Composition of restricted Macro Tree Transducers

(Revised version)
Contents

1 Introduction .................................................................. 1

2 Preliminaries ................................................................. 6
   2.1 Functions on Trees .................................................. 7
   2.2 Macro Tree Transducers ......................................... 8
   2.3 Proving a Lemma ................................................... 12

3 The Construction ............................................................ 14
   3.1 The Idea ................................................................. 14
   3.2 Formal Development ............................................... 20
   3.3 The symmetric Result does not hold ......................... 31

4 Tidying up ..................................................................... 33
   4.1 Removing superfluous Context Parameters ................ 34
   4.2 Eliminating Copy-states .......................................... 38
   4.3 Turning non-terminating Programs into terminating ones 43

5 Useful Things ............................................................... 46
   5.1 A Size-Height Property for non-copying Macro Tree Transducers 46
   5.2 Counting Reduction Steps of non-copying Macro Tree Transducers 50
   5.3 An Implementation ................................................ 53

6 Conclusions ................................................................ 55

Bibliography .................................................................. 57

Appendix: Proving the Correctness of Construction 3.2 .......... 59
Chapter 1

Introduction

An important style of writing programs in a functional language is to define new functions by composition of existing functions. This means that the result of a function application is passed as argument to another function. This technique of solving an overall problem by combining solutions of partial problems, which might be solved by predefined or self-written functions, allows to write simple and modular programs, but can cause inefficiencies. If the created intermediate results are structured objects—for example lists or trees—their creation and eventual destruction will consume time and memory space. Furthermore, it is possible that, e.g., lists are traversed more often, than would really be necessary for solving the overall problem. Often, other function definitions can be given that solve the same problem, but avoid the mentioned inefficiencies. Thus, we would like to have transformation techniques, which enable us to optimize functions written in the modular style, by eliminating intermediate data structures. Several such techniques have been studied in the literature, e.g., the fold/unfold-technique from [BD77], its algorithmic instances supercompilation [Tur86] and deforestation [Wad90], catamorphism fusion [Mal89, MFP91] and shortcut deforestation [GLP93].

In this thesis, we follow an approach for eliminating intermediate results, which is based on the theory of tree transducers. Particularly, we consider Macro Tree Transducers [Eng80], which can represent a large class of functional programs. Composition of functions then corresponds to consecutive execution of Macro Tree Transducers, that is, the output of one transducer is given as input to the second one, and so on (we will only consider the composition of two Macro Tree Transducers). The goal of avoiding intermediate data structures corresponds to the problem of replacing such a composition of Macro Tree Transducers by a single Macro Tree Transducer, which computes the same function. It is well known that this is not possible in general, because the class of Macro Tree Transductions is not closed under composition [EV85]. For subclasses, however, such constructions can be found.

For illustration of the problem of intermediate results and as informal introduction to the concept of Macro Tree Transducers, consider the following example. Assume given a representation of arithmetic terms, built from two variables and the binary operations addition and multiplication, as trees with nullary symbols \( A \) and \( B \) and binary symbols \( + \) and \( \ast \) (this representation corresponds to an algebraic data type in a functional language, e.g., data Term = A | B | + Term Term | \ast Term Term in Haskell). Further, assume given a function for computing the prefix-notation of such a term as monadic tree of now

\footnote{Macro Tree Transducers are extended schemes of primitive recursion, allowing simultaneous definition of several functions and nested function calls in parameter positions.}
unary symbols \(A, B, +, \ast\), and a nullary symbol \(\varepsilon\) for denoting the end of the list\(^2\). As a Macro Tree Translucen, this function \(pre\) can be given by the following rules:

(i) \(pre(+ (x_1, x_2), y) \rightarrow + (pre(x_1, pre(x_2, y)))\)
(ii) \(pre(\ast (x_1, x_2), y) \rightarrow \ast (pre(x_1, pre(x_2, y)))\)
(iii) \(pre(A, y) \rightarrow A(y)\)
(iv) \(pre(B, y) \rightarrow B(y)\)

These rules can be seen as just another notation for the defining equations in a functional language. The function is defined by pattern matching on its first argument and uses an additional accumulating argument, which here will be called context parameter. In order to compute the prefix-notation of a term \(t\), the rules (i)–(iv) are used to exhaustively rewrite the initial expression \(pre(t, \varepsilon)\), as shown in Figure 1.1 for \(t = \ast (+ (A, B), A)\).

![Figure 1.1: Computation by rewriting](image)

Now, consider the problem of computing, for a given term as above, a sequence of instructions for a stack-machine with two registers and instructions for addition and multiplication (represented as monadic tree, labelled with instructions \(LOAD_A, LOAD_B, ADD\) and \(MUL\)). Instead of writing a program for this problem from scratch, we might realize that we can solve the task by reversing the prefix-notation of the given term and replacing labels \(A, B, +\) and \(\ast\) by \(LOAD_A, LOAD_B, ADD\) and \(MUL\), respectively. Thus, we define the following function \(aux\)\(^3\):

(v) \(aux(A(x), y) \rightarrow aux(x, LOAD_A(y))\)
(vi) \(aux(B(x), y) \rightarrow aux(x, LOAD_B(y))\)
(vii) \(aux(+ (x), y) \rightarrow aux(x, ADD(y))\)
(viii) \(aux(\ast (x), y) \rightarrow aux(x, MUL(y))\)
(ix) \(aux(\varepsilon, y) \rightarrow y\)

The instruction sequence for a given term \(t\) can now be computed by composing the existing function \(pre\) and our new auxiliary function, namely by rewriting the composite expression \(aux(pre(t, \varepsilon), \varepsilon)\) with rules (i)–(ix), as shown in Figure 1.2.

This modular solution, however, is inefficient, because it creates and consumes an intermediate result. Depending on the used rewriting strategy, respectively the evaluation order of our functional language, this intermediate data structure might never exist as a whole, but nevertheless, for all of its nodes, memory cells have to be allocated and later deallocated. Also, even with a lazy evaluation strategy, this program performs a superfluous traversal on the intermediate list. To avoid these inefficiencies, we could instead have given the

\(^2\)In most functional languages, one would use a built-in type of lists instead, but for our illustration this representation of lists as monadic trees is more handy. Our techniques are also transferable to, e.g., polymorphic lists.

