Authors' version of a paper appearing in Journal of Functional Programming **14** (3): 317–363, May 2004. © 2004 Cambridge University Press

Composition of functions with accumulating parameters

JANIS VOIGTLÄNDER∗ and ARMIN KÜHNEMANN
Department of Computer Science, Dresden University of Technology, D-01062 Dresden, Germany
(*e-mail:* {voigt,kuehne}@tcs.inf.tu-dresden.de)

Abstract

Many functional programs with accumulating parameters are contained in the class of macro tree transducers. We present a program transformation technique that can be used to solve the efficiency problems due to creation and consumption of intermediate data structures in compositions of such functions, where classical deforestation techniques fail.

In order to do so, given two macro tree transducers under appropriate restrictions, we construct a single macro tree transducer that implements the composition of the two original ones. The imposed restrictions are more liberal than those in the literature on macro tree transducer composition, thus generalising previous results.

1 Introduction

An important style of writing programs in a functional language is to define new functions by composition of existing ones. Thus, the result of a function application is passed as argument to another function. This modular programming technique of solving an overall problem by combining solutions of partial problems simplifies the design and verification of programs and encourages reuse. Unfortunately, modular programs often lack efficiency compared to other—often less understandable—programs that solve the same tasks. 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 more data structure traversals are performed than would really be necessary for solving the overall problem. Thus, one would like to have program transformation techniques that allow the optimisation of functions written in the modular style by eliminating intermediate data structures. Several such techniques have been studied in the literature, e.g., the *unfold/fold-technique* by Burstall & Darlington (1977), its algorithmic instances *supercompilation* (Turchin, 1986; Sørensen *et al.*, 1996; Secher & Sørensen, 1999) and *deforestation* (Wadler, 1990; Chin, 1994), *program calculation* (Malcolm, 1989; Meijer *et al.*, 1991; Sheard & Fegaras, 1993; Hu *et al.*, 1996; Bird & de Moor, 1997) and *shortcut deforestation* (Gill *et al.*, 1993; Gill, 1996).

∗ This author was supported by the DFG under grant KU 1290/2-1.

In this paper we follow an approach for eliminating intermediate results that is based on the theory of tree transducers (Fülöp & Vogler, 1998). Particularly, we consider *macro tree transducers* (for short *mtts*; Engelfriet, 1980), which are extended schemes of primitive recursion—allowing simultaneous definition of several functions and nested function calls in parameter positions—that translate trees over a ranked alphabet of input symbols into trees over a ranked alphabet of output symbols. For this translation process an mtt uses functions that have at least rank one, and a set of rewrite rules. Every function f is defined by pattern matching on the root symbol σ of its first argument t. The right-hand side of the rule for f at σ may contain the other arguments of f, output symbols, and recursive function calls the first arguments of which must be variables that refer to subtrees of t. Since many typical functions on algebraic data types are defined by such a structural descent, mtts represent a large class of functional programs using accumulating parameters, which in the scope of mtts are called *context parameters*.

For illustration of the problem of intermediate results, 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 constructor symbols A and B and binary constructor symbols $+$ and $\times$ (this representation corresponds to an algebraic data type in a functional language, e.g., `data Term = A | B | + Term Term | × Term Term` in Haskell-style). Further, assume given a function *pfx* for computing the prefix-notation of such a term as monadic tree of now unary symbols A, B, $+$, $\times$, and a nullary ϵ as end symbol:

$$\begin{array}{rlcl}
\text{(i)} : & pfx(+(u_1,u_2),y_1) & \to & +(pfx(u_1,pfx(u_2,y_1))) \\
\text{(ii)} : & pfx(\times(u_1,u_2),y_1) & \to & \times(pfx(u_1,pfx(u_2,y_1))) \\
\text{(iii)} : & pfx(A,y_1) & \to & A(y_1) \\
\text{(iv)} : & pfx(B,y_1) & \to & B(y_1).
\end{array}$$

In order to compute the prefix-notation of a term t, the rules (i)–(iv) are used to exhaustively rewrite the initial expression $pfx(t,\epsilon)$.

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 programming a solution for this problem from scratch, we can solve the task by simply reversing the prefix-notation of the given term and replacing labels A, B, $+$ and $\times$ by $LOAD_A$, $LOAD_B$, ADD and MUL, respectively. Hence, we define the following auxiliary function:

$$\begin{array}{rlcl}
\text{(v)} : & aux(A(v_1),z_1) & \to & aux(v_1,LOAD_A(z_1)) \\
\text{(vi)} : & aux(B(v_1),z_1) & \to & aux(v_1,LOAD_B(z_1)) \\
\text{(vii)} : & aux(+(v_1),z_1) & \to & aux(v_1,ADD(z_1)) \\
\text{(viii)} : & aux(\times(v_1),z_1) & \to & aux(v_1,MUL(z_1)) \\
\text{(ix)} : & aux(\epsilon,z_1) & \to & z_1.
\end{array}$$

The instruction sequence for a given term t can now be computed by composing the functions *pfx* and *aux*, namely by rewriting the composite expression $aux(pfx(t,\epsilon),\epsilon)$ with rules (i)–(ix).

However, this modular solution is inefficient, because it creates and consumes an intermediate result. Depending on the used evaluation strategy, 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 lazy evaluation, this program performs a superfluous traversal through the intermediate result.

To avoid these inefficiencies, we could instead use the following function definition:

$$\begin{array}{rcl}
(\text{x}) &:& ins(+(u_1,u_2),z_1) \;\rightarrow\; ins(u_2, ins(u_1, ADD(z_1))) \\
(\text{xi}) &:& ins(\times(u_1,u_2),z_1) \;\rightarrow\; ins(u_2, ins(u_1, MUL(z_1))) \\
(\text{xii}) &:& ins(A,z_1) \;\rightarrow\; LOAD_A(z_1) \\
(\text{xiii}) &:& ins(B,z_1) \;\rightarrow\; LOAD_B(z_1).
\end{array}$$

If for a given term t we use rules (x)–(xiii) on $ins(t,\epsilon)$, we will calculate the same instruction sequence as before with the modular solution, but without creating and traversing the intermediate data structure, thus requiring less rewrite steps.

Consequently, it would be worthwhile to automatically transform the modular solution (i)–(ix) into the efficient solution (x)–(xiii). To the best of our knowledge, techniques such as deforestation and program calculation cannot perform the optimisation that we want to achieve here. In particular, classical deforestation fails due to its well-known problem of not reaching accumulating parameters (Chin, 1994). An approach that is applicable to our example is based on *attribute grammars* (Knuth, 1968) and was proposed independently by Kühnemann (1997; 1998) and Correnson *et al.* (1998; 1999). The idea is to transform the two functions (in our formalism represented by two restricted mtts) that we want to compose, into attribute grammars (Courcelle & Franchi-Zannettacci, 1982), respectively *attributed tree transducers* (Fülöp, 1981), which are abstractions of attribute grammars. If the first attributed tree transducer fulfils the *single-use* restriction, 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 (Ganzinger, 1983; Ganzinger & Giegerich, 1984; Giegerich, 1988)). Applying a construction based on (Franchi-Zannettacci, 1982), this attributed tree transducer can then be transformed into an mtt, thus giving a functional program for the composition of the two original functions, but without producing and consuming the intermediate result. This and related techniques of combining results from the theory of tree transducers for functional program optimisation are presented in a uniform framework by Kühnemann & Voigtländer (2001). A restricted instance of attribute grammar composition is also handled by Kakehi *et al.* (2001), using a single local rule to eliminate intermediate lists in compositions of `map`-style list transformers.

The approach of transformation by composition of attribute grammars does not work for all mtts, because only restricted mtts can be transformed into attributed tree transducers. One such restriction is the property of an mtt to be *weakly single-use* (Kühnemann, 1998), which roughly speaking means that at every node in the input tree recursive calls of functions on subtrees are restricted to appear at most

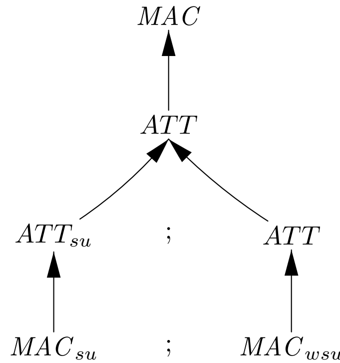

Fig. 1. Indirect composition.

once[1]. Moreover, in order to obtain a single-use attributed tree transducer—such that the composition result for attributed tree transducers becomes applicable—the first mtt 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. The latter restriction is called *non-copying*. Figure 1 shows the transformation steps that one has to take, where arrows indicate transformations and the semicolon stands for composition of tree transduction classes. Here, the class of functions computable by mtts 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 su and wsu, respectively.

There are several reasons to be interested in a direct construction for composing mtts without the above indirection. Firstly, there would be benefits for an implementation, 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 optimisation's sake. Secondly, a direct construction could produce better program code than the indirection by several transformations, because—for the sake of generality—every single program transformation tends to introduce a certain "ballast" into the program, like, e.g., superfluous function parameters. Thirdly, a direct composition construction on the level of functional programs is more accessible to formal efficiency comparisons with other program transformation techniques. Finally, we want to broaden the applicability of mtt composition by generalising the result $MAC_{su}; MAC_{wsu} \subseteq MAC$.

It is well known that one cannot compose two arbitrary mtts, because the class of macro tree transductions is not closed under composition (Engelfriet & Vogler, 1985). But, we can aim for weaker restrictions on the mtts than necessary so far, which still allow a composition. The success of this intention is limited if we stick

[1] Other restrictions of mtts that allow the transformation into attributed tree transducers are, e.g., the *well-presented* restriction (Courcelle & Franchi-Zannettacci, 1982), the *attributed-like* restriction (Fülöp & Vogler, 1999) and the *restricted-use* condition (Kühnemann & Voigtländer, 2001), which generalises the weakly single-use property.

to the above indirect construction via attributed tree transducers, because then we can only compose mtts that can be transformed into attributed tree transducers, which is not always possible as there is a *strict* inclusion $ATT \subset MAC$ (Engelfriet, 1980; Franchi-Zannettacci, 1982). A direct composition construction has no such a priori limitation. In fact, we prove—with MAC_{nc} being the class of functions computable by non-copying mtts[2]—the inclusion $MAC_{nc}; MAC_{wsu} \subseteq MAC$, which is a generalisation of the result quoted above, because single-use mtts are, by definition, non-copying. We will observe that the direct composition construction allows us to further weaken the restriction on the second mtt by requiring the weakly single-use condition only for those of its functions that have at least one context parameter. The presented construction is also applicable if one of the two mtts has no context parameters—i.e., is a top-down tree transducer (Rounds, 1970; Thatcher, 1970; Engelfriet, 1975)—and the other one is an unrestricted mtt, thus incorporating into the new construction two known results (Engelfriet, 1981; Engelfriet & Vogler, 1985).

The direct composition construction together with two post-processing constructions developed by Voigtländer (2001) are then able to transform the above modular program—rules (i) to (ix)—into the more efficient program of rules (x)–(xiii).

