem is to detect, if a function q of a macro tree transducer does not need all its accumulating parameters. This means, we are interested in

describing, whether a certain parameter of a state is ever needed during any computation.

Our implementation first normalizes the names of the parameters to integers. Then we compute the *dependency graph* G of the transducer. The dependency graph illustrates the calling behaviour of the mtt. It is constructed in the following way: The vertices are the function names of the mtt, and it contains an edge from function q to function q', if q' occurs in a right-hand side of a q-rule. For graph G we compute the strongly connected components, and sort these topologically. Then we propagate the used variables along the strongly connected components, where a variable is used, if it occurs in a right-hand side of one of the functions in a component. Each variable which is not propagated through the component graph, is considered to be superfluous and thus deleted from the rules.

Note that this method also works for nondeterministic stay macro tree transducers, but may change their semantics. Consider for example the following inside-out stay macro tree transducer:

$$\begin{aligned} q_0(\mathtt{a}(x_1,x_2)) &\to \mathtt{b}(q_1(x_1,q_2(x_2))) \\ q_1(\mathtt{a}(x_1,x_2),y) &\to \mathtt{b}(q_1(x_1,q_2(x_2))) \\ q_1(\mathtt{e}(x_1,x_2),y) &\to \mathtt{e}() \\ q_2(x_0) &\to q_2(x_0) \end{aligned}$$

The parameter y of function q_1 is obviously is superfluous, because it is never used. Removing it, turns the transducer into always terminating, while its current transformation never terminates. When function q_1 is called in the first line, function q_2 recurses infinitely on the second input argument. For inside-out evaluation of nested function calls, this leads to nonterminating transformations.

Even for macro tree transducers, the parameter reduction can change the induced transformation. Consider the following mtt:

$$\begin{aligned} q_0(\mathtt{a}(x_1,x_2)) &\to \mathtt{b}(q_1(x_1,q_2(x_2))) \\ q_0(\mathtt{e}(x_1,x_2),y) &\to \mathtt{e}() \\ q_1(\mathtt{a}(x_1,x_2),y) &\to \mathtt{b}(q_1(x_1,q_2(x_2))) \\ q_1(\mathtt{e}(x_1,x_2),y) &\to \mathtt{e}() \end{aligned}$$

Since function q_2 is undefined, removing the parameter will change the mtt from an erronous transformation into a total deterministic transformation.

7.3 Dealing with Infinite Alphabets

Our considerations are based on finite alphabets so far. Recall that stay macro tree transducers are defined to deal with an explicitely given finite alphabet Σ (cf. Definition 4.1). The original definitions of Engelfriet and Vogler even

specify a finite output alphabet Δ [EV85]. Implementing a tree transformation with stay-mtts consequently results in rules for each occurring symbol. Assume, for example, the transformation *Id*, which copies the input tree to the output. For alphabet $\Sigma = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$, it consists of the rules

$$\begin{array}{ll} q(\mathtt{a}(x_1, x_2)) & \to \mathtt{a}(q(x_1), q(x_2)) \\ q(\mathtt{b}(x_1, x_2)) & \to \mathtt{b}(q(x_1), q(x_2)) \\ q(\mathtt{c}(x_1, x_2)) & \to \mathtt{c}(q(x_1), q(x_2)) \\ & \vdots \\ q(\mathtt{z}(x_1, x_2)) & \to \mathtt{z}(q(x_1), q(x_2)) \\ q(\epsilon) & \to \epsilon. \end{array}$$

Although *Id* defines a simple transformation, the rule set consists of disproportinate many rules, which differ only in the transformed symbol.

A first improvement certainly is to define stay macro tree transducers to work on *sets of symbols*. Our transducer Id_1 has then two rules of the form

$$\begin{array}{ll} q(S(x_1, x_2)) & \to S(q(x_1), q(x_2)) \textit{ with } S \textit{ in } \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z} \\ q(\epsilon) & \to \epsilon. \end{array}$$

Whatever this transducer computes, Id_1 is not equivalent to *Id* because in each transformation step the output symbol can freely be chosen from Σ. Besides, using sets of symbols does not solve the problem that the transducer has to be redefined for every new alphabet.

An appropriate solution to both problems is to define stay macro tree transducers to work on *infinite* alphabets. Transducer Id_2 consequently consists of the two rules

$$\begin{array}{ll} q(\star(x_1, x_2)) & \to \star(q(x_1), q(x_2)) \\ q(\epsilon) & \to \epsilon, \end{array}$$

where the new symbol $\star$ in the left-hand sides indicates that the label of the currently inspected node is copied to every occurrence of $\star$ in the right-hand side. Transducer Id_2 obviously implements the same transformation as *Id* because every input symbol is mapped to the same symbol in the output. Moreover, Id_2 works for all binary trees independent of the underlying alphabet.

Clearly, this works best for stay-mfts, because they work on unranked trees. Accordingly, the symbol alphabet is defined as $\Sigma_0 = \Omega \cap \{\epsilon\}$, where Ω is the infinite alphabet and ϵ the symbol denoting the empty forest. Then, the working alphabet of a stay-mft is specified as $\Sigma = \Sigma_0 \cup \Delta$, where Δ contains all fixed symbols needed for the constructions, i.e., $\Delta = \{\star, \$, @, \sigma_i, \alpha_j, \bot\}$.

In order to specify an output type for a stay macro tree transducer with an infinite alphabet, we have to enhance our type model appropriately. Using the new symbol $\star$ in a tree automaton, leads to a nondeterministic automaton, since transitions for the $\star$ can be chosen for every symbol. A natural and nevertheless elegant way to describe the output language for a stay-mtt

working on an infinite alphabet is to use the *boolean algebra over finite and cofinite sets* of an infinite alphabet Σ_0. Thus, our new automata work on the alphabet

$$\Sigma = \{A \mid A \subseteq \Sigma_0\} \cup \{\overline{A} \mid A \subseteq \Sigma_0\}.$$

For $\sigma_1, \ldots, \sigma_n \in \Sigma_0$, let ~[$\sigma_1, \ldots, \sigma_n$] denote a confinite set, defining the set $\{\sigma \in \Sigma_0 \mid \sigma \notin \{\sigma_1, \ldots, \sigma_n\}\}$. The incorrect output for our running example of transforming a mail file can be defined as:

```
1  p_Mail, Mail, p_Leaf, p_Leaf;
2  p_Mail, Mail, p_Leaf, p_Mail;
3  p_Spam, Spam, p_Leaf, p_Leaf;
4  p_Spam, ~[Mail], p_Leaf, p_Mail;
5  p_Spam, ~[], p_Leaf, p_Spam;
6  p_Spam, Spam, p_Leaf, Error;
7  p_Inbx, Inbox, p_Leaf, p_Trsh;
8  p_Inbx, Inbox, p_Mail, p_Trsh;
9  Error, Inbox, p_Spam, p_Trsh;
10 p_Trsh, Trash, p_Leaf, p_Leaf;
11 p_Trsh, Trash, p_Mail, p_Leaf;
12 p_Trsh, Trash, p_Spam, p_Leaf;
13 p_Doc, Doc, p_Inbx, p_Leaf;
14 Error, Doc, Error, p_Leaf;
15 p_Leaf, e;.
16 Error
```

Line 4 specifies that every other symbol than `Mail` above a sequence of mails leads to state *p_Spam*. Line 5 uses the special confinite set ~[] denoting that every symbol matches. Thus, this transition is allowed for every symbol of Σ_0 and reflects the fact that if once in a sequence a `Spam` element occurs, state *q_spam* is assigned to the complete sequence. This automaton specifies all documents which should not be produced by the transformation. Although we have specified that Σ consists of the finite and cofinite subsets, it is sufficient to use single elements of Σ_0 to identify the symbols that are explicitely allowed.