\(^3\)On a built-in list type, this auxiliary function might be obtained from fusion of a map-function and the standard list \(reverse\)-function.
following function definition:

(\(x\)) : \(\text{ins}(+(x_1,x_2),y) \rightarrow \text{ins}(x_2,\text{ins}(x_1,\text{ADD}(y)))\)

(\(x_{i1}\)) : \(\text{ins}(* (x_1,x_2),y) \rightarrow \text{ins}(x_2,\text{ins}(x_1,\text{MUL}(y)))\)

(\(x_{i2}\)) : \(\text{ins}(A,y) \rightarrow \text{LOAD}_A(y)\)

(\(x_{i3}\)) : \(\text{ins}(B,y) \rightarrow \text{LOAD}_B(y)\)

If now, for a given term \(t\), we use rules (\(x\))–(\(x_{i3}\)) on \(\text{ins}(t,\epsilon)\), we will calculate the same instruction sequence as before with the modular solution, but without creating and traversing the intermediate list (see Figure 1.3).

Figure 1.3: Avoiding the intermediate data structure

For the programmer, however, it would probably be more difficult (and, arguably, bad non-modular style) to come up with this more efficient solution to the problem from scratch, instead of reusing the already existing function for the prefix-notation of a term together with a simple list processing function.

Consequently, it would be worthwhile to have a technique, which can transform the modular solution (i)–(ix) into the efficient solution (x)–(xiii). To the best of our knowledge, techniques such as deforestation and catamorphism fusion cannot perform the optimization that we want to achieve here. In particular, deforestation fails due to its well known problem of not reaching accumulating arguments. An approach that is applicable to our example is based on attribute grammars [Knuth98] and was proposed independently in [Kühl98] and [CDPR99]. The idea is to transform the two functions (in our formalism represented by two restricted Macro Tree Transducers), which we want to compose, into attribute grammars, respectively Attributed Tree Transducers [Fili81], which are an abstraction of attribute grammars in the same sense as Macro Tree Transducers are used as primitive
recursive program schemes. If the first Attributed Tree Transducer fulfills the single-use restriction (cf. the syntactic single-use requirement in [Gie88]), which essentially means that every attribute instance in a tree may be used at most once in calculating the values of other attribute instances, the two transducers can be composed into a single Attributed Tree Transducer (based on composition results from [Gan83, GG84, Gie88]). Applying a construction based on [Fra82], this Attributed Tree Transducer can then be transformed into a Macro Tree Transducer, thus giving a functional program for the composition of the two original functions, but without producing and consuming the intermediate result.

This approach does not work for all Macro Tree Transducers, because only restricted Macro Tree Transducers can be transformed into Attributed Tree Transducers, which can then be composed. One such restriction is the property of a Macro Tree Transducer to be weakly single-use, which roughly speaking means that at every occurrence in the input tree, recursive calls of functions on subtrees are restricted to appear at most once. Moreover, in order to obtain a single-use Attributed Tree Transducer such that the composition result for Attributed Tree Transducers becomes applicable, the first Macro Tree Transducer has to be further restricted to be single-use, which in addition to the restriction of being weakly single-use means that context parameters cannot be copied. Figure 1.4 shows the transformation steps we have to take, where arrows indicate transformations and the semicolon stands for composition of tree transduction classes. Here, the class of functions computable by Macro Tree Transducers is denoted as MAC; the class of Attributed Tree Transductions as ATT, and the restrictions single-use and weakly single-use are indicated by subscripts \textit{su} and \textit{wsu}, respectively.

![Diagram](image)

**Figure 1.4: Indirect composition**

This composition of restricted Macro Tree Transducers by indirect via Attributed Tree Transducers has been implemented in the Haskell\textsuperscript{+} program transformation system [Les99, HVM+01]. There are several reasons, why we are interested in a direct construction for composing Macro Tree Transducers without the above indirection. Firstly, there would be benefits for the implementation, both, because a direct construction could be implemented more efficiently, and because the implementor—e.g., a compiler constructor—could work directly on functional programs, instead of having to consider the formalism of attribute grammars, just for optimization's sake\textsuperscript{4}. Secondly, there is a chance that a direct construction would produce better program code than the indirectity by several transformations, because every single program transformation tends to introduce a certain "ballast" into the program, like, e.g., superfluous function parameters. Thirdly, we want to understand better and broaden the applicability of Macro Tree Transducer composition

\textsuperscript{4}This argument does not apply to the Haskell\textsuperscript{+} system, which was designed as extension to the Haskell language and allows the definition of attribute grammars.
by generalizing the result \( MAC_{su} ; MAC_{w} \subseteq MAC \). As we already noted, we cannot expect to be able to compose two arbitrary Macro Tree Transducers, because the class of Macro Tree Transducers is known not to be closed under composition. But, we can aim for weaker restrictions on the Macro Tree Transducers than necessary so far, which still allow a composition. The success of this intention is limited, if we stick to the indirect construction via Attributed Tree Transducers, because then we can only compose Macro Tree Transducers that can be transformed into Attributed Tree Transducers, which is not always possible, because we have a strict inclusion \( ATT \subseteq MAC \) [Eng86, Fra82]. A direct composition construction has no such a priori limitation. In fact, we will prove—with \( MAC_{nc} \) being the class of functions computable by non-copying Macro Tree Transducers—the characterization \( MAC_{nc} ; MAC_{w} \subseteq MAC \), which is a generalization of the result quoted above, because single-use Macro Tree Transducers cannot copy context parameters and thus are, by definition, non-copying.

Another problem that we want to consider in this thesis is the already mentioned “ballast”, which can be introduced by our program transformations. Besides the problem of introducing superfluous function parameters (which will be detected by a fixpoint construction), another phenomenon is of special interest. In [CDPR99] it was called the problem of copy-rules and refers to the introduction—by the composition of attribute grammars—of rules that lead to superfluous traversals through the input tree. Thus, it can happen that the intermediate data structure is eliminated, but the resulting program is no more efficient than the original one (with respect to the number of reduction steps). In [CDPR99] it is stated that in most cases, a static analysis of the Attributed Tree Transducer can detect these copy-rules, such that they can be eliminated before the transformation into the final Macro Tree Transducer. This analysis is not used in our construction, because we do not use the indirection via Attributed Tree Transducers. However, after having introduced our direct construction for \( MAC_{nc} ; MAC_{w} \subseteq MAC \), we will note that the problem of superfluous traversals through the input tree is not specific to the composition of attribute grammars, but also arises from our construction. We will generalize the concept of copy-rules to the concept of copy-states on the level of our functional programs and present a construction, which detects and removes all copy-states of a given Macro Tree Transducer, thus eliminating superfluous traversals. Our direct composition construction together with the presented post-processing constructions will then be able to transform the modular program—rules (i) to (ix)—from the introductory example into the more efficient program corresponding to rules (x)–(xiii).