The remainder of this paper is organised as follows. We define necessary notions in Section 2 and introduce the basic concepts of mtts in Section 3. In Section 4 we discuss the underlying ideas for the direct composition construction of a non-copying and a weakly single-use mtt, which is formally given in Section 5. In Section 6 we consider practical aspects of the direct composition construction, namely a post-processing phase and an implementation. Section 7 compares our technique with related approaches for eliminating intermediate results. In Section 8 we discuss efficiency aspects of our transformation technique, with respect to abstract efficiency measures and by runtime measurements. In Section 9 we show that a symmetric composition—a general construction for composing a weakly single-use mtt and a non-copying one in this order—cannot exist, and give some theoretical results on mtts. Finally, Section 10 concludes with an outlook for future research. The correctness proof for the composition construction can be found in the Appendix, available online as (Voigtländer & Kühnemann, 2003).

2 Preliminaries

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

We use several sets of lowercase variables. We denote by U the set $\{u_1, u_2, u_3, \ldots\}$ of variables, and for $p \in \mathbb{N}$ by U_p the finite set $\{u_1, \ldots, u_p\} \subseteq U$; analogous for V, Y, Z and $Y' = \{y'_1, y'_2, y'_3, \ldots\}$.

For a set S, we denote by S^* the set of finite sequences of elements of S, where ε denotes the empty sequence. The power set of a given set S will be denoted by

[2] Note that in Theorem 9.6 of this paper $MAC_{nc} \not\subseteq ATT$ will be shown. Hence, the indirect construction cannot be used for non-copying mtts.

$\mathcal{P}(S)$. The number of elements of a finite set S will be denoted by $|S|$. For a set S and a binary relation $\lhd \subseteq S \times S$, we denote by $\lhd^+$ the transitive and by $\lhd^*$ the reflexive, transitive closure of $\lhd$, respectively.

We define substitution over strings (elements of S^* for some finite set S) as follows. For a string $w \in S^*$, pairwise different symbols $x_1, \ldots, x_n \in S$, and strings $w_1, \ldots, w_n \in S^*$ (for some $n \in \mathbb{N}$), we denote by $w[x_i \leftarrow w_i, i \in [n]]$ the string obtained from w by replacing all occurrences of every x_i in w by w_i. We will also use the alternative notation $w[x_1, \ldots, x_n \leftarrow w_1, \ldots, w_n]$ and appropriate multi-line notations for long substitutions. We write substitutions left-associative.

A *ranked alphabet* is a pair $(\Sigma, rank_\Sigma)$, where Σ is a finite set of symbols and $rank_\Sigma$ assigns to each of these symbols a natural number, its *rank*. In the following, we will drop the $rank_\Sigma$-function from the denotation and only mention Σ when referring to a ranked alphabet. For every $p \in \mathbb{N}$, we define $\Sigma^{(p)} = \{\sigma \in \Sigma \mid rank_\Sigma(\sigma) = p\}$. The rank p of a symbol σ will also be denoted by writing $\sigma^{(p)}$. For the sake of brevity, quantifications over a symbol in a ranked alphabet will implicitly quantify also over the rank of the symbol. E.g., we will write "for every $\sigma \in \Sigma^{(p)}$" instead of "for every $p \in \mathbb{N}$, $\sigma \in \Sigma^{(p)}$" and "there exists $f \in F^{(r+1)}$" instead of "there exist $r \in \mathbb{N}$ and $f \in F^{(r+1)}$". For a ranked alphabet Σ, we denote the set of all its ranks as $rank(\Sigma) = \{p \in \mathbb{N} \mid \exists \sigma \in \Sigma : rank_\Sigma(\sigma) = p\}$.

For a ranked alphabet Σ and a set S disjoint from Σ, we define the set $T_\Sigma(S)$ of *trees over* Σ *indexed by* S as the smallest set $T \subseteq (\Sigma \cup S \cup \{(,)\} \cup \{,\})^*$ such that (i) $S \subseteq T$ and (ii) for every $\sigma \in \Sigma^{(p)}$ and $t_1, \ldots, t_p \in T$: $\sigma(t_1, \ldots, t_p) \in T$. For nullary symbols, we simply write α instead of $\alpha()$. We denote the set $T_\Sigma(\emptyset)$ by T_Σ.

Let Σ be a ranked alphabet and X a set of variables, where $\Sigma \cap X = \emptyset$. A *rewrite rule* over Σ and X is a rule of the form $lhs \to rhs$ with $lhs, rhs \in T_\Sigma(X)$, such that the left-hand side lhs does not contain two occurrences of the same variable and every variable occurring in the right-hand side rhs is also contained in lhs. A set R of rewrite rules over Σ and X is called a *rewrite system* (over Σ and X)[3] (Dershowitz & Jouannaud, 1990; Baader & Nipkow, 1998). For every $\Sigma' \supseteq \Sigma$, it induces a binary *reduction relation* $\Rightarrow_R \subseteq T_{\Sigma'} \times T_{\Sigma'}$, such that $t \Rightarrow_R t'$ iff R contains a rule $lhs \to rhs$, there is a tree $c \in T_{\Sigma'}(\{x\})$ (with $x \notin X$) that contains x exactly once, and there exist $n \in \mathbb{N}$, trees $t_1, \ldots, t_n \in T_{\Sigma'}$ and pairwise different variables $x_1, \ldots, x_n \in X$ such that:

$$\begin{aligned} t &= c[x \leftarrow lhs\,[x_1, \ldots, x_n \leftarrow t_1, \ldots, t_n]] \\ t' &= c[x \leftarrow rhs[x_1, \ldots, x_n \leftarrow t_1, \ldots, t_n]]. \end{aligned}$$

A reduction relation $\Rightarrow_R \subseteq T_\Sigma \times T_\Sigma$ is called *confluent*, if for every $t, t_1, t_2 \in T_\Sigma$ with $t \Rightarrow_R^* t_1$ and $t \Rightarrow_R^* t_2$, there exists $t' \in T_\Sigma$ with $t_1 \Rightarrow_R^* t'$ and $t_2 \Rightarrow_R^* t'$. A reduction relation $\Rightarrow_R$ is called *terminating*, if there is no infinite chain $t_1 \Rightarrow_R t_2 \Rightarrow_R t_3 \Rightarrow_R \ldots$. If $t \Rightarrow_R^* t'$ and there is no t'' with $t' \Rightarrow_R t''$, then t' is called a *normal form* of t with respect to $\Rightarrow_R$. If $\Rightarrow_R$ is confluent and terminating, then every tree $t \in T_\Sigma$ has a unique normal form, denoted as $nf(\Rightarrow_R, t)$.

[3] When Σ and X are clear from the context, we will only mention R. Note that a rewrite system over Σ and X is also a rewrite system over $\Sigma' \supseteq \Sigma$ and $X' \supseteq X$, if $\Sigma' \cap X' = \emptyset$.

Let Σ, Δ and Ω be ranked alphabets. We call a total function $\tau : T_\Sigma \to T_\Delta$ a *tree transduction (from T_Σ to T_Δ)*. 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 $\tau_1;\tau_2$, as $(\tau_1;\tau_2)(t) = \tau_2(\tau_1(t))$, 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 $\mathcal{T}_1;\mathcal{T}_2 = \{\tau_1;\tau_2 \mid \tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2\}$.

Let Σ be a ranked alphabet and S a set disjoint from Σ. We will need the *set of paths* in a tree, given by the function $paths : T_\Sigma(S) \to \mathcal{P}((\mathbb{N}_+)^*)$, which is defined by structural recursion as follows: (i) if $t \in \Sigma^{(0)} \cup S$, then $paths(t) = \{\varepsilon\}$, and (ii) if $t = \sigma(t_1, \ldots, t_p)$ with $p \in \mathbb{N}_+$, $\sigma \in \Sigma^{(p)}$, $t_1, \ldots, t_p \in T_\Sigma(S)$, then $paths(t) = \{\varepsilon\} \cup \{i\pi \mid i \in [p], \pi \in paths(t_i)\}$.

We will also need the *label at a path in a tree*, given by the mapping $lab : \{(t, \pi) \mid t \in T_\Sigma(S), \pi \in paths(t)\} \to \Sigma \cup S$, defined by: (i) if $t \in \Sigma^{(0)} \cup S$, then $lab(t, \varepsilon) = t$, and (ii) if $t = \sigma(t_1, \ldots, t_p)$ with $p \in \mathbb{N}_+$, $\sigma \in \Sigma^{(p)}$, $t_1, \ldots, t_p \in T_\Sigma(S)$, then $lab(t, \varepsilon) = \sigma$ and $lab(t, i\pi) = lab(t_i, \pi)$ for $i \in [p]$ and $\pi \in paths(t_i)$. The label $lab(t, \varepsilon)$ is called the *root symbol* of the tree t.

The *subtree at a path in a tree* is given by the function $sub : \{(t, \pi) \mid t \in T_\Sigma(S), \pi \in paths(t)\} \to T_\Sigma(S)$, defined by: (i) $sub(t, \varepsilon) = t$, and (ii) if $t = \sigma(t_1, \ldots, t_p)$ with $p \in \mathbb{N}_+$, $\sigma \in \Sigma^{(p)}$, $t_1, \ldots, t_p \in T_\Sigma(S)$, then $sub(t, i\pi) = sub(t_i, \pi)$ for $i \in [p]$ and $\pi \in paths(t_i)$.

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

3 Macro Tree Transducers

In this section we introduce macro tree transducers and syntactic restrictions for them, both formally and using examples. Then, we present a new characterisation relating the class of functions computable by unrestricted macro tree transducers and the composition of two restricted classes.

3.1 Definitions and Examples

Definition 3.1 (*macro tree transducer, RHS*)
A *macro tree transducer* (for short *mtt*) M is a tuple $(F, \Sigma, \Delta, e, R)$ with:

- a ranked alphabet F of *states*, where $F^{(0)} = \emptyset$
- a ranked alphabet Σ of *input symbols*, where $\Sigma^{(0)} \neq \emptyset$ and $F \cap \Sigma = \emptyset$
- a ranked alphabet Δ of *output symbols*, where $\Delta^{(0)} \neq \emptyset$ and $F \cap \Delta = \emptyset$
- an *initial expression* $e \in RHS(F, \Delta, \{x\}, \emptyset)$
- a set R containing for every $f \in F^{(r+1)}$ and $\sigma \in \Sigma^{(p)}$ exactly one *rule* of the form $f(\sigma(u_1, \ldots, u_p), y_1, \ldots, y_r) \to rhs_{f,\sigma}$, with $rhs_{f,\sigma} \in RHS(F, \Delta, U_p, Y_r)$,

where for sets X and X', the set $RHS(F, \Delta, X, X')$ is the smallest set $RHS \subseteq T_{F\cup\Delta}(X \cup X')$ satisfying the following conditions:

- $X' \subseteq RHS$
- for every $\delta \in \Delta^{(q)}$ and $\phi_1, \ldots, \phi_q \in RHS$: $\delta(\phi_1, \ldots, \phi_q) \in RHS$
- for every $f \in F^{(r+1)}$, $x \in X$, $\phi_1, \ldots, \phi_r \in RHS$: $f(x, \phi_1, \ldots, \phi_r) \in RHS$. $\diamond$

Note that R in the above definition is a rewrite system over $F \cup \Sigma \cup \Delta$ and $U \cup Y$. A rule of the form $f(\sigma(\ldots), \ldots) \to \ldots$ is also called a *σ-rule*. A subtree of the form $f(t, \ldots)$ is referred to as a *call* of f on t. The first argument of a state f is called *recursion argument*, the others are called *context parameters*. Correspondingly, variables from U are called *recursion variables* and variables from Y are called *context variables*. Of course, the actual variable names used in mtt rules are not fixed to come from U_p and Y_r for some $p, r \in \mathbb{N}$; consistent renaming is allowed. For example, we will later use recursion variables from V and context variables from Z for the second mtt in the composition construction.