For an alphabet like the *Unicode* alphabet, one possibly would prefer to use the algebra over *finite conjunctions of ranges*. This means, instead of enumerating each symbol of a set, one specifies an interval containing the appropriate symbols.

7.4 Dealing with Attributes

The W3C recommendation allows that every XML element can be equipped with *attributes* [BPSM+04] describing additional information about an element. An attribute is a pair consisting of a *name* and a *value*. They are listed along with the start-tag of an element, as for example in

⟨Chapter name = "Introduction" numbering = "arabic"⟩
  When Axel Thue ...
⟨/Chapter⟩

At first sight, attributes do not add to the expressiveness of XML. Therefore, Berlea and Neumann separatly suggest [Ber05; Neu99] to represent them by using dedicated element names, which are grouped under a special Attributes-element. This leads to the following representation:

⟨Chapter⟩
  ⟨Attributes⟩
    ⟨name⟩Introduction⟨/name⟩
    ⟨numbering⟩arabic⟨/numbering⟩
  ⟨/Attributes⟩
  ⟨Content⟩
    When Axel Thue ...
  ⟨/Content⟩
⟨/Chapter⟩

To avoid that the content of the Chapter-element is a mixture of normal text and XML elements (called *mixed content* in XML [BPSM+04]), we have introduced the Content-element, which now contains the original content of Chapter.

From a type checking point of view, attributes do change the expressiveness of XML, because the XML standard clearly states that the order of attribute specifications in a start-tag is *not* significant. Allowing attributes thus breaks our fundamental assumption of ordered trees. This means a type for an XML document in which attributes are used has to take into account that elements under the Attributes-element may occur in any order.

In order to deal with attributes, we have to enhance our typing mechanism by allowing that for sequences of siblings may only the occurring element names are given, while their order is left unspecified. Consequently, for attribute sequences, a type checker can only test whether the attribute elements are present. In [AMF+03] it is shown that type checking for a restricted from of TreeQL can be done in polynomial time in the presence of types specifying only cardinality constraints on the tags. Hence, we conjecture that enhancing our model by attributes (and unordered lists thereof) does not influence our type checking results.

7.5 Notes and References

Voigtländer and Kühnemann have a different view on macro tree transducers. They take as starting point that many functional programs with accumulating parameters are contained in the class of macro tree transducers. In [VK04] they present a transformation technique which can be used to solve

inefficiencies due to the creation and consumption of intermediate data structures. In [Voi01] Voigländer identifies restrictions for which two macro tree transducers can be composed. The mtt applied first has to be *non-copying*, and the second mtt has to be *weakly single-use*. Non-copying means that each accumulating parameter occurs at most once in a right-hand side. Weakly single-use is a property on the input, which says that each subtree of a node is processed at most once.

In order to test their ideas, they implemented the compostion algorithm in the Haskell$^+$ program transformation system [HMKV01]. Each macro tree transducer is directly embedded into the program code, whereas we prefered to represent mtts by means of a data structure. Accordingly, they need a sophisticated algorithm to detect the transducer definition inside a program.

Another difference between the two systems is that Haskell$^+$ is a *lazy* programming language, which means the evaluation of a function application is postponed until its value is needed. Thus, their transducers also evaluate in outside-in order in the presence of nested function calls. On the other hand, they assume each macro tree transducer to be total deterministic, and thus, the order of evaluation does not influence the transformation result. Our transducer suite can be equipped with different evaluation strategies. Currently, the usual inside-out mode is implemented.

In a future version of our macro tree transducer suite, we want to generalize the transducer model to *stay* macro tree transducer, because then, we are able to add regular look-ahead (cf. 4.5). In order to implement a more or less comfortable frontend, reading TL programs, we want to enhance our suite in such a way, that it can handle "unknown" symbols.

Furthermore, we want to improve the run-time of the pre-image computation by integrating the input type. Then, it is possible to stop the automata construction as early as the first inconsistency with the input type is detected.

Another problem that is worth to be solved, is to give detailed informations about the occurring type errors. For the user of our suite it would be of great benefit, if the system could give hints, which functions cause the errors.

8

Conclusion

We have presented several techniques to type check XML transformations. Type checking guarantees that for correct input documents, only correct output documents are produced. Since XML can be considered as the de facto standard for data exchange, type checking can help to ensure that only the intended information is transfered.

In order to be independent from the syntactic sugar of existing transformation languages, and to reduce the provided functionality to a transformation core for which type checking is still decidable, we introduced the transformation language TL. It subsumes the so-called tree transformation core of most of the existing XML transformation languages. TL is a rule-based domain specific language for processing XML data. Rule selection is done via monadic second-order patterns. The navigation on the input is also realized by (binary) monadic second-order formulas. TL rules have access to the context via accumulating parameters.

The key idea of type checking TL was to compile a given TL program into at most three stay macro tree transducers. Therefore, we had to translate the global selections, which are inherent in the select patterns, into local movements to traverse the complete input for all nodes matching a pattern. Since the obtained intermediate transducer contains up moves, we had to simulate these by a parameter construction, in order to get three stay macro tree transducers performing the transformation of the TL program. Then, type checking can be done via inverse type inference, which means that the pre-image of the transducer composition is computed with respect to the complement of a given output language. The TL program is type-safe only if the intersection of the pre-image and a given input language is empty.

Since this method for type checking XML transformations, has even for small programs an exorbitant run-time, we developed an alternative approach, which allows at least for a large class of transformations that they can be type checked in polynomial time. This class contains all transformations which can be expressed by a single stay macro tree transducer, which copies its input only a bounded number of times. Here, type checking can be

done by exact characterizations of the ouptput languages of the tree transducers.

We implemented most of the ideas in a macro tree transducer suite, which provides basic manipulation functions as well as different type checking solutions. Probably, the forward type inference by means of exact characterizations of the ouput languages of mtts is the most innovative method. It yields acceptable performances even for larger programs (with several states and parameters).