This thesis is divided into six chapters and an appendix. In Chapter 2 we define necessary notations and introduce the basic concepts of Macro Tree Transducers and a proof principle. In Chapter 3 we develop our main construction for composing two Macro Tree Transducers—the first of which is non-copying and the second one weakly single-use—to a single one. We will also show that a symmetric result—composing a weakly single-use Macro Tree Transducer and a non-copying one in this order—is not possible in general. In Chapter 4 we develop two fixpoint constructions for post-processing Macro Tree Transducers obtained by the composition construction, eliminating inefficiencies caused by superfluous context parameters and by copy-states, respectively. In Chapter 5 we show two interesting applications of the composition result to problems about non-copying Macro Tree Transducers. We will also present an implementation of our composition construction in the Haskell\(^5\) system by Andreas Maletti. In Chapter 6 we draw some conclusions. The correctness proof for our main construction was moved to the Appendix.

---

\(^5\) Another possibility is a new way of applying results from [Küh98], by first decomposing tree transducers, thus introducing new intermediate results, but creating new opportunities for compositions, which altogether can still lead to an optimization.
Chapter 2

Preliminaries

We denote by $\mathbb{N}$ the set of natural numbers including 0, and for $n \in \mathbb{N}$, by $[n]$ the set $\{1, \ldots, n\}$. For a finite, non-empty set $S$ of natural numbers, we denote by $\text{max}(S)$ the maximum of all its elements.

We will use several sets of lowercase variables. As a convention, we denote for $k \in \mathbb{N}$, by $X_k$ the finite set $\{x_1, \ldots, x_k\}$ of variables, analogous for $Y$ and $Z$.

For a set $S$, we denote by $S^*$ the set of finite sequences of elements of $S$, where $\epsilon$ will denote the empty sequence. $\mathcal{P}(S)$ will denote the power set of a given set $S$. $|S|$ will denote the number of elements of a finite set $S$.

For simplicity, we define substitution as substitution over strings (elements of $S^*$ for some finite set $S$). For a string $v$ and two lists of strings $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ (with $n \in \mathbb{N}$), such that no two strings $u_i$ and $u_j$ ($i \neq j$) overlap in $v$, we denote by $v[u_i \leftarrow v_i, 1 \leq i \leq n]$ the string obtained from $v$ by replacing all occurrences of every $u_i$ in $v$ by $v_i$. We will also use the alternative notation $v[u_1, \ldots, u_n \leftarrow v_1, \ldots, v_n]$. We write substitutions left-associative.

A ranked alphabet is a pair $(\Sigma, \text{rank}_\Sigma)$, where $\Sigma$ is a finite set of symbols and $\text{rank}_\Sigma$ assigns to each of these symbols a natural number, its rank. By convention, we will assume that ranked alphabets are disjoint from $\mathbb{N}$. For every $k \in \mathbb{N}$, we define $\Sigma^{(k)} = \{\sigma \in \Sigma \mid \text{rank}_\Sigma(\sigma) = k\}$. The rank $k$ of a symbol $\sigma$ will also be denoted by writing $\sigma^{(k)}$.

For a ranked alphabet $\Sigma$, we denote the set of all its ranks as $\text{rank}(\Sigma) = \{k \in \mathbb{N} \mid \exists \sigma \in \Sigma : \text{rank}_\Sigma(\sigma) = k\}$. A ranked alphabet of states is a ranked alphabet $Q$ with $Q^{(0)} = \emptyset$. A monadic ranked alphabet is a ranked alphabet, where all symbols have rank zero or one.

For a ranked alphabet $\Sigma$ and a set $A$ disjoint from $\Sigma$, we define the set $T_\Sigma(A)$ of trees over $\Sigma$ indexed by $A$ as the smallest set $T$ such that (i) $A \subseteq T$ and (ii) if $k \in \mathbb{N}$, $\sigma \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in T$, then also $\sigma(t_1, \ldots, t_k) \in T$. For nullary symbols, we simply write $\sigma$ instead of $\sigma(\epsilon)$. By $T_\Sigma$ we denote the set $T_\Sigma(\emptyset)$.

For a set $S$ and a binary relation $\rel \subseteq S \times S$, we denote by $\rel^+$ the transitive and by $\rel^*$ the reflexive, transitive closure of $\rel$, respectively.

Let $\Sigma$ be a ranked alphabet. A rewrite rule over $\Sigma$ is a rule of the form $\text{lhs} \rightarrow \text{rhs}$ with $\text{lhs}, \text{rhs} \in T_\Sigma(V)$ for some set of variables $V$, such that $\text{lhs}$ does not contain two occurrences of the same variable and every variable occurring in $\text{rhs}$ is also contained in $\text{lhs}$. A set $R$ of such rules induces a term rewriting system (see for example [Hue80]), denoted $(\Sigma, R)^1$, with a binary reduction relation $\Rightarrow_R \subseteq T_\Sigma \times T_\Sigma$, such that $s \Rightarrow_R t$, if $R$ contains a rule $\text{lhs} \rightarrow \text{rhs}$, there is a tree $c \in T_\Sigma(\{v\})$ (with $v \notin V$), which contains $v$ exactly once and there exist $n \in \mathbb{N}$, trees $t_1, \ldots, t_n \in T_\Sigma$ and variables $v_1, \ldots, v_n \in V$.