Example 3.2 (*the functions pfx and aux from the introduction as mtts*)
Let $\Sigma_{term} = \{+^{(2)}, \times^{(2)}, A^{(0)}, B^{(0)}\}$, $\Delta_{list} = \{+^{(1)}, \times^{(1)}, A^{(1)}, B^{(1)}, \epsilon^{(0)}\}$ and $\Omega_{ins} = \{ADD^{(1)}, MUL^{(1)}, LOAD_A^{(1)}, LOAD_B^{(1)}, \epsilon^{(0)}\}$.

We define the mtt $M_{pfx} = (\{pfx^{(2)}\}, \Sigma_{term}, \Delta_{list}, e_{pfx}, R_{pfx})$, where R_{pfx} contains the rules (i)–(iv) from the introduction, and $e_{pfx} = pfx(x, \epsilon)$. We also define the mtt $M_{aux} = (\{aux^{(2)}\}, \Delta_{list}, \Omega_{ins}, aux(x, \epsilon), R_{aux})$, where R_{aux} contains the rules (v)–(ix) from the introduction. $\diamond$

The semantics of an mtt is a function from trees over the input ranked alphabet to trees over the output ranked alphabet. It is given by substituting the input tree for x in the initial expression e of an mtt, and then calculating the normal form of this expression with respect to the reduction relation induced by the set R of rules. This normal form exists and is unique, because the rules of an mtt induce a confluent and terminating reduction relation (cf., e.g., Fülöp & Vogler, 1998).

Definition 3.3 (*semantics of an mtt*)
The *tree transduction induced by an mtt* $M = (F, \Sigma, \Delta, e, R)$ is the total function $\tau(M) : T_\Sigma \to T_\Delta$ that assigns to every tree $t \in T_\Sigma$ the value $nf(\Rightarrow_R, e[x \leftarrow t])$. We denote the class of *tree transductions induced by mtts* as MAC. $\diamond$

Although the mtts presented here have a more general initial expression than those used by Fülöp & Vogler (1998)—allowing just the call of some state with fixed context parameters—our tree transduction class MAC coincides with theirs, because mtts in the two representations can be transformed into each other. Note that the same needs not be true for restricted classes of tree transductions, because our representation is more flexible.

Though this paper does not perform formal efficiency considerations, we sometimes informally argue with respect to efficiency. For this purpose we fix lazy evaluation as our intended deterministic reduction strategy.

Example 3.4 (*semantics of* M_{pfx})
We apply the mtt M_{pfx} from Example 3.2 to the input term $+(A,B)$, i.e., we calculate the normal form of $e_{pfx}[x \leftarrow +(A,B)]$ with respect to $\Rightarrow_{R_{pfx}}$:

$$pfx(+(A,B),\epsilon) \Rightarrow_{R_{pfx}} +(pfx(A, pfx(B,\epsilon))) \Rightarrow_{R_{pfx}} +(A(pfx(B,\epsilon))) \Rightarrow_{R_{pfx}} +(A(B(\epsilon))). \quad \diamond$$

We introduce two important syntactic restrictions of mtts (Kühnemann, 1998) and the corresponding classes of tree transductions induced by such restricted mtts.

Definition 3.5 (*non-copying*)
An mtt is *non-copying*, if there is at most one occurrence of every context variable in the right-hand side of every rule. We denote the class of *tree transductions induced by non-copying mtts* as MAC_{nc}. $\diamond$

The following property will later be required of the second mtt involved in a composition, hence we already here define it according to the notational conventions used for this second mtt, namely with states in G, input symbols in Δ, output symbols in Ω, recursion variables in V and context variables in Z.

Definition 3.6 (*weakly single-use*)
An mtt $M = (G, \Delta, \Omega, e, R)$ is *weakly single-use*, if the following two conditions hold:

(i) For every $\delta \in \Delta^{(q)}$, $j \in [q]$ and $g \in G$, a call of the form $g(v_j, \ldots)$ occurs in a right-hand side of at most one δ-rule and there only once.
(ii) For every $g \in G$, the initial expression e contains at most one occurrence of a call $g(x, \ldots)$.

We denote the class of *tree transductions induced by weakly single-use mtts* as MAC_{wsu}. $\diamond$

We also identify mtts that have both of the above properties.

Definition 3.7 (*single-use*)
An mtt is *single-use*, if it is both non-copying and weakly single-use. We denote the class of *tree transductions induced by single-use mtts* as MAC_{su}. $\diamond$

We use some more examples to illustrate the introduced restrictions of mtts.

Example 3.8 (*restricted mtts*)

1. The mtts M_{pfx} and M_{aux} from Example 3.2 are both non-copying and weakly single-use, hence they are also single-use.
2. Let $\Delta_{bin} = \{\delta^{(2)}, \epsilon^{(0)}\}$ and $Nat = \{succ^{(1)}, zero^{(0)}\}$. Then the mtt $M_{count} = (\{count^{(2)}\}, \Delta_{bin}, Nat, count(x, zero), R_{count})$ with set of rules

$$\begin{array}{lcl} count(\delta(u_1,u_2),y_1) & \rightarrow & succ(count(u_1, count(u_2,y_1))) \\ count(\epsilon, y_1) & \rightarrow & succ(y_1) \end{array}$$

is single-use. Note that for every $t \in T_{\Delta_{bin}}$, we have: $height(\tau(M_{count})(t)) = size(t)$. This statement is an instance of a more general one for arbitrary ranked alphabets in Construction 9.3.

3. The mtt $M_{exp} = (\{exp^{(2)}\}, Nat, Nat, exp(x, zero), R_{exp})$ with set of rules

$$\begin{array}{lcl} exp(succ(v_1), z_1) & \to & exp(v_1, exp(v_1, z_1)) \\ exp(zero, z_1) & \to & succ(z_1) \end{array}$$

   is non-copying, but it is not weakly single-use. Note that for every $t \in T_{Nat}$, we have: $height(\tau(M_{exp})(t)) = 2^{height(t)}$. This statement can easily be proven by induction.
4. The mtt $M_{bin} = (\{bin^{(2)}\}, \Delta_{bin}, \Delta_{bin}, bin(x, \epsilon), R_{bin})$ with set of rules

$$\begin{array}{lcl} bin(\delta(v_1, v_2), z_1) & \to & bin(v_1, bin(v_2, z_1)) \\ bin(\epsilon, z_1) & \to & \delta(z_1, z_1) \end{array}$$

   is weakly single-use, but it is not non-copying. Note that for every fully balanced binary tree $t \in T_{\Delta_{bin}}$ of height h, $\tau(M_{bin})(t)$ is a fully balanced binary tree of height 2^h, and hence $size(\tau(M_{bin})(t)) = 2^{(2^h+1)} - 1$. This statement can easily be proven by induction. $\diamond$

Kühnemann & Voigtländer (2001) give further examples and show how functions on polymorphic data types—such as the well-known Haskell `reverse` and `(++)` on lists—are handled as mtts by using enriched constructor symbols, and how functions such as `map` and `foldr` can be viewed as mtts by using the idea of *higher-order macros* (Wadler, 1990).

Another important restriction of mtts is that of having no context parameters at all, which gives us top-down tree transducers (Rounds, 1970; Thatcher, 1970; Engelfriet, 1975).

Definition 3.9 (*top-down tree transducer*)
An mtt is a *top-down tree transducer* (for short *tdtt*), if all its states have rank one. We denote the class of *tree transductions induced by tdtts* as TOP. $\diamond$

Note that no context parameters appear in the rules of a tdtt, hence every tdtt is, by definition, non-copying.

3.2 A Characterisation

The aim of this paper is to construct for two given mtts—of which the first one is non-copying and the second one is weakly single-use—a single new mtt that implements the composition of the two original ones. We give an a priori justification of the feasibility of this aim by reasoning with the help of other tree transduction classes (Fülöp, 1981; Kühnemann, 1997), which are not covered in this paper (ATT, $YIELD$ and ATT_{su} denote the classes of attributed tree transductions, *yield*-functions and tree transductions induced by single-use attributed tree transducers, respectively).

Theorem 3.10
$MAC_{nc}; MAC_{wsu} = MAC$

Proof sketch

	$MAC_{nc}; MAC_{wsu}$	
$\subseteq$	$TOP; ATT_{su}; MAC_{wsu}$	(an analogue of Lemma 5.3 in (Kühnemann, 1998))
$\subseteq$	$TOP; ATT_{su}; ATT$	(Theorem 7.1 in (Kühnemann, 1998))
$\subseteq$	$TOP; ATT$	(Lemma 6.4 in (Kühnemann, 1997))
$\subseteq$	$TOP; MAC$	(cf. Franchi-Zannettacci (1982), also Lemma 6.1 in (Fülöp & Vogler, 1998))
$\subseteq$	MAC	(Corollary 4.10 in (Engelfriet & Vogler, 1985))
$\subseteq$	$TOP; YIELD$	(Theorem 3 in (Engelfriet, 1980))
$\subseteq$	$MAC_{nc}; MAC_{wsu}$	($TOP \subseteq MAC_{nc}$; Example 4.5 in (Engelfriet & Vogler, 1985), Lemma 6.10 in (Kühnemann, 1997)) □

The same kind of reasoning can be applied to obtain new results on well-presented mtts (Courcelle & Franchi-Zannettacci, 1982) and attributed-like mtts (Fülöp & Vogler, 1999)—namely, $MAC_{nc}; MAC_{wp} = MAC = MAC_{nc}; MAC_{al}$ [4]—as well as on classes of macro attributed tree transducers (Kühnemann & Vogler, 1994).

The precise proof for the inclusion $MAC_{nc}; MAC_{wsu} \subseteq MAC$ will be given by presenting an effective composition construction in Subsection 5.1 and proving its correctness in the Appendix (Voigtländer & Kühnemann, 2003). Before giving this formal construction, we provide an intuitive explanation of the underlying ideas in the next section.

4 Ideas of the Direct Composition Construction

Given two mtts M_1 (from T_Σ to T_Δ, with rules R_1) and M_2 (from T_Δ to T_Ω, with rules R_2) with respective sets of states F and $G = \{g_1, \dots, g_\mu\}$, we want to create an mtt $M_{1;2}$ (from T_Σ to T_Ω, with rules $R_{1;2}$) such that $\tau(M_{1;2}) = \tau(M_1); \tau(M_2)$.