Due to this positive result, a natural question is, whether this technique of forward type inference can be generalized to compositions of (stay) macro tree transducers. Engelfriet and Vogler show in [EV88] that the so-called n-level tree transducer is equivalent to the n-fold composition of macro tree transducers. "*n-level tree transducers combine the features of n-level tree grammars and of top-down tree transducers in the sense that the derivations of the grammars are syntax-directed by input trees*" [EV88]. Hence, the n-level tree transducer is a single transformation model expressing the functionality of compositions of mtts. Moreover, Engelfriet and Vogler show that the class of regular tree languages is closed under the inverse of high level tree transductions, and thus, type checking is decidable. These are hints that the interesting question, whether compositions of mtts can be type checked by characterizations of their output languages, might be positively answered. As a consequence, it would be possible to type check TL in a more direct way.

A

Proofs

A.1 Proof of Theorem 3.7

Before we can prove the equivalence of the operational and the denotational semantics, we first show that each derivation can be decomposed into several derivations of shorter length.

Lemma A.1 Let $P = (R, A_0)$ be a TL program, let f be an input forest, and let $Y = \{y_1, \ldots, y_k\}$ denote a set of accumulating parameters. Let $w_0, w_1, \ldots, w_k \in \mathcal{DT}(Q, \Sigma, f)$ be a derived term with

$$w = w_0[w_1/y_1, \ldots, w_k/y_k],$$

then there exists a derivation Π, such that

$$\Pi : w \Rightarrow_f^* s$$

with $s = s_0[s_1/r_1, \ldots, s_n/r_n]$ for some nodes $r_1, \ldots, r_n$ of s_0, where

$$\begin{aligned} &\Pi_0 : w_0 \Rightarrow_f^* s_0 \\ &\Pi_i : w_{j_i} \Rightarrow_f^* s_i \quad \text{for } j = 1, \ldots, k \text{ and } i = 1, \ldots, n \end{aligned}$$

with $\sum_{i=0}^{n} |\Pi_i| = |\Pi|$.

Proof. We prove the assertion by induction on the length of Π_0.

(i) $|\mathbf{\Pi_0}| = \mathbf{0}$, then w_0 does not contain any function calls. Thus, the derivation is composed of the derivations of the w_i and the assertion follows.

(ii) $|\mathbf{\Pi_0}| > \mathbf{0}$, then we proceed by induction on the structure of w_0.
   a) for $w_0 = \epsilon$ and $w_0 = y$ the claim trivially follows;
   b) $w_0 = w_1' \, w_2'$, then the claim follows by induction hypothesis for w_1' and w_2';
   c) $w_0 = \langle \mathtt{a} \rangle w' \langle / \mathtt{a} \rangle$, then by induction hypothesis for w' the assertion follows;
   d) $w_0 = q(v, w_1', \ldots, w_m')$, then there exists a derivation

$$q(v, w_1', \ldots, w_m') \Rightarrow_f w_{0_0}[w_1'/y_1, \ldots, w_m'/y_m].$$

This means, we have a derivation Π' with

$$\begin{aligned} \Pi' : \quad & q(v, w_1'[w_1/y_1, \ldots, w_k/y_k], \ldots, w_m'[w_1/y_1, \ldots, w_k/y_k]) \Rightarrow_f \\ & w_{0_0}[(w_1'[w/y]_k)/y_1, \ldots, (w_m'[w/y]_k)/y_k] \\ = \; & w_{0_0}[w_1'/y_1, \ldots, w_m'/y_m][w_1/y_1, \ldots, w_k/y_k] \end{aligned}$$

where $[w/y]_k$ abbreviates $[w_1/y_1, \ldots, w_k/y_k]$. By applying the induction hypothesis to $w_{0_0}[w_1'/y_1, \ldots, w_m'/y_m]$, there is a derivation $\Pi' : w_{0_0}[w_1'/y_1, \ldots, w_m'/y_m] \Rightarrow_f^* s$, with derivations $\Pi_0' : w_{0_0} \Rightarrow_f^* s_0$ and $\Pi_i' : w_{j_i}' \Rightarrow_f^* s_i$ with $\sum_{i=0}^{m} |\Pi_i'| = |\Pi'|$.

Since Π' is a part of Π, we have

$$\begin{aligned}\Pi: \quad & q(v, w_1'[w_1/y_1, \ldots, w_k/y_k], \ldots, w_m'[w_1/y_1, \ldots, w_k/y_k]) \\ & \Rightarrow_f w_{0_0}[(w_1'[w/y]_k)/y_1, \ldots, (w_m'[w/y]_k)/y_k] \\ & \Rightarrow_f^* s_0[s_1/r_1, \ldots, s_n/r_n = s\end{aligned}$$

with $|\Pi| = 1 + |\Pi'| = 1 + \sum_{i=0}^{m} |\Pi_i'|$, which is what we wanted to prove.

This completes the proof of Lemma A.1 □

Now, we are able to show that the two characterizations of the transduction induced by a TL program coincide. Assume we are given a TL program $P = (R, A_0)$ and let r denote the root node of the input forest f. In order to show that the fixpoint and the operational semantics are equivalent for TL, we have to show the two implications:

$$\begin{aligned}& s \in [\![A_0]\!]_{f,\mu}\, v \text{ implies that } [A_0]_{f,v} \Rightarrow_{f,\mu}^* s \\ & [A_0]_{f,v} \Rightarrow_{f,\mu}^* s \text{ implies that } s \in [\![A_0]\!]_{f,\mu}\, v\end{aligned}$$

We first show for an arbitrary action that the implication holds for outside-in evaluation order in presence of parameters.

A.1.1 Outside-in Evaluation

(1) In order to show that

$$s \in [\![A]\!]_f^i\, v \quad \text{implies} \quad [A]_{f,v} \Rightarrow_f^* s \tag{A.1}$$

we proceed by an induction on the number i of fixpoint iterations.

a) $\mathbf{i = 0}$: Then the assertion follows.

b) $\mathbf{i > 0}$: We proceed by structural induction on A.

i. For $A = \epsilon$ and $A = y$, the claim follows.

ii. For $A = \langle \mathtt{a} \rangle A_1 \langle / \mathtt{a} \rangle$, then $s = \langle \mathtt{a} \rangle s' \langle / \mathtt{a} \rangle \in [\![A]\!]_f^i\, v$. Applying the induction hypothesis we obtain that there exists a derivation $[A_1]_{f,v} \Rightarrow_f^* s'$, and thus $[A]_{f,v} \Rightarrow_f^* \langle \mathtt{a} \rangle s' \langle / \mathtt{a} \rangle$, which we wanted to prove.

iii. If $A = A_1\, A_2$ then the assertion follows also by induction hypothesis for A_1 and A_2.