---

1Usually $\Sigma$, as well as $V$, will be clear from the context and we will only mention $R$. 
such that:
\[
\begin{align*}
  s &= \xi[v \leftarrow \text{lhs}[v_i \leftarrow t_i, 1 \leq i \leq n]] \\
  t &= \xi[v \leftarrow \text{rhs}[v_i \leftarrow t_i, 1 \leq i \leq n]]
\end{align*}
\]

A reduction relation \( \Rightarrow_R \) is called confluent, if for every \( s, s_1, s_2 \in T_\Sigma \) with \( s \Rightarrow_R^* s_1 \) and \( s \Rightarrow_R^* s_2 \), there exists a so-called confluence element \( s' \in T_\Sigma \) with \( s_1 \Rightarrow_R s' \) and \( s_2 \Rightarrow_R s' \). A reduction relation \( \Rightarrow_R \) is called terminating, if there is no infinite chain \( s_1 \Rightarrow_R s_2 \Rightarrow_R s_3 \Rightarrow_R \ldots \). If \( s \Rightarrow_R^* t \) and there is no \( t' \) with \( t \Rightarrow_R t' \), then \( t \) is called a normal form of \( s \) with respect to \( \Rightarrow_R \). If \( \Rightarrow_R \) is confluent and terminating, then every tree \( s \in T_\Sigma \) has a unique normal form, denoted as \( \text{nf}(\Rightarrow_R, s) \).

Let \( S \) be a finite set. A set operator \( \Phi : \mathcal{P}(S) \to \mathcal{P}(S) \) is called monotonic, if for every \( s_1 \subseteq s_2 \subseteq S \) holds: \( \Phi(s_1) \subseteq \Phi(s_2) \). Define \( \Phi^0 = \emptyset \) and \( \Phi^{n+1} = \Phi(\Phi^n) \), for every \( n \in \mathbb{N} \). If \( \Phi \) is monotonic, then we have \( \Phi^0 \subseteq \Phi^1 \subseteq \Phi^2 \subseteq \ldots \), and since \( S \) is finite, this infinite chain cannot be strictly increasing. Thus, there exists the least fixpoint \( \Phi^* = \bigcup_{n \in \mathbb{N}} \Phi^n \), which is reached after finitely many iterations of \( \Phi \).

Let \( I \) be a finite set and for every \( i \in I \), let be given some set \( S(i) \). We write the indexed product \( \prod_{i \in I} S(i) \) for the type of dependently typed functions \( f : I \to \bigcup_{i \in I} S(i) \), such that for every \( i \in I \), we have \( f(i) \in S(i) \). Such a function \( f \in \prod_{i \in I} S(i) \) will be denoted as \([i \mapsto f(i_1), \ldots, i_n \mapsto f(i_n)]\) for some ordering \( i_1, \ldots, i_n \) of the elements in \( I \). Further, for a ranked alphabet of states \( Q \), we will write \( \prod_{q \in Q^{(n+1)}} S(r) \) for \( \prod_{q \in Q^{(n+1)}} S'(q) \), where for every \( q \in Q \), we define \( S'(q) = S(\text{rank}_Q(q) - 1) \).

**Example 2.1** For \( Q = \{q^{(2)}, p^{(3)}\} \), the indexed product \( \prod_{q \in Q^{(n+1)}} [r] \) contains the function \( f_1 = [q \mapsto 1, p \mapsto 2] \), but not the function \( f_2 = [q \mapsto 2, p \mapsto 1] \), because \( 2 \notin \text{[(rank}_Q(q)-1)] \).

### 2.1 Functions on Trees

Let \( \Sigma, \Delta \) and \( \Omega \) be ranked alphabets. We call a total function \( \tau : T_\Sigma \to T_\Delta \) a tree transduction from \( T_\Sigma \) to \( T_\Delta \). We define the composition of two tree transductions \( \tau_1 : T_\Sigma \to T_\Delta \) and \( \tau_2 : T_\Delta \to T_\Omega \), denoted by \( \tau_1 \cdot \tau_2 \), as \( (\tau_1 \cdot \tau_2)(t) = \tau_2(\tau_1(t)) \), for every \( t \in T_\Sigma \). Further, we denote the composition of two classes \( \mathcal{T}_1 \) and \( \mathcal{T}_2 \) of tree transductions by \( \mathcal{T}_1 \cdot \mathcal{T}_2 = \{\tau_1 \cdot \tau_2 \mid \tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2\} \).

Let \( A \) be a set disjoint from \( \Sigma \). We will need the set of occurrences in a tree over \( \Sigma \) indexed by \( A \), given by the function \( \text{occ} : T_\Sigma(A) \to \mathcal{P}^+(\mathbb{N}) \), which is defined by structural recursion as follows: (i) if \( t \in \Sigma^{(0)} \cup A \), then \( \text{occ}(t) = \{e\} \) and (ii) if \( t = \sigma(t_1, \ldots, t_k) \) with \( k \in \mathbb{N} \), \( 0 < k \), \( t_1, \ldots, t_k \in T_\Sigma(A) \), then \( \text{occ}(t) = \{e\} \cup \{\xi[\nu] \mid 1 \leq i \leq k, \nu \in \text{occ}(t_i)\} \).

We will also need the label at an occurrence in a tree, given by the mapping \( \text{lab} : \{(t, o) \mid t \in T_\Sigma(A), o \in \text{occ}(t)\} \to \Sigma \cup A \), defined by: (i) if \( t \in \Sigma^{(0)} \cup A \), then \( \text{lab}(t, e) = t \) and (ii) if \( t = \sigma(t_1, \ldots, t_k) \) with \( k \in \mathbb{N} \), \( 0 < k \), \( t_1, \ldots, t_k \in T_\Sigma(A) \), then \( \text{lab}(t, e) = \sigma \) and \( \text{lab}(t, \nu) = \text{lab}(t_i, \nu) \), for \( i \in [k] \) and \( \nu \in \text{occ}(t_i) \). The subtree at an occurrence in a tree is given by the function \( \text{sub} : \{(t, o) \mid t \in T_\Sigma(A), o \in \text{occ}(t)\} \to T_\Sigma(A) \), defined by: (i) \( \text{sub}(t, e) = t \), for \( t \in T_\Sigma(A) \), and (ii) if \( t = \sigma(t_1, \ldots, t_k) \) with \( k \in \mathbb{N} \), \( 0 < k \), \( t_1, \ldots, t_k \in T_\Sigma(A) \), then \( \text{sub}(t, \nu) = \text{sub}(t_i, \nu) \), for \( i \in [k] \) and \( \nu \in \text{occ}(t_i) \).