This aim cannot be achieved in general, but in the following subsections we will step by step study increasingly weaker restrictions under which it is feasible.

4.1 Translating Right-Hand Sides of M_1 with Rules of M_2

We first consider the very simple case that both M_1 and M_2 are tdtts with only one state (so called homomorphism tree transducers), i.e., $F = \{f_1^{(1)}\}$ and $G = \{g_1^{(1)}\}$. Then, the *define/instantiate/unfold/fold*-strategy (Burstall & Darlington, 1977) can be used to construct the required $M_{1;2}$ as a tdtt with exactly one state h, the basic idea being that of *folding* nested calls of the form $g_1(f_1(t))$ to $h(t)$. Therefor, $M_{1;2}$ must *define* appropriately *instantiated* rules for the new state h, namely for every input symbol $\sigma \in \Sigma^{(p)}$ a rule with left-hand side $h(\sigma(u_1, \dots, u_p))$ has to be constructed. Using the facts that the previous should correspond to

[4] $MAC_{nc}; MAC_{wp} \subseteq MAC_{nc}; MAC_{al} \subseteq MAC_{nc}; ATT \subseteq TOP; ATT_{su}; ATT \subseteq MAC \subseteq TOP; YIELD \subseteq MAC_{nc}; MAC_{wp}$, cf. the trivial inclusion $MAC_{wp} \subseteq MAC_{al}$, Lemma 3.18 in (Fülöp & Vogler, 1999) and the construction from Example 4.5 in (Engelfriet & Vogler, 1985).

$g_1(\underline{f_1(\sigma(u_1,\ldots,u_p))})$ and that a rule $f_1(\sigma(u_1,\ldots,u_p)) \to rhs_{f_1,\sigma}$ is given in R_1, an *unfold*-step yields

$$h(\sigma(u_1,\ldots,u_p)) \to g_1(\underline{rhs_{f_1,\sigma}})$$

as a good candidate. In order to obtain a legal tdtt rule from this candidate, $rhs_{f_1,\sigma}$ can be translated with g_1 by applying further *unfold*-steps using rules for g_1 at symbols from the intermediate ranked alphabet Δ, and *fold*-steps as introduced above. Hence, the actual right-hand side for the rule of h at σ is obtained by reducing $g_1(rhs_{f_1,\sigma})$ with the following rewrite systems:

R_2 :	$g_1(\delta(\ldots)) \to rhs_{g_1,\delta}\,,$	$\forall \delta \in \Delta$
Fold :	$g_1(f_1(u)) \to h(u)$	

Note that it will be exactly the applications of the rules from R_2 during the translation of $rhs_{f_1,\sigma}$ with g_1 that lead to the elimination of intermediate data structures.

4.2 Pairing of States

If M_1 and M_2 are tdtts with possibly non-singleton state sets, the strategy from the previous subsection has to be adjusted, because for each pair of states $f \in F$ and $g \in G$ a different new state must be used for folding nested calls of the form $g(f(t))$, hence h alone is not enough. The solution is fairly simple by using as states for the tdtt $M_{1;2}$ the set of pairs $H = \{(f,g) \mid f \in F, g \in G\}$ and folding calls of the form $g(f(t))$ to $(f,g)(t)$. The right-hand side of the rule for such a paired state (f',g') at an input symbol σ is then obtained by reducing $g'(rhs_{f',\sigma})$ with R_2 and a new rewrite system *Pair* that replaces *Fold*:

R_2 :	$g(\delta(\ldots)) \to rhs_{g,\delta}\,,$	$\forall g \in G, \delta \in \Delta$
Pair :	$g(f(u)) \to (f,g)(u)\,,$	$\forall g \in G, f \in F$

The transformation we have described thus is exactly the product construction for tdtts from the proof of Theorem 2 in (Rounds, 1970). Note that for every rule of R_1, exactly $\mu = |G|$ new rules are constructed, i.e., $|R_{1;2}| = \mu \cdot |R_1|$.

4.3 Adding Accumulating Parameters to the Producer — Reaching Accumulating Parameters with Unary States

Since we are interested in composing mtts and not only tdtts, we consider the case that $f \in F$ has a rank greater than one and a call of the form $g(f(t,\phi_1,\ldots,\phi_r))$ arises from running M_1 and M_2 independently (M_2 is still assumed to be a tdtt, hence $g \in G^{(1)}$). Following the idea of pairing states, such a nested call should correspond to a call of (f,g) on t during the computation of $M_{1;2}$. But it is far from

trivial how this corresponding call of (f,g) must look like, in particular how the $\phi_1,\ldots,\phi_r$ should be treated. Simply to pass these context parameters of f to the state (f,g) is not enough, as the following example shows.

Example 4.1 (*passing the context parameters of f unchanged to (f,g) fails*)
Assume that R_1 contains the rule $f(\alpha,y_1) \to \gamma(y_1)$ and R_2 contains the rule $g(\gamma(v_1)) \to g'(v_1)$, giving for $t=\alpha$ the reduction:

$$g(f(t,\phi_1)) \Rightarrow_{R_1} g(\gamma(\phi_1)) \Rightarrow_{R_2} g'(\phi_1).$$

Note that the context parameter ϕ_1 of f appears in the intermediate result obtained by reducing the inner call, and that during the further computation on this intermediate result a call on ϕ_1 occurs. Such behaviour cannot be modelled by an mtt $M_{1;2}$ if we simply want to replace the call $g(f(t,\phi_1))$ by $(f,g)(t,\phi_1)$, because for doing so, the rule of (f,g) at α would have to be constructed as

$$(f,g)(\alpha,y_1) \to g'(y_1)\,,$$

which contains a call of a state on a *context* variable and thus is inadmissible. $\diamond$

In the construction for $MAC;TOP \subseteq MAC$ from Theorem 4.12 in (Engelfriet & Vogler, 1985) this problem is solved by holding available for every context parameter of f its μ translations for all the states $g_1,\ldots,g_\mu$ of M_2. Accordingly,

$$g(f(t,\phi_1,\ldots,\phi_r))$$

is replaced by the call

$$(f,g)(t,g_1(\phi_1),\ldots,g_\mu(\phi_r))\,,$$

which will be modelled by a generalised rewrite system *Pair* below.

During the construction of the rule for a state (f',g') of $M_{1;2}$ (with $f' \in F^{(r'+1)}$) at an input symbol $\sigma \in \Sigma^{(p)}$, i.e., during the reduction of the right-hand side of

$$(f',g')(\sigma(u_1,\ldots,u_p),y_{1,g_1},\ldots,y_{r',g_\mu}) \to g'(rhs_{f',\sigma})\,,$$

a new case can occur, because $rhs_{f',\sigma}$ can additionally to symbols from Δ and calls of states from F also contain the context variables $y_1,\ldots,y_{r'}$. Recalling that every call of (f',g') is provided—in the $y_{1,g_1},\ldots,y_{r',g_\mu}$-positions—with precomputed translations of the context parameters of f' with all possible states of G, we just have to select the correct one, modelled by a rewrite system *Pre* as given in the following:

R_2 :	$g(\delta(\ldots))$	$\to rhs_{g,\delta}\,,$	$\forall g \in G, \delta \in \Delta$
Pre :	$g(y_k)$	$\to y_{k,g}\,,$	$\forall g \in G, y_k \in Y$
Pair :	$g(f(u,y'_1,\ldots,y'_r))$	$\to (f,g)(u,g_1(y'_1),\ldots,g_\mu(y'_r))\,,$	$\forall g \in G, f \in F^{(r+1)}$

Here y_k and $y_{k,g}$ are treated as nullary symbols, whereas the $u,y'_1,\ldots,y'_r$ are variables.

Note that in the rules of *Pair* the construction "reaches" intermediate data structures in accumulating (context) parameters of the state f, as opposed to classical deforestation.

4.4 Adding Accumulating Parameters also to the Consumer — Reaching Accumulating Parameters with Non-unary States

Our aim is to compose two mtts, none of which is a tdtt. Hence, we have to consider nested calls of the form $g(f(t, \phi_1, \ldots, \phi_r), \eta_1, \ldots, \eta_s)$. Following the discussion from the previous subsections, such a call will be replaced by a call of the paired state (f, g) on t, provided with translations of the context parameters $\phi_1, \ldots, \phi_r$ of f with the μ states of M_2. Additionally, (f, g) must clearly keep the context parameters of the outer call of $g \in G^{(s+1)}$, i.e., (f, g) will altogether have $r \cdot \mu + s$ context parameters.

The rule for a state (f', g') (with $f' \in F^{(r'+1)}$ and $g' \in G^{(s'+1)}$) at some input symbol $\sigma \in \Sigma^{(p)}$ will again be constructed by translating $rhs_{f',\sigma}$ with g', i.e., by reducing the right-hand side of:

$$(f', g')(\sigma(u_1, \ldots, u_p), y_{1,g_1}, \ldots, y_{r',g_\mu}, z_1, \ldots, z_{s'}) \to g'(rhs_{f',\sigma}, z_1, \ldots, z_{s'}).$$

During this reduction, we will again apply rules of R_2 in order to eliminate intermediate data structures, rules of an adapted rewrite system *Pre* to select the appropriate translations of f''s context parameters with states of M_2, and rules of *Pair* to perform a pairing of states as suggested above:

R_2 : $g(\delta(\ldots), z_1, \ldots, z_s)$	$\to rhs_{g,\delta}$,	$\forall g \in G^{(s+1)}, \delta \in \Delta$
Pre : $g(y_k, z_1, \ldots, z_s)$	$\to y_{k,g}$,	$\forall g \in G^{(s+1)}, y_k \in Y$
Pair : $g(f(u, y'_1, \ldots, y'_r), z_1, \ldots, z_s) \to$ $(f, g)(u, g_1(y'_1, \ldots), \ldots, g_\mu(y'_r, \ldots), z_1, \ldots, z_s)$,		$\forall g \in G^{(s+1)}, f \in F^{(r+1)}$

Note that *Pair* is only partially specified here, as the context parameters of the $g_1, \ldots, g_\mu$-calls on $y'_1, \ldots, y'_r$ are not yet determined.

Also, note that the variables $z_1, \ldots, z_s$ are discarded in the rules of *Pre*. The reason is our assumption that the $y_{k,g}$-parameter of (f', g') already contains the correct g-translation of the y_k-parameter of f'. For two unrestricted mtts, this assumption is not feasible, because it might happen that not all calls of g on the kth context parameter of f' have the same parameters. In the next two subsections we will discuss this problem and restrict the mtts M_1 and M_2 in such a way that the above idea of precomputing the translations of context parameters will work.