iv. For $A = q(\psi, A_1, \ldots, A_k)$, there are nodes $v_1 < \ldots < v_m \in \mathcal{N}(f)$ such that $(v, v_i) \models_f \psi$. The i-th fixpoint iteration results in

$$\begin{aligned}[\![A]\!]_f^i\, v &= ([\![q]\!]_f^i\, v_1)[y_1 \leftarrow [\![A_1]\!]_f^i, \ldots, y_k \leftarrow [\![A_k]\!]_f^i] \cdots \\ &\quad ([\![q]\!]_f^i\, v_m)[y_1 \leftarrow [\![A_1]\!]_f^i, \ldots, y_k \leftarrow [\![A_k]\!]_f^i] \\ &= ([\![t_{i_1}]\!]_f^{i-1}\, v_1)[y_1 \leftarrow [\![A_1]\!]_f^i, \ldots, y_k \leftarrow [\![A_k]\!]_f^i] \cdots \\ &\quad ([\![t_{i_m}]\!]_f^{i-1}\, v_m)[y_1 \leftarrow [\![A_1]\!]_f^i, \ldots, y_k \leftarrow [\![A_k]\!]_f^i],\end{aligned}$$

where each t_{i_j} is a right-hand side of a matching q-rule. An expression $([\![q]\!]^i_f\ v_1)[y_1 \leftarrow [\![A_1]\!]^i_f, \ldots, y_k \leftarrow [\![A_k]\!]^i_f]$ denotes the simultaneous OI-substitution of the semantics of the actual parameters for the formal parameters. Thus, $s = s_1 \cdots s_m$ is an element of the i-th fixpoint iteration of $[\![A]\!]^i_f\ v$, where for $j = 1, \ldots, m$, each $s_j = s'_{0_j}[s'_{1_j}/r_{1_j}, \ldots, s'_{l_j}/r_{l_j}]$ with $s'_{\iota_j} \in [\![A_\kappa]\!]^i_f\ v$ if the label at r_ι is y_κ. Now, we have to construct a derivation for s in order to show the assertion. We have:

$$\begin{aligned}[A]_{f,v} &= [q(\psi, A_1, \ldots, A_k)]_{f,v} \\ &= q(v_1, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \cdots \\ &\quad q(v_m, [A_1]_{f,v}, \ldots, [A_k]_{f,v})\end{aligned}$$

For all nodes v_j ($j = 1, \ldots, m$), we obtain

$$q(v_j, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \Rightarrow_f [t_{i_j}]_{f,v_j}[[A_1]_{f,v}/y_1, \ldots, [A_k]_{f,v}/y_k]$$

where t_{i_j} is a right-hand side for q. Since each t_{i_j} is evaluated in the $(i-1)$st iteration, we can apply the induction hypothesis for $[\![t_{i_j}]\!]^{i-1}_f\ v_j$ and get:

$$\Pi_0[t_{i_j}]_{f,v_j} \Rightarrow^*_f s'_{0_j}.$$

Since the A_κ ($\kappa = 1, \ldots, k$) are smaller terms than A, we obtain by applying the induction hypothesis to $[\![A_\kappa]\!]^i_f\ v$:

$$\Pi_\kappa : [A_\kappa]_{f,v} \Rightarrow^*_f s'_1.$$

Assembling all subderivations, we can construct for each function call $q(v_j, A_1, \ldots, A_k)$ ($j = 1, \ldots, m$) a derivation for s_j:

$$\begin{aligned}\Pi_j : \quad & q(v_j, A_1, \ldots, A_k) \\ & \Rightarrow_f [t_{i_j}]_{f,v_j}[[A_1]_{f,v}/y_1, \ldots, [A_k]_{f,v}/y_k] \\ & \Rightarrow^*_f s'_{0_j}[s'_{1_j}/r_{1_j}, \ldots, s'_{l_j}/r_{l_j}] = s_j\end{aligned}$$

The complete derivation for $s = s_1 \cdots s_m$ is composed from the derivations for the s_j, which is what we wanted to prove.

This ends the proof for implication (A.1).

(2) We prove the second implication

$$[A]_{f,v} \Rightarrow^*_f s \quad \text{implies} \quad s \in [\![A]\!]_f\ v \tag{A.2}$$

by an induction on the length n of the derivation.

a) $\mathbf{n} = \mathbf{0}$: The expression A does not contain expressions of the form $q(v, s_1, \ldots, s_k)$, and thus the assertion follows.

b) $\mathbf{n} > \mathbf{0}$: We proceed by structural induction on A.

i. For $A = \epsilon$ and $A = y$, the claim follows.
ii. For $A = \langle \mathtt{a} \rangle A_1 \langle / \mathtt{a} \rangle$, then $[A]_{f,v} \Rightarrow_f^* \langle \mathtt{a} \rangle s' \langle / \mathtt{a} \rangle = s$. Applying the induction hypothesis we obtain that $s' \in [\![A_1]\!]_f^i\ v$. Thus, we get $s \in \{\langle \mathtt{a} \rangle s_1 \langle / \mathtt{a} \rangle \mid s_1 \in [\![A_1]\!]_f\ v\} = [\![A]\!]_f^i\ v$, which we wanted to prove.
iii. If $A = A_1\ A_2$ then the assertion follows also by induction hypothesis for A_1 and A_2.
iv. For $A = q(\psi, A_1, \ldots, A_k)$, we have

$$\begin{aligned}[A]_{f,v} &= [q(\psi, A_1, \ldots, A_k)]_{f,v} \\ &= q(v_1, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \cdots \\ &\quad\ q(v_m, [A_1]_{f,v}, \ldots, [A_k]_{f,v})\end{aligned}$$

For each function call $q(v_1, [A_1]_{f,v}, \ldots, [A_k]_{f,v})$ there exists a derivation

$$\begin{aligned}\Pi_j : \quad & q(v_j, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \\ \Rightarrow_f\ & [t_{i_j}]_{f,v_j}[[A_1]_{f,v}/y_1, \ldots, [A_k]_{f,v}/y_k] \\ \Rightarrow_f^*\ & s'_{0_j}[s'_{1_j}/r_{1_j}, \ldots, s'_{l_j}/r_{l_j}] = s_j\end{aligned}$$

By Lemma A.1 we have:

$$\begin{aligned}\Pi_0 &: [t_{i_j}]_{f,v_j} \Rightarrow_f^* s'_{0_j} \\ \Pi_\iota &: [A_{\kappa_\iota}]_{f,v} \Rightarrow_f^* s'_{\iota_j}\end{aligned}$$

with $|\Pi_j| = \sum_{\iota=0}^{l} |\Pi_\iota|$. Since they have a shorter derivation, we can apply the induction hypothesis to Π_0 and Π_ι, and obtain:

$$s'_{0_j} \in [\![t_{i_j}]\!]_f\ v_j \quad \text{and} \quad s'_{\iota_j} \in [\![A_{\kappa_\iota}]\!]_f\ v.$$

Furthermore,

$$\begin{aligned}& s'_{0_j}[s'_{1_j}/r_{1_j}, \ldots, s'_{l_j}/r_{l_j}] \\ \in\ & \{t[t'_{1_j}/r_{1_j}, \ldots, t'_{l_j}/r_{l_j}] \mid t \in [\![t_{i_j}]\!]_f\ v_j, t'_{\mu_j} \in [\![A_{\kappa_\iota}]\!]_f\ v \text{ if } \lambda(r_{\mu_j}) = y_\kappa\} \\ =\ & ([\![t_{i_j}]\!]_f\ v_j)[y_1 \leftarrow [\![A_1]\!]_f\ v, \ldots, y_k \leftarrow [\![A_k]\!]_f\ v],\end{aligned}$$

which is contained in the denotational semantics of

$$([\![q]\!]_f\ v_j)[y_1 \leftarrow [\![A_1]\!]_f\ v, \ldots, y_k \leftarrow [\![A_k]\!]_f\ v],$$

The claim follows for s by analoguos considerations for all function evaluations on nodes $v_1, \ldots, v_m$.