For some symbol \( s \in \Sigma \cup A \), we obtain the number of occurrences of \( s \) in a tree by the
function \( \#_s : T_{\Sigma}(A) \to \mathbb{N} \), defined as follows: (i) if \( t \in \Sigma^{(0)} \cup A \), then \( \#_s(t) = (\text{if } t = s \text{ then } 1 \text{ else } 0) \) and (ii) if \( t = \sigma(t_1, \ldots, t_k) \) with \( k \in \mathbb{N}, 0 < k, t_1, \ldots, t_k \in T_{\Sigma}(A) \), then \( \#_s(t) = (\text{if } \sigma = s \text{ then } 1 \text{ else } 0) + \sum_{i \in [k]} \#_s(t_i) \).

We define the **height of a tree** by the function \( \text{height} : T_{\Sigma}(A) \to \mathbb{N} \) as follows: (i) if \( t \in \Sigma^{(0)} \cup A \), then \( \text{height}(t) = 0 \) and (ii) if \( t = \sigma(t_1, \ldots, t_k) \) with \( k \in \mathbb{N}, 0 < k, t_1, \ldots, t_k \in T_{\Sigma}(A) \), then \( \text{height}(t) = 1 + \max(\{\text{height}(t_i) \mid i \in [k]\}) \). Finally, we have the notion of **size of a tree**, defined by the mapping \( \text{size} : T_{\Sigma}(A) \to \mathbb{N} \) with: (i) if \( t \in \Sigma^{(0)} \cup A \), then \( \text{size}(t) = 1 \) and (ii) if \( t = \sigma(t_1, \ldots, t_k) \) with \( k \in \mathbb{N}, 0 < k, t_1, \ldots, t_k \in T_{\Sigma}(A) \), then \( \text{size}(t) = 1 + \sum_{i \in [k]} \text{size}(t_i) \).

### 2.2 Macro Tree Transducers

We will define **Macro Tree Transducers** of a slightly more general form than in [FV98], in that we allow for a more general initial expression than just the call of some state with fixed environment\(^2\). First, we describe the possible shapes of right-hand sides of rules.

**Definition 2.2** Let \( Q \) be a ranked alphabet of states, let \( \Delta \) be a ranked alphabet, \( k \in \mathbb{N} \) and \( V \) a set of variables. Then the set \( \text{RHS}(Q, \Delta, k, V) \) of right-hand sides over \( Q \) and \( \Delta \), with \( k \) recursion variables and set of context variables \( V \), is defined as the smallest set \( \text{RHS} \subseteq T_{Q, \Delta, k, V} \) satisfying the following conditions:

1. \( V \subseteq \text{RHS} \)
2. For every \( \delta \in \Delta^{(a)} \) with \( a \in \mathbb{N} \), and \( \phi_1, \ldots, \phi_0 \in \text{RHS} \) holds:
   \[ \delta(\phi_1, \ldots, \phi_0) \in \text{RHS} \]
3. For every \( q \in Q^{(a+1)} \) with \( a \in \mathbb{N} \), \( x_i \in X_k \) and \( \phi_1, \ldots, \phi_0 \in \text{RHS} \) holds:
   \[ q(x_i, \phi_1, \ldots, \phi_0) \in \text{RHS} \]

Now, we can define the components of a Macro Tree Transducer.

**Definition 2.3** A **Macro Tree Transducer** \( M \) is a tuple \((Q, \Sigma, \Delta, e, R)\) with:

1. A ranked alphabet of states \( Q \)
2. A ranked alphabet \( \Sigma \) of input symbols
3. A ranked alphabet \( \Delta \) of output symbols, where \( \Delta^{(0)} \neq \emptyset \) and \( Q \cap (\Sigma \cup \Delta) = \emptyset \)
4. An initial expression \( e \in \text{RHS}(Q, \Delta, 1, \emptyset) \)
5. A finite set \( R \) of rules of the form \( q(\sigma(x_1, \ldots, x_k), y_1, \ldots, y_r) \rightarrow \text{rhs}_{q}\sigma \) with \( k, r \in \mathbb{N}, \sigma \in \Delta^{(k)}, q \in Q^{(r+1)} \) and \( \text{rhs}_{q}\sigma \in \text{RHS}(Q, \Delta, k, Y_r) \), such that there is exactly one rule for every combination of \( q \) and \( \sigma \)

For an input symbol \( \sigma \), we refer to the rules in \( R \) with left-hand side \( q(\sigma(\cdot, \cdot, \cdot, \cdot), \cdot) \) for some \( q \in Q \), as rules at \( \sigma \). The first argument of a state \( q \) is called recursion argument, the others are called context parameters. Of course, the actual variable names used in left- and right-hand sides of Macro Tree Transducer rules are not fixed to come from \( X_k \) and \( Y_r \) for some \( k, r \in \mathbb{N} \); consistent renaming is allowed, and we will also write examples using different variable names.

\(^2\)A similar generalization is used in [Klip00], the alternative would have been to introduce an explicit root symbol as in [Klip00].
Example 2.4 Let $Q_{rev} = \{rev^{(2)}\}$ and $\Sigma_{mon} = \{A^{(1)}, B^{(1)}, E^{(0)}\}$. Then we define the Macro Tree Transducer $M_{rev} = (Q_{rev}, \Sigma_{mon}, \varepsilon_{rev}, R_{rev})$ with set of rules $R_{rev}$:

(i) : $rev(A(x), z) \rightarrow rev(x, A(z))$
(ii) : $rev(B(x), z) \rightarrow rev(x, B(z))$
(iii) : $rev(E, z) \rightarrow z$

and $\varepsilon_{rev} = rev(x_1, E)$. 

The semantics of a Macro Tree Transducer is a function from trees over the input ranked alphabet to trees over the output ranked alphabet. It can be given either by substituting the input tree for $x_1$ in the initial expression $e$ of a Macro Tree Transducer, and then calculating the normal form of this expression with respect to the reduction relation induced by the set $R$ of rules, or by the following definition.