In order to specify the exact shape of the rules in *Pair*, we will step by step give approximations (in the following marked with (*), (**) and (***)) of the effect of *Pair* on a concrete situation

$$g(f(u_i, \phi_1, \ldots, \phi_r), \eta_1, \ldots, \eta_s)$$

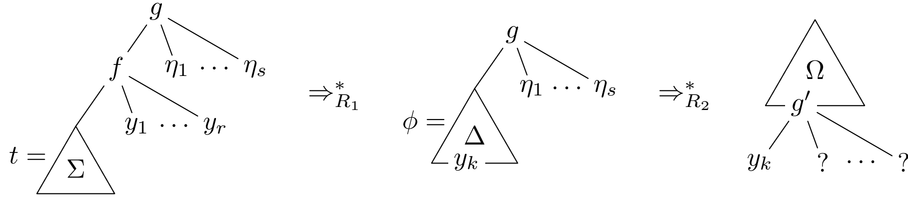

Fig. 2. Can we uniquely determine the values in question mark positions?

with subtrees $\phi_1, \ldots, \phi_r$ of $rhs_{f',\sigma}$ and $\eta_1, \ldots, \eta_s$ being parts of right-hand side expressions for $M_{1;2}$ that have been built up during the translation of $rhs_{f',\sigma}$ with g'.

We already know that we should replace such a nested call with a call of (f, g) on u_i, taking as additional arguments the correct translations of the context parameters of f with all possible states of M_2, and the context parameters of g:

$$\textbf{(*)}\ (f, g)(u_i, g_1(\phi_1, ?, \ldots, ?), \ldots, g_\mu(\phi_r, ?, \ldots, ?), \eta_1, \ldots, \eta_s).$$

4.5 Towards Sufficient Conditions for the Composition Construction

What are we supposed to provide in the places of the question marks in **(*)**? Clearly, it should be the context parameters with which the states $g_1, \ldots, g_\mu$ are expected to "arrive" at occurrences of $\phi_1, \ldots, \phi_r$ during computation of the nested call $g(f(u_i, \phi_1, \ldots, \phi_r), \eta_1, \ldots, \eta_s)$ considered above (and thus will clearly depend on the tree substituted for u_i).

However, it is far from obvious, whether we can always provide this information. So, we better first answer the following question in general (see Fig. 2).

Q : "Given two states $f \in F^{(r+1)}$ and $g \in G^{(s+1)}$ and some input tree t for M_1, can we for every state g' of M_2 and every context variable y_k from $y_1, \ldots, y_r$ uniquely determine, what will be the context parameters in every occurrence of a call of g' on y_k during the reduction of $g(f(t, y_1, \ldots, y_r), \eta_1, \ldots, \eta_s)$?"

To further illustrate the problem, we consider some—rather artificial—examples.

Example 4.2 (*positive answer to question* **Q**)
Let $\Sigma_{mon} = \{A^{(1)}, B^{(1)}, E^{(0)}\}$, $\Delta_{mon} = \{\alpha^{(1)}, \beta^{(1)}, \epsilon^{(0)}\}$ and consider the mtt $M_1 = (\{f_1^{(2)}\}, \Sigma_{mon}, \Delta_{mon}, f_1(x, \epsilon), R_1)$ with set of rules R_1:

$$\begin{array}{rcll}
\text{(i)} & : & f_1(A(u_1), y_1) & \rightarrow\ \alpha(f_1(u_1, y_1)) \\
\text{(ii)} & : & f_1(B(u_1), y_1) & \rightarrow\ f_1(u_1, \beta(y_1)) \\
\text{(iii)} & : & f_1(E, y_1) & \rightarrow\ y_1.
\end{array}$$

Let $\Omega_{tree} = \{\omega^{(2)}, \gamma^{(1)}, \kappa^{(0)}\}$ and $M_2 = (\{g_1^{(2)}, g_2^{(2)}\}, \Delta_{mon}, \Omega_{tree}, g_2(x, \kappa), R_2)$ with set of rules R_2:

$$\begin{array}{rcll}
\text{(iv)} & : & g_1(\alpha(v_1), z_1) & \to g_1(v_1, g_2(v_1, z_1)) \\
\text{(v)} & : & g_1(\beta(v_1), z_1) & \to \gamma(z_1) \\
\text{(vi)} & : & g_1(\epsilon, z_1) & \to z_1 \\
\text{(vii)} & : & g_2(\alpha(v_1), z_1) & \to z_1 \\
\text{(viii)} & : & g_2(\beta(v_1), z_1) & \to g_2(v_1, \omega(g_1(v_1, z_1), z_1)) \\
\text{(ix)} & : & g_2(\epsilon, z_1) & \to z_1.
\end{array}$$

Now, we might ask what will be the context parameters in every occurrence of a call of g_2 on y_1 during the reduction of $g_1(f_1(A(B(E)), y_1), z_1)$:

$$g_1(f_1(A(B(E)), y_1), z_1) \Rightarrow^*_{R_1} g_1(\underbrace{\alpha(\beta(y_1))}_{\phi \text{ of Fig. 2}}, z_1) \Rightarrow^*_{R_2} g_1(\beta(y_1), \underline{g_2(y_1, \underline{\omega(\underline{g_1(y_1, z_1)}, z_1)})})$$

Here the answer is unique, because no other calls of g_2 on y_1 will occur during the further possible reduction. $\diamond$

Example 4.3 (*negative answer to question* **Q**)

(a). Consider rules $f(\epsilon, y_1) \to \delta(y_1, y_1)$ and $g(\delta(v_1, v_2)) \to \omega(g'(v_1, \epsilon), g'(v_2, \kappa))$ of M_1 and M_2, respectively, giving for $t = \epsilon$ the reduction:

$$g(f(\epsilon, y_1)) \Rightarrow_{R_1} g(\delta(y_1, y_1)) \Rightarrow_{R_2} \omega(g'(y_1, \epsilon), g'(y_1, \kappa)).$$

(b). Consider rules $f(\epsilon, y_1) \to \gamma(y_1)$ and $g(\gamma(v_1)) \to g'(v_1, g'(v_1, \epsilon))$ of M_1 and M_2, respectively, giving for $t = \epsilon$ the reduction:

$$g(f(\epsilon, y_1)) \Rightarrow_{R_1} g(\gamma(y_1)) \Rightarrow_{R_2} g'(y_1, g'(y_1, \epsilon)).$$

In both cases the reduction leads to occurrences of calls of g' on y_1 with different context parameters. $\diamond$

Although Example 4.3 shows that in general the answer to question **Q** is *no*, we claim that it can be answered positively if M_1 is non-copying and M_2 is weakly single-use (which excludes the above "counterexamples" (a) and (b))—or, trivially, if one of them is a tdtt—by reasoning as in the following subsection.

4.6 Walking Upwards in Intermediate Results Containing Parameters — The par-Functions

If M_1 is non-copying, then for every tree $t \in T_\Sigma$ the normal form ϕ of $f(t, y_1, \ldots, y_r)$ contains at most one occurrence of every $y_1, \ldots, y_r$. Assume that y_k occurs at path π_{y_k} in ϕ. If, moreover, M_2 is weakly single-use, then we can find out the lth context parameter in every occurrence of a call of state g' on y_k during the reduction of $g(\phi, \eta_1, \ldots, \eta_s)$ by "walking upwards from π_{y_k} in ϕ" as described by a function par_ϕ that takes as arguments a path in ϕ, a state of M_2 and a context parameter position of this state (the initial call hence being $par_\phi(\pi_{y_k}, g', l)$):

(1) If the given path is ε, then either $g' = g$ and $par_\phi(\varepsilon, g', l)$ should deliver the lth context parameter of $g' = g$ at the root of ϕ, i.e. η_l, or $g' \neq g$ and we can take some arbitrary dummy context parameter, because no call of g' will reach the root of ϕ during reduction of $g(\phi, \eta_1, \ldots, \eta_s)$.

(2) If the given path is $\pi j \in paths(\phi)$ and $lab(\phi,\pi) = \delta \in \Delta^{(q)}$, then there is at most one possible way, how the path πj can be reached by a call of state g', namely by the unique (since M_2 is weakly single-use) occurrence of a $g'(v_j,\ldots)$-call in some right-hand side of a δ-rule of M_2.
If there is *no* such call, then we can safely take some dummy symbol for $par_\phi(\pi j, g', l)$, because then the path πj cannot be reached by a call of state g'. If however, there *is* such a call in the right-hand side of a rule

$$g''(\delta(v_1,\ldots,v_q), z_1,\ldots,z_{s''}) \to \ldots g'(v_j,\psi_1,\ldots,\psi_{s'})\ldots ,$$

then our sought lth context parameter of calls of g' reaching the path πj is essentially ψ_l, except that it might contain references to the children of the occurrence of δ (variables $v_1,\ldots,v_q$) and to context parameters of calls of g'' on reaching the occurrence of δ (variables $z_1,\ldots,z_{s''}$). The former are obtained as the direct subtrees of the occurrence of δ in ϕ, while the latter can be computed by using the par_ϕ-function on the path π as in the following:

$$\boxed{\begin{aligned} par_\phi(\pi j, g', l) = \psi_l[v_1,\ldots,v_q &\leftarrow sub(\phi,\pi 1),\ldots,sub(\phi,\pi q),\\ z_1,\ldots,z_{s''} &\leftarrow par_\phi(\pi,g'',1),\ldots,par_\phi(\pi,g'',s'')].\end{aligned}}$$

The occurrence of δ can now again be either at the root of ϕ, i.e. $\pi = \varepsilon$, or in the argument of some output symbol of M_1 in ϕ, so we either are in case (1) or again "walk upwards" in ϕ using this second case, until finally we reach the root.

Example 4.4 (*walking upwards*)
Recall the mtts M_1 and M_2 from Example 4.2 (which are non-copying and weakly single-use, respectively), and assume that we are interested in the first context parameter in a call of g_2 on the occurrence of y_1 in $\phi = \beta(y_1)$, that is, we want to compute $par_{\beta(y_1)}(1, g_2, 1)$. We know that the only way to have a call of g_2 on the occurrence of y_1 at path $\pi j = 1$ (i.e., $\pi = \varepsilon$ and $j = 1$) is via a call of g_2 on v_1 in the right-hand side of some β-rule of M_2. The only such call occurs in the rule (viii):

$$g_2(\beta(v_1), z_1) \to \underline{g_2(v_1, \omega(g_1(v_1,z_1),z_1))}.$$

Hence, the first context parameter of g_2 on y_1 can be calculated by:

$$\begin{aligned} par_{\beta(y_1)}(1,g_2,1) &= \omega(g_1(v_1,z_1),z_1)[v_1 \leftarrow sub(\beta(y_1),1),\\ &\qquad\qquad\qquad\qquad\quad z_1 \leftarrow par_{\beta(y_1)}(\varepsilon,g_2,1)]\\ &= \omega(g_1(y_1, par_{\beta(y_1)}(\varepsilon,g_2,1)), par_{\beta(y_1)}(\varepsilon,g_2,1)). \quad \diamond \end{aligned}$$

The informal explanation above Example 4.4 suggests that question **Q** can be answered with *yes*. However, this does not immediately help our (static) construction of $M_{1;2}$, because an mtt has to work locally and cannot follow our proposed (dynamic) procedure of "reduce $f(t,y_1,\ldots,y_r)$ to its normal form ϕ, check where the context variables $y_1,\ldots,y_r$ are in ϕ, and walk your way up from there".