This ends the proof for implication (A.2).

The equivalence of the characterization by means of a derivation relation and the inductive characterization follows by taking the initial action for A.

A.1.2 Inside-out Evaluation

(1) In order to show that

$$s \in [\![A]\!]^i_f\, v \quad \text{implies} \quad [A]_{f,v} \Rightarrow^*_f s \tag{A.3}$$

we proceed by an induction on the number i of fixpoint iterations.

a) $\mathbf{i = 0}$: Then the assertion follows.

b) $\mathbf{i > 0}$: We proceed by structural induction on A.

i. For $A = \epsilon$ and $A = y$, the claim follows.

ii. For $A = \langle \mathtt{a} \rangle A_1 \langle / \mathtt{a} \rangle$ and $A = A_1\, A_2$ the assertion follows by induction hypothesis.

iii. For $A = q(\psi, A_1, \ldots, A_k)$, there are nodes $v_1 < \ldots < v_m \in \mathcal{N}(f)$ such that $(v, v_i) \models_f \psi$. The i-th fixpoint iteration results in

$$\begin{aligned}
[\![A]\!]^i_f\, v &= ([\![q]\!]^i_f\, v_1)[[\![A_1]\!]^i_f/y_1, \ldots, [\![A_k]\!]^i_f/y_k] \cdots \\
&\quad\ ([\![q]\!]^i_f\, v_m)[[\![A_1]\!]^i_f/y_1, \ldots, [\![A_k]\!]^i_f/y_k] \\
&= ([\![t_{i_1}]\!]^{i-1}_f\, v_1)[[\![A_1]\!]^i_f/y_1, \ldots, [\![A_k]\!]^i_f/y_k][f[v]/x_1] \cdots \\
&\quad\ ([\![t_{i_m}]\!]^{i-1}_f\, v_m)[[\![A_1]\!]^i_f/y_1, \ldots, [\![A_k]\!]^i_f/y_k][f[v]/x_1],
\end{aligned}$$

where each t_{i_j} is a right-hand side of a matching q-rule. An expression $([\![q]\!]^i_f\, v_1)[[\![A_1]\!]^i_f/y_1, \ldots, [\![A_k]\!]^i_f/y_k]$ denotes the simultaneous IO-substitution of the semantics of the actual parameters for the formal parameters. Thus, $s = s_1 \cdots s_m$ is an element of the i-th fixpoint iteration of $[\![A]\!]^i_f\, v$, where for $j = 1, \ldots, m$, each $s_j = s'_{0_j}[s'_{1_j}/r_{1_j}, \ldots, s'_{l_j}/r_{l_j}]$ with $s'_{\iota_j} \in [\![A_\kappa]\!]^i_f\, v$ if the label at r_ι is y_κ. We have:

$$\begin{aligned}
[A]_{f,v} &= [q(\psi, A_1, \ldots, A_k)]_{f,v} \\
&= q(v_1, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \cdots \\
&\quad\ q(v_m, [A_1]_{f,v}, \ldots, [A_k]_{f,v})
\end{aligned}$$

For all nodes v_j ($j = 1, \ldots, m$), we obtain

$$q(v_j, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \Rightarrow_f [t_{i_j}]_{f,v_j}[[A_1]_{f,v}/y_1, \ldots, [A_k]_{f,v}/y_k]$$

where t_{i_j} is a right-hand side for q. Since each t_{i_j} is evaluated in the $(i-1)$st iteration, we can apply the induction hypothesis for $[\![t_{i_j}]\!]^{i-1}_f\, v_j$ and get:

$$\Pi_0[t_{i_j}]_{f,v_j} \Rightarrow^*_f s'_{0_j}.$$

Since the A_κ ($\kappa = 1, \ldots, k$) are smaller terms than A, we obtain by applying the induction hypothesis to $[\![A_\kappa]\!]^i_f\, v$:

$$\Pi_\kappa : [A_\kappa]_{f,v} \Rightarrow^*_f s'_1.$$

We can construct for each function call $q(v_j, A_1, \ldots, A_k)$ ($j = 1, \ldots, m$) a derivation for s_j:

$$\begin{aligned}\Pi_j : \quad & q(v_j, A_1, \ldots, A_k) \\ & \Rightarrow_f [t_{i_j}]_{f,v_j}[[A_1]_{f,v}/y_1, \ldots, [A_k]_{f,v}/y_k] \\ & \Rightarrow_f^* s'_{0_j}[s'_{1_j}/r_{1_j}, \ldots, s'_{l_j}/r_{l_j}] = s_j\end{aligned}$$

The complete derivation for $s = s_1 \cdots s_m$ is composed from the derivations for the s_j, which is what we wanted to prove.

This ends the proof for implication (A.3).

(2) We prove the second implication

$$[A]_{f,v} \Rightarrow_f^* s \quad \text{implies} \quad s \in [\![A]\!]_f\, v \tag{A.4}$$

by an induction on the length n of the derivation.

a) $\mathbf{n} = \mathbf{0}$: The expression A does not contain expressions of the form $q(v, s_1, \ldots, s_k)$, and thus the assertion follows.

b) $\mathbf{n} > \mathbf{0}$: We proceed by structural induction on A.

i. For $A = \epsilon$ and $A = y$, the claim follows.

ii. For $A = \langle \mathtt{a} \rangle A_1 \langle / \mathtt{a} \rangle$ and $A = A_1\, A_2$ the assertion follows also by induction hypothesis.

iii. For $A = q(\psi, A_1, \ldots, A_k)$, we have

$$\begin{aligned}[A]_{f,v} &= [q(\psi, A_1, \ldots, A_k)]_{f,v} \\ &= q(v_1, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \cdots \\ &\quad\ q(v_m, [A_1]_{f,v}, \ldots, [A_k]_{f,v})\end{aligned}$$

For each function call $q(v_1, [A_1]_{f,v}, \ldots, [A_k]_{f,v})$ there exists a derivation

$$\begin{aligned}\Pi_j : \quad & q(v_j, [A_1]_{f,v}, \ldots, [A_k]_{f,v}) \\ & \Rightarrow_f [t_{i_j}]_{f,v_j}[[A_1]_{f,v}/y_1, \ldots, [A_k]_{f,v}/y_k] \\ & \Rightarrow_f^* s'_{0_j}[s'_{1_j}/r_{1_j}, \ldots, s'_{l_j}/r_{l_j}] = s_j\end{aligned}$$