Definition 2.5 We define the tree transduction induced by a Macro Tree Transducer $M = (\Sigma, \Delta, e, R)$ as the total function $\tau(M) : T_\Sigma \rightarrow T_\Delta$, which assigns to every tree $t \in T_\Sigma$ the value of $\tau_{M,k,t}(e, t)$. For a given set $A$ disjoint from $\Delta$, the following set of functions is defined by simultaneous recursion:

$$\tau_{M,k,r,A} : RHS(Q, \Delta, k, Y_r) \times \left( \bigotimes_{i=1}^{k} T_\Sigma \right) \rightarrow T_\Delta \times \left( \bigotimes_{i=1}^{r} T_\Delta \right) \quad | \quad k \in \mathbb{N}\cup \{1\} \quad \text{r-times}$$

For every $k \in \mathbb{N}$, $(r+1) \in \text{rank}(Q) \cup \{1\}$, the function $\tau_{M,k,r,A}$ is defined as:

- $\tau_{M,k,r,A}(y_h, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r) = \theta_h$, for $y_h \in Y_r$, $t_1, \ldots, t_k \in T_\Sigma$, and $\theta_1, \ldots, \theta_r \in T_\Delta(A)$

- $\tau_{M,k,r,A}(\delta(\phi_1, \ldots, \phi_a), t_1, \ldots, t_k, \theta_1, \ldots, \theta_r) = \delta(\tau_{M,k,r,A}(\phi_1, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r), \ldots, \tau_{M,k,r,A}(\phi_a, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r))$ for $a \in \mathbb{N}$, $\delta \in \Delta(\phi), \phi_1, \ldots, \phi_a \in RHS(Q, \Delta, k, Y_r)$, $t_1, \ldots, t_k \in T_\Sigma$, and $\theta_1, \ldots, \theta_r \in T_\Delta(A)$

- $\tau_{M,k,r,A}(q(x_i, \phi_1, \ldots, \phi_a), t_1, \ldots, t_k, \theta_1, \ldots, \theta_r) = \tau_{M,q,A}(t_1, \tau_{M,k,r,A}(\phi_1, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r), \ldots, \tau_{M,q,A}(\phi_a, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r))$ for $a \in \mathbb{N}$, $q \in Q^{(a+1)}$, $x_i \in X_k$, $\phi_1, \ldots, \phi_a \in RHS(Q, \Delta, k, Y_r)$, $t_1, \ldots, t_k \in T_\Sigma$, and $\theta_1, \ldots, \theta_r \in T_\Delta(A)$

Additionally, for every $r, k \in \mathbb{N}$, $q \in Q^{(r+1)}$, $\sigma \in \Sigma^{(k)}$, $t_1, \ldots, t_k \in T_\Sigma$, and $\theta_1, \ldots, \theta_r \in T_\Delta(A)$ holds:

$\tau_{M,q,A}(\sigma(t_1, \ldots, t_k, \theta_1, \ldots, \theta_r)) = \tau_{M,k,r,A}(\sigma(t_1, \ldots, t_k, \theta_1, \ldots, \theta_r))$.

We denote the class of all tree transductions induced by Macro Tree Transducers as MAC.
Although the Macro Tree Transducers presented here have a more general initial expression than those used in [FV98], our tree transduction class MAC coincides with the corresponding one in [FV98], because Macro Tree Transducers in the two representations can be transformed to each other. Note, that the same need not be true for restricted classes of tree transductions, because our representation is more flexible.

We now give an example for the calculation of a tree transduction using our recursively defined semantics.

Example 2.6 We apply the Macro Tree Transducer from the previous example to the input tree $A(B(E))$, that is, we calculate $\tau(M_{rev})(A(B(E)))$:

$\tau(M_{rev})(A(B(E))) =$

$\tau_{M_{rev}1,0}(e_{rev}, A(B(E))) =$

$\tau_{M_{rev}1,0}(A(B(E)), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$\tau_{M_{rev}1,0}(rev(x_1, A(y_1)), B(E), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$\tau_{M_{rev}1,0}(B(E), \tau_{M_{rev}1,0}(A(y_1), B(E), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$\tau_{M_{rev}1,0}(rev(x_1, B(y_1)), E, \tau_{M_{rev}1,0}(A(y_1), B(E), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$\tau_{M_{rev}1,0}(E, \tau_{M_{rev}1,0}(B(y_1), E, \tau_{M_{rev}1,0}(A(y_1), B(E), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$\tau_{M_{rev}1,0}(B(y_1), \tau_{M_{rev}1,0}(A(y_1), B(E), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$\tau_{M_{rev}1,0}(A(x_1, B(E)), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$\tau_{M_{rev}1,0}(A(x_1, B(E)), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$\tau_{M_{rev}1,0}(B(E), \tau_{M_{rev}1,0}(E, A(B(E)))) =$

$B(A(M_{rev}1,0,0)(E, A(B(E)))) =$

$B(A(E))$  \hfill $\Box$

Note, that the functions, which are given in Definition 2.5, are parameterized over a set $A$ disjoint from $\Delta$. For now, this set is just the empty set, but later we will need the generalization for technical reasons.

We already noted that it is also possible to define the semantics of a Macro Tree Transducer by the rewriting system of its rules. The following proposition, which is a consequence of results in Chapter 4 of [FV98], states the equivalence between the recursively defined semantics and reduction semantics.

Proposition 2.7 Let $M = (Q, \Sigma, \Delta, \epsilon, R)$ be a Macro Tree Transducer.

1. $\Rightarrow_R$ is confluent and terminating

2. for every $t \in T_2$ holds: $\tau(M)(t) = nf(\Rightarrow_R \epsilon[x_1 \leftarrow t])$

$\Box$

Example 2.8 Using the reduction semantics, we compute $\tau(M_{rev})(A(B(E)))$ as the normal form of $e_{rev}[x_1 \leftarrow A(B(E))]$ with respect to $\Rightarrow_{rev}$:

$\Rightarrow_{rev}(A(B(E)), E) \Rightarrow_{rev} rev(B(E), A(E)) \Rightarrow_{rev} rev(E, B(A(E))) \Rightarrow_{rev} B(A(E))$  \hfill $\Box$

We introduce two important syntactic properties of Macro Tree Transducers (cf. [Kil98]) and the corresponding computability classes.
Definition 2.9 A Macro Tree Transducer is called non-copying, if in every right-hand side there is no more than one occurrence of every context variable. We denote the class of all tree transductions induced by non-copying Macro Tree Transducers as \( \text{MAC}_{\text{nc}} \).

Definition 2.10 A Macro Tree Transducer \( M = (\mathcal{Q}, \Sigma, \Delta, e, R) \) is called weakly single-use, if: (i) for every \( k \in \mathbb{N}, \sigma \in \Sigma^{(k)}, i \in [k] \) and \( q \in \mathcal{Q} \), a call of the form \( q(x_i, \cdots) \) occurs in a right-hand side of at most one rule at \( \sigma \) and there only once, and (ii) for every \( q \in \mathcal{Q} \), the initial expression \( e \) contains at most one occurrence of a call \( q(x_1, \cdots)^3 \). We denote the class of all tree transductions induced by weakly single-use Macro Tree Transducers as \( \text{MAC}_{\text{wsu}} \).