But, we claim that the above idea can in fact be implemented by introducing new states for the mtt $M_{1;2}$ as explained in the next subsection.

4.7 Introducing New States for Computing Context Parameters

For every two states f and g' of M_1 and M_2, and every pair of context parameter positions k and l of f and g', respectively, we introduce a new state $(k_f, l_{g'})$ [5] of $M_{1;2}$. For every tree t, such a state $(k_f, l_{g'})$ shall compute the lth context parameter in a call $g'(y_k, \ldots)$ resulting from a reduction of $g(f(t, \ldots, y_k, \ldots), \ldots)$ for some g, where g is unique (due to our reasoning of "walking upwards in the intermediate tree, using the weakly single-use property").

Coming back to **(*)** in Subsection 4.4, we can then fill the question marks by using these new states:

$$\textbf{(**)}\quad (f,g)(u_i, g_1(\phi_1, \underline{(1_f, 1_{g_1})(u_i, ?, \ldots, ?), \ldots, (1_f, s_{1_{g_1}})(u_i, ?, \ldots, ?)}), \ldots, g_\mu(\phi_r, \underline{(r_f, 1_{g_\mu})(u_i, ?, \ldots, ?), \ldots, (r_f, s_{\mu_{g_\mu}})(u_i, ?, \ldots, ?)}), \eta_1, \ldots, \eta_s),$$

assuming that the states $g_1, \ldots, g_\mu$ of M_2 have $s_1, \ldots, s_\mu$ context parameters.

But, this produced new question marks, and before we can fill them we have to consider what context parameters such a new state $(k_f, l_{g'})$ requires, respectively, how its rules can be constructed[6]. For every input symbol $\sigma \in \Sigma^{(p)}$ of M_1, we have to construct a rule

$$(k_f, l_{g'})(\sigma(u_1, \ldots, u_p), \ldots) \to ?\,,$$

aimed at computing the lth context parameter in a call of g' on y_k, resulting from a reduction of

$$g(f(\sigma(u_1, \ldots, u_p), \ldots, y_k, \ldots), \ldots)$$

for some (unique) g. Clearly, we should use our knowledge of M_1's rule

$$f(\sigma(u_1, \ldots, u_p), \ldots, y_k, \ldots) \to rhs_{f,\sigma}.$$

If $rhs_{f,\sigma}$ does *not* contain y_k, then there cannot result a call of g' on y_k from the reduction, so we are safe to choose some dummy right-hand side for $(k_f, l_{g'})$ at σ. In the case that $rhs_{f,\sigma}$ *does* contain y_k, we will reuse the idea of "walking upwards from the unique occurrence of y_k", namely with a $par_{rhs_{f,\sigma}}$-function. However, since besides context variables and output symbols, $rhs_{f,\sigma}$ can also contain recursive function calls—which were not present in the discussion in the previous subsection—an appropriate extension to the $par_{rhs_{f,\sigma}}$-function will be necessary. We will discuss this extension in the next subsection. In the absence of function calls from $rhs_{f,\sigma}$, we can use the *par*-functions as introduced so far.

Example 4.5 (*walking upwards in a right-hand side without recursive calls*)
Consider the mtts M_1 and M_2 from Example 4.2, but assume the simpler right-hand side $rhs_{f_1,B} = \beta(y_1)$ for rule (ii). If we want to construct the right-hand side

[5] This shall not denote indexed integers, rather it is used as a compacter notation for the pair of pairs $((k, f), (l, g'))$.
[6] In addition to the $\mu \cdot |R_1|$ rules for (f, g)-states, $R_{1;2}$ will contain $\leq \mu \cdot |R_1| \cdot r_{max} \cdot s_{max}$ rules for $(k_f, l_{g'})$-states, where $r_{max} = max(rank(F)) - 1$ and $s_{max} = max(rank(G)) - 1$.

for $(1_{f_1}, 1_{g_2})$ at B (that is, we are interested in the first context parameter in a call of g_2 on the occurrence of y_1 in $rhs_{f_1,B}$), we obtain according to Example 4.4:

$$par_{rhs_{f_1,B}}(1, g_2, 1) = \omega(g_1(y_1, par_{rhs_{f_1,B}}(\varepsilon, g_2, 1)), par_{rhs_{f_1,B}}(\varepsilon, g_2, 1))\,,$$

which depends both on the g_1-translation of y_1 and on the first context parameter of g_2 at the root of $rhs_{f_1,B}$. $\diamond$

Since in general we do not know, which states—in the previous example g_1 and g_2 in $\omega(\underline{g_1}(y_1, par_{rhs_{f_1,B}}(\varepsilon, \underline{g_2}, 1)), par_{rhs_{f_1,B}}(\varepsilon, \underline{g_2}, 1))$—will be concerned, we might need the information

1. all g-translations of all context parameters of f for every state g of M_2
2. all context parameters for every state g of M_2 at the root of $rhs_{f,\sigma}$ (respectively, at the occurrence of $f(\sigma(u_1, \ldots, u_p), \ldots, y_k, \ldots))$

to compute $(k_f, l_{g'})$ at σ.

Thus, the rule discussed above Example 4.5 will indeed have the form:

$$(k_f, l_{g'})(\sigma(u_1, \ldots, u_p), y_{1,g_1}, \ldots, y_{r,g_\mu}, z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}) \to ?\,,$$

and the definition of *par*-functions on the empty path ε can be made more precise by stating for every $g' \in G^{(s'+1)}$ and $l \in [s']$:

$$\boxed{par_{rhs_{f,\sigma}}(\varepsilon, g', l) = z_{g',l}.}$$

Example 4.6 (*completing the rule for* $(1_{f_1}, 1_{g_2})$ *at* B*, assuming* $rhs_{f_1,B} = \beta(y_1)$)
Continuing the discussion from Example 4.5 we obtain:

$$\begin{aligned} par_{rhs_{f_1,B}}(1, g_2, 1) &= \omega(g_1(y_1, par_{rhs_{f_1,B}}(\varepsilon, g_2, 1)), par_{rhs_{f_1,B}}(\varepsilon, g_2, 1)) \\ &= \omega(g_1(y_1, z_{g_2,1}), z_{g_2,1}). \end{aligned}$$

Since $(1_{f_1}, 1_{g_2})$ has the translations of y_1 with all states of M_2 as context parameters, we can use the rewrite system *Pre* to replace $g_1(y_1, z_{g_2,1})$ with y_{1,g_1}, and obtain the following rule:

$$(1_{f_1}, 1_{g_2})(B(u_1), y_{1,g_1}, y_{1,g_2}, z_{g_1,1}, z_{g_2,1}) \to \omega(y_{1,g_1}, z_{g_2,1}). \quad \diamond$$

Note that—as follows from the reasoning in Subsection 4.6—for every input tree t, state g' and context variable y_k, there will be at most one state $g \in G^{(s+1)}$ such that reducing $g(f(t, \ldots, y_k, \ldots), \ldots)$ leads to a $g'(y_k, \ldots)$-call. The result of reducing $(k_f, l_{g'})(t, y_{1,g_1}, \ldots, y_{r,g_\mu}, z_{g_1,1}, \ldots, z_{g_\mu,s_\mu})$, which shall compute the lth context parameter of this call of g' on y_k, will then contain only those variables of the $z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}$ that are associated with this unique g. However, since for different input trees t also this g might differ, it is unavoidable for the state $(k_f, l_{g'})$ to have all the context variables $z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}$. Intuitively, in order to get the lth context parameter of a call $g'(y_k, \ldots)$ resulting from reduction of $g(f(t, \ldots, y_k, \ldots), \eta_1, \ldots, \eta_s)$, we have to compute

$$(k_f, l_{g'})(t, y_{1,g_1}, \ldots, y_{r,g_\mu}, z_{g_1,1}, \ldots, z_{g_\mu,s_\mu})$$

and replace the $z_{g,1}, \ldots, z_{g,s}$ (no other variables will occur from $z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}$ in the result) by the $\eta_1, \ldots, \eta_s$.

4.8 Walking Upwards in Right-Hand Sides of M_1 *— Extending the* par*-Functions*

The following example demonstrates the necessity to extend our par_ϕ-functions, and prepares the ground for this extension in general.

Example 4.7 (*walking upwards from a parameter position of a recursive call*)
In Examples 4.5 and 4.6 we considered the construction of the right-hand side for $(1_{f_1}, 1_{g_2})$ at B for the mtts M_1 and M_2 from Example 4.2, but used a simplified right-hand side $rhs_{f_1,B}$, where no function call occurred. We will now resume this discussion for the original $rhs_{f_1,B} = f_1(u_1, \beta(y_1))$.

Recall that in order to build the rule

$$(1_{f_1}, 1_{g_2})(B(u_1), y_{1,g_1}, y_{1,g_2}, z_{g_1,1}, z_{g_2,1}) \to ?\,,$$

i.e., to find the first context parameter in a call of g_2 reaching y_1 in $rhs_{f_1,B}$, we have to start from the unique occurrence of y_1 in this right-hand side, which gives us in analogy to Example 4.5:

$$par_{f_1(u_1,\beta(y_1))}(21, g_2, 1) = \omega(g_1(y_1, par_{f_1(u_1,\beta(y_1))}(2, g_2, 1)), par_{f_1(u_1,\beta(y_1))}(2, g_2, 1)).$$

In analogy to Example 4.6, we can later use the rewrite system *Pre* to replace $g_1(y_1, \ldots)$ with y_{1,g_1}. For calculating $par_{f_1(u_1,\beta(y_1))}(2, g_2, 1)$, however, we need further discussion, because it stands for the context parameter of g_2 on reaching the root of a context parameter of f_1, a situation we did not consider so far. But, we have designed the state $(1_{f_1}, 1_{g_2})$ to compute for instantiated u_1 the first context parameter of g_2 on reaching the first context parameter of f_1 during a reduction on $f_1(u_1, \ldots)$ with some state of M_2. Hence, we can obtain our needed value for $par_{f_1(u_1,\beta(y_1))}(2, g_2, 1)$ by calling this state $(1_{f_1}, 1_{g_2})$ on u_1, yielding:

$$(1_{f_1}, 1_{g_2})(B(u_1), y_{1,g_1}, y_{1,g_2}, z_{g_1,1}, z_{g_2,1}) \to \omega(y_{1,g_1}, (1_{f_1}, 1_{g_2})(u_1, ?, \ldots, ?)).$$

How to fill the context parameter positions of this call will now be discussed in general. $\diamond$

We abstract from the previous example and explain the working of the par_ϕ-function for a path πj that corresponds to the $(j-1)$st context parameter position of a call of some $f \in F$ on some $u_i \in U$ in ϕ, i.e., $lab(\phi, \pi) = f$ and $lab(\phi, \pi 1) = u_i$. Recall that $par_\phi(\pi j, g', l)$ shall—modulo reductions with $R_2 \cup Pre \cup Pair$—give a right-hand side expression for $M_{1;2}$ that computes the lth context parameter in a call of state g' reaching the path πj in M_1's right-hand side ϕ. Since the path πj is the $(j-1)$st context parameter position of a call of f, this can be realised by a call of $((j-1)_f, l_{g'})$ on u_i, because this state was *designed* to compute the lth context parameter in a call of state g' reaching the $(j-1)$st context parameter position of a call of state f on a given input. In order to do so, the state $((j-1)_f, l_{g'})$ has to be provided with translations of f's context parameters—at the particular occurrence of the call of f at path π in ϕ—with states of M_2, and furthermore, with the context parameters of M_2's states at this occurrence. The latter ones can be computed by

par_ϕ-function calls on the path of the occurrence of f in ϕ. Hence,

$$\boxed{par_\phi(\pi j, g', l) = ((j-1)_f, l_{g'})(u_i, \underbrace{\ldots}_{nest_f}, par_\phi(\pi, g_1, 1), \ldots, par_\phi(\pi, g_\mu, s_\mu))\,,}$$

still missing the translations of f's context parameters (the subtrees of ϕ in context parameter positions of the call of f) with states of M_2. Actually a (finite!) nesting of such translations will be necessary, to be discussed at the end of Subsection 4.10.