By Lemma A.1 we have:

$$\begin{aligned}\Pi_0 &: [t_{i_j}]_{f,v_j} \Rightarrow_f^* s'_{0_j} \\ \Pi_\iota &: [A_{\kappa_\iota}]_{f,v} \Rightarrow_f^* s'_{\iota_j}\end{aligned}$$

with $|\Pi_j| = \sum_{\iota=0}^{l} |\Pi_\iota|$. Since they have a shorter derivation, we can apply the induction hypothesis to Π_0 and Π_ι, and obtain:

$$s'_{0_j} \in [\![t_{i_j}]\!]_f\, v_j \quad \text{and} \quad s'_{l_j} \in [\![A_{\kappa_\iota}]\!]_f\, v.$$

Furthermore,

$$
\begin{aligned}
& s'_{0_j}[s'_{1_j}/r_{1_j}, \ldots, s'_{l_j}/r_{l_j}] \\
\in\ & \{t[t'_{1_j}/r_{1_j}, \ldots, t'_{l_j}/r_{l_j}] \mid t \in [\![t_{i_j}]\!]_f\ v_j, t'_{\mu_j} \in [\![A_{\kappa_i}]\!]_f\ v \text{ if } \lambda(r_{\mu_j}) = y_\kappa\} \\
=\ & ([\![t_{i_j}]\!]_f\ v_j)[[\![A_1]\!]_f\ v/y_1, \ldots, [\![A_k]\!]_f\ v/y_k] \\
\subseteq\ & ([\![q]\!]_f\ v_j)[[\![A_1]\!]_f\ v/y_1, \ldots, [\![A_k]\!]_f\ v/y_k].
\end{aligned}
$$

The claim follows for s by analoguos considerations for all function evaluations on nodes $v_1, \ldots, v_m$.

This ends the proof for implication (A.4).

Again, the equivalence of the characterization by means of a derivation relation and the inductive characterization follows by taking the initial action for A.

References

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web*. Morgan Kaufmann, 2000.

[Adl01] Sharon Adler et al. Extensible Stylesheet Language (XSL) Version 1.0. W3C Recommendation, World Wide Web Consortium, October 2001. Available online http://www.w3.org/TR/2001/REC-xsl-20011015/.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, Reading, Massachusetts, 1995.

[AMF+01] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. Typechecking XML Views of Relational Databases. In *16th IEEE Symposium on Logic in Computer Science (LICS)*, pages 421–430, 2001.

[AMF+03] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. Typechecking XML Views of Relational Databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003.

[AU71] Alfred V. Aho and Jeffrey D. Ullman. Translations on a Context-Free Grammar. *Information and Control*, 19(5):439–475, 1971.

[BBC+06] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon, editors. XML Path Language (XPath) 2.0. W3C Candidate Recommendation, World Wide Web Consortium, June 2006. Available online http://www.w3.org/TR/xpath20.

[BCF03] Véronique Benzaken, Guiseppe Castagna, and Alain Frisch. CDuce: An XML-Centric General-Purpose Language. In *8th ACM International Conference on Functional Programming (ICFP)*, pages 51–63. ACM Press, 2003.

[BCF+06] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniel Florescu, Jonathan Robie, and Jérôme Siméon, editors. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation, World Wide Web Consortium, June 2006. Available online http://www.w3.org/TR/2006/CR-xquery-20060608/.

[BE00] Roderick Bloem and Joost Engelfriet. A Comparison of Tree Transductions defined by Monadic Second Order Logic and by Attribute Grammars. *Journal of Computer and System Sciences (JCSS)*, 61(1):1–50, 2000.

[Ber05] Alexandru Berlea. *Efficient XML Processing with Tree Automata*. PhD thesis, Institut für Informatik der Technischen Universität, München, 2005.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau, editors. Extensible Markup Language (XML) 1.0. W3C Recommendation, World Wide Web Consortium, February 2004. Available online http://www.w3.org/TR/2000/REC-xml-20040204.

[Bra69] Walter S. Brainerd. Tree Generating Regular Systems. *Information and Control*, 14(2):217–231, 1969.

[BS02] Alexandru Berlea and Helmut Seidl. fxt – A Transformation Language for XML Documents. *Journal of Computing and Information Technology*, 10(1):19–35, 2002.

[BW04] Anne Brüggemann-Klein and Derick Wood. Balanced Context-Free Grammars, Hedge Grammars and Pushdown Caterpillar Automata. In *Extreme Markup Languages*, Montréal, Quebec, 2004. Available online http://www.mulberrytech.com/Extreme/Proceedings.

[BW05] Mikołai Bojańczyk and Igor Walukiewicz. Unranked Tree Algebra. Technical report, Wydział Matematyki, Informatyki i Mechaniki, Uniwersytetu Warszawskiego (Warsaw University), 2005.

[CD99a] J. Clark and S. DeRose, editors. *XML Path Language (XPath) 1.0*. W3C, November 1999. Available online http://www.w3.org/TR/xpath.

[CD99b] James Clark and Steve DeRose, editors. XML Path Language (XPath) 1.0. W3C Recommendation, World Wide Web Consortium, November 1999. Available online http://www.w3.org/TR/xpath.

[CDG+97] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. Available online http://www.grappa.univ-lille3.fr/tata, 1997. release October, 1rst 2002.

[CIMP03] David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier, editors. Mathematical Markup Language (MathML) Version 2.0. W3C Recommendation, World Wide Web Consortium, October 2003. Available online http://www.w3.org/TR/2003/REC-MathML2-20031021/.

[Cla99] James Clark, editor. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, World Wide Web Consortium, November 1999. Available online http://www.w3.org/TR/xslt.

[DBO06] Patrick Durusau, Michael Brauer, and Lars Oppermann, editors. Open Document Format for Office Applications (Open Document) v1.1. Committe Draft, Oasis, July 2006. Available online http://www.oasis-open.org/committees/download.php/19321/OpenDocument-v1.1-cd2.pdf.

[DG84] William F. Dowling and Jean H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[Don65] John Doner. Decidability of the Weak Second-Order Theory of Two Successors. *Notices of the American Mathematical Society*, 12:365–468, 1965.

[Don70] John Doner. Tree Acceptors and some of their Applications. *Journal of Computer and System Sciences (JCSS)*, 4(5):406–451, 1970.

[ECM06] ECMA Technical Commitee 39. C# Language Specification. Standard ECMA-334, ECMA International, June 2006. Available online

http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[EH99] Joost Engelfriet and Hendrik J. Hoogeboom. Tree-Walking Pebble Automata. In J. Karkumäki, H. Maurer, G. Paun, and G. Rozenberg, editors, *Jewels are forever, contributions to Theoretical Computer Science in honor of Arto Salomaa*, pages 72–83. Springer, 1999.

[EHV99] Joost Engelfriet, Hendrik J. Hoogeboom, and Jan P. Van Best. Trips on Trees. *Acta Cybernetica*, 14(1):51–64, 1999.

[EM99] Joost Engelfriet and Sebastian Maneth. Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. *Information and Computation*, 154(1):34–91, 1999.