We also identify Macro Tree Transducers, which have both of the above properties.

Definition 2.11 A Macro Tree Transducer is called single-use, if it is both non-copying and weakly single-use. We denote the class of all tree transductions induced by single-use Macro Tree Transducers as \( \text{MAC}_{\text{su}} \).

Another important restriction of Macro Tree Transducers is that of having no context parameters at all, which gives us Top-Down Tree Transducers \([\text{Ron70}]\).

Definition 2.12 A Macro Tree Transducer is called a Top-Down Tree Transducer, if all its states have rank one, that is, there are no context parameters present. We denote the class of all tree transductions induced by Top-Down Tree Transducers as \( \text{TOP} \).

Example 2.13

- \( M_{\text{rev}} \) is both non-copying and weakly single-use, thus it is also single-use.
- \( M_{\text{top}} = (\{p^{(1)}, q^{(1)}\}, \Sigma_{\text{mon}}, \Sigma_{\text{mon}}, p(x_1), R_{\text{top}}) \) with set of rules \( R_{\text{top}} \):
  
  \[
  \begin{align*}
  (i) & : p(A(x_1)) \rightarrow A(y(x_1)) \\
  (ii) & : p(B(x_1)) \rightarrow q(x_1) \\
  (iii) & : p(E) \rightarrow E \\
  (iv) & : q(A(x_1)) \rightarrow p(x_1) \\
  (v) & : q(B(x_1)) \rightarrow B(q(x_1)) \\
  (vi) & : q(E) \rightarrow E
  \end{align*}
  \]

  is a Top-Down Tree Transducer and thus trivially non-copying, but it is not weakly single-use, because a call \( p(x_1) \) occurs twice in right-hand sides of rules at \( A \).

- The Macro Tree Transducer \( M' = (\{q^{(2)}\}, \Sigma_{\text{mon}}, \{\alpha^{(2)}, \beta^{(2)}, e^{(0)}\}, q(x_1, e), R') \) with set of rules \( R' \):
  
  \[
  \begin{align*}
  (i) & : q(A(x_1), y_1) \rightarrow q(x_1, \alpha(y_1, y_1)) \\
  (ii) & : q(B(x_1), y_1) \rightarrow q(x_1, \beta(y_1, y_1)) \\
  (iii) & : q(E, y_1) \rightarrow y_1
  \end{align*}
  \]

  is not non-copying, because the right-hand sides are not linear in context variables.

\[\Box\]

\footnote{Our main result would still hold if we generalize the property weakly single-use by dropping the second condition.}
The following proposition will later be used to identify Macro Tree Transducers with above properties.

**Proposition 2.14** Let \( M = (Q, \Sigma, \Delta, c, R) \) be a Macro Tree Transducer. \( M \) is non-copying, if for every \( \sigma \in \Sigma, r \in \mathbb{N}, q \in Q^{(r+1)} \) and \( h \in [r] \) holds: \( \#_{y_h}(\text{rhs}_{q,r}) \leq 1 \). If \( Q \) is a singleton set \( \{q\} \), then \( M \) is weakly single-use, if for every \( k \in \mathbb{N}, \sigma \in \Sigma^{(k)} \) and \( i \in [k] \) holds: \( \#x_i(\text{rhs}_{q,r}) \leq 1 \). \( \square \)

### 2.3 Proving a Lemma

In this section, we introduce the principle of proof by simultaneous induction and illustrate its application by proving a substitution lemma, which we will need later.

**Definition 2.15** Let \( \Sigma \) be a ranked alphabet. Given two statements (I) and (II), where (I) has a free variable \( t \in T_\Sigma \) and (II) has free variables \( k \in \mathbb{N} \) and \( t_1, \ldots, t_k \in T_\Sigma \), we say that (I) and (II) are proven by simultaneous induction, if we can prove the following two induction steps:

(I) \( \Leftrightarrow \) (II): for every \( k \in \mathbb{N} \), \( \sigma \in \Sigma^{(k)} \) and \( t_1, \ldots, t_k \in T_\Sigma \), such that (II) holds, we can show that (I) holds for \( t = \sigma(t_1, \ldots, t_k) \)

(II) \( \Leftrightarrow \) (I): for every \( k \in \mathbb{N} \) and \( t_1, \ldots, t_k \in T_\Sigma \), such that (I) holds for each of the \( t_1, \ldots, t_k \), we can show that (II) holds.

By proving these two induction steps, we have proven that (I) holds for every \( t \in T_\Sigma \) and that (II) holds for every \( k \in \mathbb{N} \) and \( t_1, \ldots, t_k \in T_\Sigma \). \( \square \)

As an example, we prove the following lemma by using simultaneous induction.

**Lemma 2.16** Let \( M = (Q, \Sigma, \Delta, c, R) \) be a Macro Tree Transducer. For every set \( A \) disjoint from \( \Delta \), \( t \in T_\Sigma \), \( r \in \mathbb{N} \), \( q \in Q^{(r+1)} \) and \( \theta_1, \ldots, \theta_r \in T_\Delta(A) \), we have:

\[
\tau_{M,q,t}(\theta_1, \ldots, \theta_r) = (\tau_{M,q,r}(t, 1, \ldots, r))[l \leftarrow \theta_l, 1 \leq l \leq r].
\]

**Proof:** We prove the following two statements by simultaneous induction.

(I) For \( t \in T_\Sigma \): for every set \( A \) with \( A \cap \Delta = \emptyset \), \( r \in \mathbb{N} \), \( q \in Q^{(r+1)} \) and \( \theta_1, \ldots, \theta_r \in T_\Delta(A) \) holds:

\[
\tau_{M,q,t}(\theta_1, \ldots, \theta_r) = (\tau_{M,q,r}(t, 1, \ldots, r))[l \leftarrow \theta_l, 1 \leq l \leq r]
\]