4.9 Towards the $Pair$*-Rules*

Let us now further consider the effect that the rewrite system *Pair* should have on $g(f(u_i, \phi_1, \ldots, \phi_r), \eta_1, \ldots, \eta_s)$. In particular, we need to fill the question mark slots of **(**)** in Subsection 4.7.

Recall that we introduced the calls of $(1_f, 1_{g_1}), \ldots, (r_f, s_{\mu_{g_\mu}})$ on u_i in order to compute the context parameters with which the states $g_1, \ldots, g_\mu$ are expected to "arrive" at $\phi_1, \ldots, \phi_r$ during the reduction of $g(f(u_i, \phi_1, \ldots, \phi_r), \eta_1, \ldots, \eta_s)$ for instantiated u_i. Further, recall that—by the discussion below Example 4.5—every state $(k_f, l_{g'}) \in \{(1_f, 1_{g_1}), \ldots, (r_f, s_{\mu_{g_\mu}})\}$ expects in the $z_{g_1,1}, \ldots, z_{g_\mu, s_\mu}$-positions of its call on u_i the context parameters of all possible states of M_2 at the occurrence of $f(u_i, \phi_1, \ldots, \phi_r)$.

However, only those of the translations $g_1(\phi_1, \ldots), \ldots, g_\mu(\phi_r, \ldots)$ will be needed during the computation of **(**)** that actually occur during the reduction of

$$\underline{g}(f(u_i, \phi_1, \ldots, \phi_r), \eta_1, \ldots, \eta_s)$$

for instantiated u_i. For such a g' and ϕ_k we know by the discussion at the end of Subsection 4.7, that a call of state $(k_f, l_{g'})$ on the concrete tree substituted for u_i will depend only on those of its $z_{g_1,1}, \ldots, z_{g_\mu, s_\mu}$-positions that are associated with the $\underline{g}$ fixed above. We also gave the intuition that for these $z_{g,1}, \ldots, z_{g,s}$-positions, we have to provide the $\eta_1, \ldots, \eta_s$. The other $z_{g_1,1}, \ldots, z_{g_\mu, s_\mu}$-positions—those not associated with $\underline{g}$—are filled with a dummy nullary output symbol nil [7]. This leaves us with the following expression:

$$\begin{aligned}
\textbf{(***)}\ (f,g)(u_i, &g_1(\phi_1, (1_f, 1_{g_1})(u_i, ?, \ldots, ?, \underline{nil, \ldots, nil, \eta_1, \ldots, \eta_s, nil, \ldots, nil}),\\
&\qquad\ldots,\\
&\qquad(1_f, s_{1_{g_1}})(u_i, ?, \ldots, ?, \underline{nil, \ldots, nil, \eta_1, \ldots, \eta_s, nil, \ldots, nil})),\\
&\ldots,\\
&g_\mu(\phi_r, (r_f, 1_{g_\mu})(u_i, ?, \ldots, ?, \underline{nil, \ldots, nil, \eta_1, \ldots, \eta_s, nil, \ldots, nil}),\\
&\qquad\ldots,\\
&\qquad(r_f, s_{\mu_{g_\mu}})(u_i, ?, \ldots, ?, \underline{nil, \ldots, nil, \eta_1, \ldots, \eta_s, nil, \ldots, nil})),\\
&\eta_1, \ldots, \eta_s)\,,
\end{aligned}$$

still having to fill the $y_{1,g_1}, \ldots, y_{r,g_\mu}$-positions of calls of $(1_f, 1_{g_1}), \ldots, (r_f, s_{\mu_{g_\mu}})$. Again, this will lead to nested translations, as discussed in the next subsection.

[7] In Subsection 5.3 we will demonstrate that a seemingly simpler approach—constructing states with reduced ranks instead—is not feasible.

4.10 Nesting Translations and Cutting Cycles

For the question mark positions in **(***)** we have to provide the $g_1, \ldots, g_\mu$-translations of the $\phi_1, \ldots, \phi_r$, which might be needed during the computations of the states $(1_f, 1_{g_1}), \ldots, (r_f, s_{\mu_{g_\mu}})$ on instantiated u_i. This looks suspiciously like a circular construction, because we would use $g_1(\phi_1, \ldots)$ within $g_1(\phi_1, \ldots)$ and the like. However, we can "cut" these cycles by keeping track of the nesting (similar to the unfolding in the construction for $ATT \subseteq MAC$, cf. Franchi-Zannettacci (1982) and Fülöp & Vogler (1999)), because the positions where such cyclic dependencies would occur, can for no possible input tree influence the computation. The intuitive reason for this is that otherwise we would meet a situation corresponding to the "counterexample" (b) in Example 4.3, like $g'(y_1, g'(y_1, \epsilon))$, which *cannot* occur, because we argued that all occurrences of $g'(y_1, \ldots)$ will have identical context parameters, provided that M_1 is non-copying and M_2 is weakly single-use. The formal reason is established in Lemma A.8 of the Appendix (Voigtländer & Kühnemann, 2003).

The nested translations of the context parameters $\phi_1, \ldots, \phi_r$ of f with the states $g_1, \ldots, g_\mu$ of M_2 will be built up with the help of a $nest_f$-function that takes as arguments the position $k \in [r]$ of the context parameter to be translated, a state $g' \in G^{(s'+1)}$ with which to translate, and a set $\mathcal{C} \subseteq [r] \times G$ that keeps track of nested parameter-state-combinations, in order to perform the "cutting" of cycles during the nesting process as mentioned above. Hence, if $(k, g') \in \mathcal{C}$, then $nest_f(k, g', \mathcal{C})$ will return the dummy symbol nil, otherwise a call of the form

$$\boxed{g'(y'_k, (k_f, 1_{g'})(u, \underbrace{\ldots}_{nest_f}, z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}), \ldots, (k_f, s'_{g'})(u, \underbrace{\ldots}_{nest_f}, z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}))\,,}$$

where the $y_{1,g_1}, \ldots, y_{r,g_\mu}$-positions of the $(k_f, 1_{g'}), \ldots, (k_f, s'_{g'})$-calls contain recursive applications of $nest_f$ with the enlarged set of "cut-positions" $\mathcal{C} \cup \{(k, g')\}$.

Abstracting from the concrete u_i, $\phi_1, \ldots, \phi_r$ and $\eta_1, \ldots, \eta_s$ in **(***)**, such $nest_f$-functions will be used in Construction 5.1 to determine the rules for the rewrite system *Pair* from Subsection 4.4.

Concluding the discussion of the "walking upwards" in a right-hand side ϕ of M_1 for the case that a call of state f is encountered (cf. Subsection 4.8), the $nest_f$-function will also take care of preparing the $g_1, \ldots, g_\mu$-translations of the context parameters of such a call, necessary to obtain $par_\phi(\pi j, g', l)$. Since the lth context parameter in a call of state g' on reaching the $(j-1)$st context parameter position of a call of state f cannot depend on the g'-translation of this $(j-1)$st context parameter (cf. the "cutting" discussion), this nesting will not be started with the empty set of "cut-positions", but with $\{(j-1, g')\}$.

5 Composition of Restricted Macro Tree Transducers

In this section we present the construction for composing a non-copying and a weakly single-use mtt by formally defining the rewrite systems *Pre* and *Pair*, and the $nest_f$- and par_ϕ-functions discussed in the previous section. For illustration, we

will apply it to the mtts from Example 4.2, and discuss why a seemingly simpler construction cannot work.

5.1 The Complete Construction

The following construction is essentially Construction 3.2 of Voigtländer (2001), but described in a simplified way. It is straightforward—though tiresome and thus omitted here—to establish that it produces a well-defined mtt.

Construction 5.1 (*direct composition of restricted mtts*)
Let $M_1 = (F, \Sigma, \Delta, e_1, R_1)$ and $M_2 = (G, \Delta, \Omega, e_2, R_2)$ be mtts, such that M_1 is non-copying and M_2 is weakly single-use. Assume that M_1 uses recursion variables from U and context variables from Y, whereas M_2 uses V and Z, respectively.

Let $\mu = |G|$ and fix some ordering of the states in G, such that $G = \{g_1, \ldots, g_\mu\}$. For $n \in [\mu]$, let $s_n \in \mathbb{N}$ be such that $g_n \in G^{(s_n+1)}$. Additionally, let $r_{max} = max(rank(F)) - 1$ and let $nil \in \Omega^{(0)}$ be some arbitrary output symbol and $Z_G = \{z_{g_1,1}, \ldots, z_{g_1,s_1}, \ldots, z_{g_\mu,1}, \ldots, z_{g_\mu,s_\mu}\}$.
Then, the components of the mtt $M_{1;2} = (H, \Sigma, \Omega, e_{1;2}, R_{1;2})$ are obtained as follows:

- $H = \{(f,g)^{(r\cdot\mu+s+1)} \mid f \in F^{(r+1)}, g \in G^{(s+1)}\} \cup \{(k_f, l_g)^{(r\cdot\mu+|Z_G|+1)} \mid f \in F^{(r+1)}, g \in G^{(s+1)}, k \in [r], l \in [s]\}$
- $e_{1;2} = nf(\Rightarrow_{R_2 \cup Pair}, e_2[x \leftarrow e_1])$
- $R_{1;2}$ contains:
  — for every $f \in F^{(r+1)}$, $g \in G^{(s+1)}$ and $\sigma \in \Sigma^{(p)}$, the rule:

$$(f,g)(\sigma(u_1, \ldots, u_p), y_{1,g_1}, \ldots, y_{r,g_\mu}, z_1, \ldots, z_s) \rightarrow nf(\Rightarrow_{R_2 \cup Pre \cup Pair}, g(rhs_{f,\sigma}, z_1, \ldots, z_s)),$$

  — for every $f \in F^{(r+1)}$, $g \in G^{(s+1)}$, $k \in [r]$, $l \in [s]$ and $\sigma \in \Sigma^{(p)}$, the rule:

$$(k_f, l_g)(\sigma(u_1, \ldots, u_p), y_{1,g_1}, \ldots, y_{r,g_\mu}, z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}) \rightarrow nf(\Rightarrow_{R_2 \cup Pre \cup Pair}, \varrho),$$

  where $\varrho = nil$, if $rhs_{f,\sigma}$ does not contain the context variable y_k; otherwise $\varrho = par_{rhs_{f,\sigma}}(\pi_{y_k}, g, l)$, where $\pi_{y_k} \in paths(rhs_{f,\sigma})$ is the path of the unique occurrence of y_k in $rhs_{f,\sigma}$ (notice that M_1 is non-copying).