[EM03a] Joost Engelfriet and Sebastian Maneth. A Comparison of Pebble Tree Transducers with Macro Tree Transducers. *Acta Informatica*, 39(9):613–698, 2003.

[EM03b] Joost Engelfriet and Sebastian Maneth. Macro Tree Translations of Linear Size Increase are MSO Definable. *Society for Industrial and Applied Mathematics (SIAM) Journal of Computing*, 32(4):950–1006, 2003.

[Eng75] Joost Engelfriet. Bottom-up and Top-down Tree Transformations – A Comparison. *Mathematical Systems Theory*, 9(3):198–231, 1975.

[Eng77] Joost Engelfriet. Top-down Tree Transducers with Regular Look-Ahead. *Mathematical Systems Theory*, 10:289–303, 1977.

[Eng80] Joost Engelfriet. Some Open Questions and Recent Results on Tree Transducers and Tree Languages. In R.V. Book, editor, *Formal Language Theory; Perspectives and Open Problems*, pages 241–286. Academic Press, New York, 1980.

[ERS80] Joost Engelfriet, Grzegorz Rozenberg, and Giora Slutzki. Tree Transducers, L Systems, and Two-way Machines. *Journal of Computer and System Sciences (JCSS)*, 20(2):150–202, 1980.

[ES77] Joost Engelfriet and Erik M. Schmidt. IO and OI. (I). *Journal of Computer and System Sciences (JCSS)*, 15(3):328–353, 1977.

[ES78] Joost Engelfriet and Erik M. Schmidt. IO and OI. (II). *Journal of Computer and System Sciences (JCSS)*, 16(1):67–99, 1978.

[EV85] Joost Engelfriet and Heiko Vogler. Macro Tree Transducers. *Journal of Computer and System Sciences (JCSS)*, 31(1):71–146, 1985.

[EV86] Joost Engelfriet and Heiko Vogler. Pushdown Machines for the Macro Tree Transducer. *Theoretical Computer Science*, 42:251–368, 1986.

[EV88] Joost Engelfriet and Heiko Vogler. High Level Tree Transducers and Iterated Pushdown Tree Transducers. *Acta Informatica*, 26:131–192, 1988.

[Fis68a] Michael J. Fischer. *Grammars with Macro-like Productions*. PhD thesis, Harvard University, Massachusetts, 1968.

[Fis68b] Michael J. Fischer. Grammars with Macro-like Productions. In *9th IEEE Symopsium on Switching and Automata Theory (SWAT)*, pages 131–142, Schenectady, New York, 1968.

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics; Formal Models Based on Tree Transducers*. Springer, Berlin, 1998.

[FW04] David C. Fallside and Priscilla Walmsley, editors. XML Schema. W3C Recommendation, World Wide Web Consortium, October 2004. Available online http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[Gol90] Charles F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[GS97] Ferenc Gécseg and Magnus Steinby. Tree languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer, Berlin, 1997.

[HMKV01] M. Höff, R. Maletti, A. Kühnemann, and J. Voigtländer. Tree Transducer Based Program Transformations for Haskell$^+$. Progress report, Technische Universität, Dresden, 2001.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, New York, second edition, 2001.

[Hos98] Philipp Hoschka, editor. Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. W3C Recommendation, World Wide Web Consortium, June 1998. Available online http://www.w3.org/TR/1998/REC-smil-19980615.

[HP01] Haruo Hosoya and Benjamin C. Pierce. XDuce: A Typed XML Processing Language. In *Third International Workshop on The World Wide Web and Databases (WebDB). Selected Papers*, Lecture Notes in Computer Science (LNCS) Vol. 1997, pages 226–244, 2001.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technologies (TOIT)*, 3(2):117–148, 2003.

[Jam01] RELAX NG Tutorial. Oasis committee specification, The Organization for the Advancement of Structured Information Standards (OASIS), December 2001. Available online http://www.oasis-open.org/committees/relax-ng.

[Kay05] Michael Kay, editor. XSL Transformations (XSLT) Version 2.0. W3C Candidate Recommendation, World Wide Web Consortium, November 2005. Available online http://www.w3.org/TR/xslt20.

[KCM04] Chritian Kirkegaard, Aske S. Christensen, and Anders Møller. A Runtime System for XML Transformations in Java. In *Second International XML Database Symposium (XSym)*, Lecture Notes in Computer Science (LNCS) Vol. 3186, pages 143–157, 2004.

[KL03] Martin Kempa and Volker Linnemann. Type Checking in XOBE. In *10. Konferenz über Datenbanksysteme für Business, Technologie und Web (BTW)*, Lecture Notes in Informatics (LNI) Vol. 26, pages 227–246, 2003.

[KMS04] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static Analysis of XML Transformations in Java. *IEEE Transactions on Software Engineering*, 30:181–192, 2004.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[KV94] Armin Kühnemann and Heiko Vogler. Synthesized and Inherited Functions. A new Computational Model for Syntax-Directed Semantics. *Acta Informatica*, 31(5):431–477, 1994.

[LDJ$^+$04] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System, Documentation and User's*

Manual. Institute National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, France, 2004.

[Ley02] Michael Ley. The DBLP Computer Science Bibliography: Evolution, Research Issues, Perspectives. In *9th International Symposium on String Processing and Information Retrieval (SPIRE)*, Lecture Notes in Computer Science (LNCS) Vol. 2476, pages 1–10, 2002. Digital Bibliography & Library Project (DBLP) online http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/.

[Man03] Sebastian Maneth. The Macro Tree Transducer Hierarchy Collapses for Functions of Linear Size Increase. In *23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Lecture Notes in Computer Science (LNCS) Vol. 2914, pages 326–337, 2003.

[Man06] Andreas Mantke. Dokumentationen für Version 2.0. Technical report, OpenOffice.org, August 2006. Available online http://de.openoffice.org/doc/howto_2_0/index.html.

[MBPS05] Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. XML Type Checking with Macro Tree Transducers. In *24th ACM Symposium on Principles of Database Systems (PODS)*, pages 283–294. ACM Press, 2005.

[MLM01] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, Montréal, Quebec, 2001. Available online http://www.idealliance.org/papers/extreme00/.

[MN99] Sebastian Maneth and Frank Neven. Structured Document Transformations Based on XSL. In *7th International Workshop on Database Programming Languages (DBPL)*, Lecture Notes in Computer Science (LNCS) Vol. 1949, pages 80–98, 1999.

[MN02] Wim Martens and Frank Neven. Typechecking Top-Down Uniform Unranked Tree Transducers. In *9th International Conference on Database Theory (ICDT 2003)*, Lecture Notes in Computer Science (LNCS) Vol. 2572, pages 64–78. Springer, 2002.

[MN04] Wim Martens and Frank Neven. Frontiers of Tractability for Typechecking Simple XML Transformations. In *23rd ACM Symposium on Principles of Database Systems (PODS)*, pages 23–34. ACM Press, 2004.

[MN05] Wim Martens and Frank Neven. On the Complexity of Typechecking Top-Down XML Transformations. *Theoretical Computer Science*, 336(1):153–180, 2005.