(II) For \( k \in \mathbb{N}, t_1, \ldots, t_k \in T_\Sigma \): for every \( A \cap \Delta = \emptyset \), \( r + 1 \in \text{rank}(Q) \), \( \theta_1, \ldots, \theta_r \in T_\Delta(A) \) and \( \phi \in \text{RHS}(Q, \Delta, k, Y_r) \) holds:

\[
\tau_{M,q,t}(\theta_1, \ldots, \theta_k, 1, \ldots, r) = (\tau_{M,q,r}(\phi, t_1, \ldots, t_k, 1, \ldots, r))[l \leftarrow \theta_l, 1 \leq l \leq r]
\]

(I) \( \Leftrightarrow \) (II): Let \( t = \sigma(t_1, \ldots, t_k) \) with \( k \in \mathbb{N} \), \( \sigma \in \Sigma^{(k)} \) and \( t_1, \ldots, t_k \in T_\Sigma \), then:

\[
\begin{align*}
\tau_{M,q,t}(\sigma(t_1, \ldots, t_k), \theta_1, \ldots, \theta_r) &= (\text{by definition of } \tau_{M,q,t}) \\
\tau_{M,q,t}(\text{rhs}_{q,r}, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r) &= (\text{by induction hypothesis (II) for } t_1, \ldots, t_k \text{ and } \phi = \text{rhs}_{q,r}) \\
(\tau_{M,q,r}(\text{rhs}_{q,r}, t_1, \ldots, t_k, 1, \ldots, r))[l \leftarrow \theta_l, 1 \leq l \leq r] &= (\text{by definition of } \tau_{M,q,r}) \\
(\tau_{M,q,r}(\sigma(t_1, \ldots, t_k), 1, \ldots, r))[l \leftarrow \theta_l, 1 \leq l \leq r]
\end{align*}
\]
(II) $\iff$ (I) : We prove the claim by structural induction on $\phi \in \text{RHS}(Q, \Delta, k, Y_r)$.

$\phi = y_h \in Y_r$:
\[
\begin{align*}
\tau_{M,k,r,A}(y_h, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r) &= (\text{by definition of } \tau_{M,k,r,A}) \\
\theta_h &= (\text{by substitution}) \\
h[l \leftarrow \theta_h, 1 \leq l \leq r] &= (\text{by definition of } \tau_{M,k,r,[r]}) \\
(\tau_{M,k,r,[r]}(y_h, t_1, \ldots, t_k, 1, \ldots, r))[l \leftarrow \theta_h, 1 \leq l \leq r]
\end{align*}
\]

$\phi = \delta(\phi_1, \ldots, \phi_a), a \in \mathbb{N}, \delta \in \Delta^{(a)}, \phi_1, \ldots, \phi_a \in \text{RHS}(Q, \Delta, k, Y_r)$:
\[
\begin{align*}
\tau_{M,k,r,A}(\delta(\phi_1, \ldots, \phi_a), t_1, \ldots, t_k, \theta_1, \ldots, \theta_r) &= (\text{by definition of } \tau_{M,k,r,A}) \\
\delta(\tau_{M,k,r,A}(\phi_1, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r), \ldots, \\
\tau_{M,k,r,A}(\phi_a, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r)) &= (\text{by induction hypothesis for } \phi_1, \ldots, \phi_a) \\
\delta((\tau_{M,k,r,[r]}(\phi_1, t_1, \ldots, t_k, 1, \ldots, r))[l \leftarrow \theta_h, 1 \leq l \leq r], \\
\ldots, \\
(\tau_{M,k,r,[r]}(\phi_a, t_1, \ldots, t_k, 1, \ldots, r))[l \leftarrow \theta_h, 1 \leq l \leq r]) &= (\text{by substitution}) \\
(\tau_{M,k,r,[r]}(\delta(\phi_1, \ldots, \phi_a), t_1, \ldots, t_k, 1, \ldots, r))[l \leftarrow \theta_h, 1 \leq l \leq r]
\end{align*}
\]

$\phi = q(x_i, \phi_1, \ldots, \phi_a), a \in \mathbb{N}, q \in \mathbb{Q}^{(a+1)}, x_i \in X_k, \phi_1, \ldots, \phi_a \in \text{RHS}(Q, \Delta, k, Y_r)$:
\[
\begin{align*}
\tau_{M,k,r,A}(q(x_i, \phi_1, \ldots, \phi_a), t_1, \ldots, t_k, \theta_1, \ldots, \theta_r) &= (\text{by definition of } \tau_{M,k,r,A}) \\
\tau_{M,k,r,A}(t_i, \tau_{M,k,r,A}(\phi_1, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r), \ldots, \\
\tau_{M,k,r,A}(\phi_a, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r)) &= (\text{by induction hypothesis (I) for } t_i) \\
(\tau_{M,k,r,[a]}(t_i, 1, \ldots, a))[h \leftarrow \tau_{M,k,r,A}(\phi_1, t_1, \ldots, t_k, \theta_1, \ldots, \theta_r), 1 \leq h \leq a] &= (\text{by induction hypothesis for } \phi_1, h \in [a]) \\
(\tau_{M,k,r,[a]}(t_i, 1, \ldots, a))[h \leftarrow (\tau_{M,k,r,[r]}(\phi_1, t_1, \ldots, t_k, 1, \ldots, r))][l \leftarrow \theta_h, 1 \leq l \leq r, 1 \leq h \leq a] &= (\text{by convention } [r] \cap \Delta = \emptyset \text{ and properties of substitution}) \\
((\tau_{M,k,r,[a]}(t_i, 1, \ldots, a))[h \leftarrow (\tau_{M,k,r,[r]}(\phi_1, t_1, \ldots, t_k, 1, \ldots, r))][l \leftarrow \theta_h, 1 \leq l \leq r]) &= (\text{by induction hypothesis (I) for } t_i) \\
(\tau_{M,k,r,[r]}(t_i, \tau_{M,k,r,[r]}(\phi_1, t_1, \ldots, t_k, 1, \ldots, r), \ldots, \\
\tau_{M,k,r,[r]}(\phi_a, t_1, \ldots, t_k, 1, \ldots, r)))[l \leftarrow \theta_h, 1 \leq l \leq r] &= (\text{by definition of } \tau_{M,k,r,[r]}) \\
(\tau_{M,k,r,[r]}(q(x_i, \phi_1, \ldots, \phi_a), t_1, \ldots, t_k, 1, \ldots, r))[l \leftarrow \theta_h, 1 \leq l \leq r] 
\end{align*}
\]