[MOS05] Anders Møller, Mads Østerby Olesen, and Michael Schwartzbach. Static Validation of XSL Transformations. Technical Report RS-05-32, Department of Computer Science, Aarhus Universitet (BRICS), October 2005.

[MPS06] Sebastian Maneth, Thomas Perst, and Helmut Seidl. Exact XML Type Checking in Polynomial Time. Forschungsbericht nr. 06-4, Universität Trier, 2006.

[MS99] Erik Meijer and Mark Shields. XMλ: A Functional Language for Constructing and Manipulating XML Documents. 1999. Available online http://www.cse.ogi.edu/~mbs/pub/xmlambda/.

[MS05] Anders Møller and Michael I. Schwartzbach. The Design Space of Type Checkers for XML Transformation Languages. In *10th International*

Conference on Database Theory (ICDT), Lecture Notes in Computer Science (LNCS) Vol. 3363, pages 17–36, 2005.

[MS06] Anders Møller and Michael Schwartzbach. *An Introduction to XML and Web Technologies*. Addison-Wesley, Reading, Massachusetts, 2006.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML Transformers. In *19th ACM Symposium on Principles of Database Systems (PODS)*, pages 11–22, 2000.

[MSV03] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML Transformers. *Journal of Computer and System Sciences (JCSS)*, 66(1):66–97, 2003.

[MTRD97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[Mur01] Makoto Murata. Extended Path Expressions for XML. In *20th ACM Symposium on Principles of Database Systems (PODS)*, pages 153–166, 2001.

[Neu99] Andreas Neumann. *Parsing and Querying XML Documents in SML*. PhD thesis, Universität Trier, Trier, 1999.

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

[NNS+04] Flemming Nielson, Hanne Riis Nielson, Hongyan Sun, Mikael Buchholtz, René Rydhof Hansen, Henrik Pilegaard, and Helmut Seidl. The Succinct Solver Suite. In Kurt Jensen and Andreas Podelski, editors, *10th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science (LNCS) Vol. 2988, pages 251–265, 2004.

[NS01] Flemming Nielson and Helmut Seidl. Succinct Solvers. Forschungsbericht nr. 01-12, Universität Trier, 2001.

[NS02] Frank Neven and Thomas Schwentick. Query Automata. *Theoretical Computer Science*, 1-2(275):633–674, 2002.

[NSN02] Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A Succinct Solver for AFLP. *Nordic Journal of Computing*, 9(4):335–372, 2002.

[Pan03a] Tadeusz Pankowski. Specifying Transformations for XML Data. In B. Thalheim and G. Fiedler, editors, *Emerging Database Research in East Europe. Pre-Conference Workshop of VLDB 2003*, pages 86–90, Cottbus, 2003. Brandenburgische Technische Universität. Computer Science Reports, Report 14/03.

[Pan03b] Tadeusz Pankowski. Transformation of XML Data using an Unranked Tree Transducer. In Kurt Bauknecht, A. Min Tjao, and Gerald Quirchmayr, editors, *4th International Conference on E-Commerce and Web Technologies (EC-Web)*, Lecture Notes in Computer Science (LNCS) Vol. 2738, pages 259–269, 2003.

[Pem02] Steven Pemberton et al. XHTML™ 1.0 The Extensible HyperText Markup Language. W3C Recommendation, World Wide Web Consortium, August 2002. Available online http://www.w3.org/TR/2002/REC-xhtml1-20020801/.

[PS04] Thomas Perst and Helmut Seidl. Macro Forest Transducers. *Information Processing Letters*, 89(3):141–149, 2004.

[Qui06] Liam Quin. Extensible Markup Language (XML). Technical report, World Wide Web Consortium, April 2006. Available online http://www.w3.org/XML/.

[Rou70a] William C. Rounds. Context-free Grammars on Trees. In *1st Annual ACM Symposium on Theory of Computing (STOC)*, pages 143–148, Marina del Ray, California, 1970.

[Rou70b] William C. Rounds. Mappings and Grammars on Trees. *Mathematical Systems Theory*, 4(3):257–287, 1970.

[Sal73] Arto Salomaa. *Formal Languages*. ACM Monograph Series. Academic Press, New York, 1973.

[Sei94a] Helmut Seidl. Haskell Overloading is DEXPTIME Complete. *Information Processing Letters*, 52(3):57–60, 1994.

[Sei94b] Helmut Seidl. Least Solutions of Equations over $\mathcal{N}$. In *International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science (LNCS) Vol. 820, pages 400–411. Springer, 1994.

[Sei06] Helmut Seidl. ConstraintSolver Suite. Available online http://wwwseidl.informatik.tu-muenchen.de/repos/seidl/OCaml/Fix/, 2006. Source Code.

[ST00] Magnus Steinby and Wolfgang Thomas. Trees and Term Rewriting in 1910: On a Paper by Axel Thue. *Bulletin of the European Association for Theoretical Computer Science*, (72):256–269, 2000.

[Suc98] Dan Suciu. Semistructured Data and XML. In *International Conference on Foundations of Data Organization (FODO)*, 1998.

[Suc01] Dan Suciu. Typechecking for Semistructured Data. In *International Workshop on Database Programming Languages (DBPL)*, Lecture Notes in Computer Science (LNCS) Vol. 2397, pages 1–20, 2001.

[Suc02] Dan Suciu. The XML Typechecking Problem. *SIGMOD record*, 31(1):89–96, March 2002.

[Tho97] Wolfgang Thomas. Languages, Automata, and Logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer, Berlin, 1997.

[Thu10] Axel Thue. Die Lösung eines Spezialfalles eines generellen logischen Problems. *Kra. Videnskabs-Selskabets Skrifter. I. Matematisk-naturvidenskapelig Klasse 1910*, (8), 1910.

[Toz01] Akihiko Tozawa. Towards Static Type Inference for XSLT. In *ACM Symposium on Document Engineering*, pages 18–27, 2001.

[TW65] James W. Thatcher and Jesse B. Wright. Generalized Finite Automata. *Notices of the American Mathematical Society*, page 820, 1965. Abstract No. 65T-649.

[TW68] James W. Thatcher and Jesse B. Wright. Generalized Finite Automata with an Application to a Decision Problem of Second Order Logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[Via01] Victor Vianu. A Web Odyssey: From Codd to XML. In *20th ACM Symposium on Principles of Database Systems (PODS)*, pages 1–15, 2001.

[VK04] Janis Voigtländer and Armin Kühnemann. Composition of functions with accumulating parameters. *Journal of Functional Programming*, 14(3):317–363, 2004.

[Voi01] Janis Voigtländer. Composition of Restricted Macro Tree Transducers, 2001.

[Voi04] Janis Voigtländer. *Tree Transducer Composition as Program Transformation*. PhD thesis, Technische Universität, Dresden, 2004.

[XSL99] World Wide Web Consortium (W3C). *XSL Transformations (XSLT)*, 16 November 1999. Available online http://www.w3.org/TR/xslt.