The rewrite systems *Pre* (over $G \cup Y_{r_{max}} \cup \{y^{(0)}_{k,g} \mid k \in [r_{max}], g \in G\}$ and Z) and *Pair* (over $F \cup G \cup H \cup \{nil\}$ and $\{u\} \cup Y' \cup Z$) used above are defined as follows:

Pre : $g(y_k, z_1, \ldots, z_s)$	$\rightarrow y_{k,g}$,	$\forall g \in G^{(s+1)}, k \in [r_{max}]$
Pair : $g(f(u, y'_1, \ldots, y'_r), z_1, \ldots, z_s) \rightarrow (f,g)(u, nest_f(1, g_1, \emptyset), \ldots, nest_f(r, g_\mu, \emptyset), z_1, \ldots, z_s)[z_{g,1}, \ldots, z_{g,s} \leftarrow z_1, \ldots, z_s][z_{g_1,1}, \ldots, z_{g_\mu,s_\mu} \leftarrow nil, \ldots, nil]$,		$\forall g \in G^{(s+1)}, f \in F^{(r+1)}$

There are no *critical pairs* (Dershowitz & Jouannaud, 1990) in $R_2 \cup Pre \cup Pair$,

hence the reduction relation $\Rightarrow_{R_2 \cup Pre \cup Pair}$ is confluent. Since the rewrite rules in $R_2 \cup Pre \cup Pair$ can be interpreted as rules of an auxiliary mtt (which uses state set G to translate right-hand sides of M_1 into right-hand sides of $M_{1;2}$), it is also terminating. Hence, unique normal forms with respect to $\Rightarrow_{R_2 \cup Pre \cup Pair}$ exist.

For every $f \in F^{(r+1)}$, the function

$$nest_f : [r] \times G \times \mathcal{P}([r] \times G) \to T_{G \cup H \cup \{nil\}}(\{u\} \cup Y'_r \cup Z_G)$$

is defined as follows. For every $k \in [r]$, $g' \in G^{(s'+1)}$ and $\mathcal{C} \subseteq [r] \times G$:

- $nest_f(k, g', \mathcal{C}) = nil\,,$
  if $(k, g') \in \mathcal{C}$.
- $nest_f(k, g', \mathcal{C}) =$
  $$\begin{aligned} g'(y'_k, (k_f, 1_{g'})(u,\ & nest_f(1, g_1, \mathcal{C} \cup \{(k, g')\}), \dots, nest_f(r, g_\mu, \mathcal{C} \cup \{(k, g')\}),\\ & z_{g_1,1}, \dots, z_{g_\mu,s_\mu}),\\ \dots,&\\ (k_f, {s'}_{g'})(u,\ & nest_f(1, g_1, \mathcal{C} \cup \{(k, g')\}), \dots, nest_f(r, g_\mu, \mathcal{C} \cup \{(k, g')\}),\\ & z_{g_1,1}, \dots, z_{g_\mu,s_\mu}))\,, \end{aligned}$$
  if $(k, g') \notin \mathcal{C}$.

Using the $nest_f$-functions we additionally define for every $\phi \in RHS(F, \Delta, U, Y)$ the function

$$\begin{aligned} par_\phi : \{(\pi, g', l) \mid \pi \in paths(\phi), lab(\phi, \pi) \notin U, g' \in G^{(s'+1)}, l \in [s']\}\\ \to T_{F \cup G \cup H \cup \Delta \cup \Omega}(U \cup Y \cup Z_G) \end{aligned}$$

by induction on the prefix-order of paths in ϕ as follows. For every $g' \in G^{(s'+1)}$ and $l \in [s']$:

- $par_\phi(\varepsilon, g', l) = z_{g',l}$
- For every $j \in \mathbb{N}_+$, $\pi j \in paths(\phi)$ with $lab(\phi, \pi j) \notin U$, we define $par_\phi(\pi j, g', l)$ by case distinction on $lab(\phi, \pi)$ as follows:

  $\underline{lab(\phi, \pi) = \delta}$ for some $\delta \in \Delta^{(q)}$ and $j \in [q]$:

  If, with $g'' \in G^{(s''+1)}$ and $\psi_1, \dots, \psi_{s'} \in RHS(G, \Omega, V_q, Z_{s''})$, the only occurrence of a $g'(v_j, \dots)$-call in the δ-rules of the weakly single-use mtt M_2 looks as follows:

  $$g''(\delta(v_1, \dots, v_q), z_1, \dots, z_{s''}) \to \dots g'(v_j, \psi_1, \dots, \psi_{s'}) \dots\,,$$

  then:

  $$\begin{aligned} par_\phi(\pi j, g', l) = \psi_l[v_1, \dots, v_q &\leftarrow sub(\phi, \pi 1), \dots, sub(\phi, \pi q),\\ z_1, \dots, z_{s''} &\leftarrow par_\phi(\pi, g'', 1), \dots, par_\phi(\pi, g'', s'')]. \end{aligned}$$

  If no such call exists in the δ-rules of M_2, then $par_\phi(\pi j, g', l) = nil$.

$\underline{lab(\phi,\pi) = f}$ for some $f \in F^{(r+1)}$, $1 \leq j-1 \leq r$ and $lab(\phi,\pi 1) = u_i \in U$:

$$\begin{aligned}
& par_\phi(\pi j, g', l) \\
& = ((j-1)_f, l_{g'})(u, nest_f(1, g_1, \{(j-1, g')\}), \ldots, nest_f(r, g_\mu, \{(j-1, g')\}), \\
& \qquad\qquad\qquad z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}) \\
& \quad [u \leftarrow u_i, \\
& \quad\; y'_1, \ldots, y'_r \leftarrow sub(\phi,\pi 2), \ldots, sub(\phi,\pi(r+1)), \\
& \quad\; z_{g_1,1}, \ldots, z_{g_\mu,s_\mu} \leftarrow par_\phi(\pi, g_1, 1), \ldots, par_\phi(\pi, g_\mu, s_\mu)]. \quad \square
\end{aligned}$$

The following main theorem—stating the correctness of the previous construction—is proven as Theorem A.16 in the Appendix (Voigtländer & Kühnemann, 2003).

Theorem 5.2 (*correctness of Construction 5.1*)
$\tau(M_1); \tau(M_2) = \tau(M_{1;2})$ $\square$

Note that in Construction 5.1 we used only condition (i) of the weakly single-use restriction for M_2 from Definition 3.6 and this only for states of M_2 with rank greater than one (consider that par_ϕ is only defined for triples (π, g', l) where g' is non-unary). In fact, we could generalise the weakly single-use property in Definition 3.6 by dropping condition (ii) and requiring condition (i) only for states g that do have context parameters. This would require no change to Construction 5.1 or to the proofs in (Voigtländer & Kühnemann, 2003)! The informal explanations from Section 4, however, would be more involved and less intuitive for such a generalised class. That is why we kept to the stronger restriction weakly single-use of Kühnemann (1998) and only mention the possible generalisation here.

Further, note that we used the non-copying restriction of M_1 and the weakly single-use restriction of M_2 only in the construction of rules for the (k_f, l_g)-states. Since no such states are created if one of the two original mtts is a tdtt, the composition construction is also applicable if M_1 or M_2 is a tdtt (and the other one is an unrestricted mtt). In these two special cases, Construction 5.1 corresponds to Transformation 11 in (Kühnemann, 1999) (if M_1 is a tdtt), respectively, to the construction in the proof of Theorem 4.12 in (Engelfriet & Vogler, 1985) (if M_2 is a tdtt).

5.2 Calculating an Example

As an example we now present the application of Construction 5.1 to the two mtts from Example 4.2. In order to make explicit the positions where the dummy symbol *nil* is used—indicating that these positions will not influence the computation—we add *nil* to the output ranked alphabet, instead of using some existing nullary symbol as we usually would do.

Example 5.3 (*applying the direct composition construction*)
Consider M_1 and M_2 from Example 4.2. Since M_1 is non-copying and M_2 is weakly single-use, we can apply Construction 5.1 to obtain the mtt $M_{1;2} = (H, \Sigma_{mon}, \Omega_{tree} \cup \{nil^{(0)}\}, e_{1;2}, R_{1;2})$ with components as follows (where $\mu = 2$, $s_1 = s_2 = 1$, $r_{max} = 1$ and $Z_G = \{z_{g_1,1}, z_{g_2,1}\}$):

- $H = \{(f_1, g_1)^{(4)}, (f_1, g_2)^{(4)}\} \cup \{(1_{f_1}, 1_{g_1})^{(5)}, (1_{f_1}, 1_{g_2})^{(5)}\}$
- *Pre* contains the rules: $g_1(y_1, z_1) \to y_{1,g_1}$
  $g_2(y_1, z_1) \to y_{1,g_2}$.
- *Pair* contains the rules:

$$\begin{aligned}
g_1(f_1(u, y_1'), z_1) \to\ & (f_1, g_1)(u, nest_{f_1}(1, g_1, \emptyset), nest_{f_1}(1, g_2, \emptyset), z_1) \\
& [z_{g_1,1}, z_{g_2,1} \leftarrow z_1, nil] \\
= (f_1, g_1)(u,\ & g_1(y_1', (1_{f_1}, 1_{g_1})(u, nil, g_2(y_1', (1_{f_1}, 1_{g_2})(u, nil, nil, z_1, nil)), \\
& \qquad z_1, nil)), \\
& g_2(y_1', (1_{f_1}, 1_{g_2})(u, g_1(y_1', (1_{f_1}, 1_{g_1})(u, nil, nil, z_1, nil)), nil, \\
& \qquad z_1, nil)), \\
& z_1)
\end{aligned}$$

$$\begin{aligned}
g_2(f_1(u, y_1'), z_1) \to\ & (f_1, g_2)(u, nest_{f_1}(1, g_1, \emptyset), nest_{f_1}(1, g_2, \emptyset), z_1) \\
& [z_{g_1,1}, z_{g_2,1} \leftarrow nil, z_1] \\
= (f_1, g_2)(u,\ & g_1(y_1', (1_{f_1}, 1_{g_1})(u, nil, g_2(y_1', (1_{f_1}, 1_{g_2})(u, nil, nil, nil, z_1)), \\
& \qquad nil, z_1)), \\
& g_2(y_1', (1_{f_1}, 1_{g_2})(u, g_1(y_1', (1_{f_1}, 1_{g_1})(u, nil, nil, nil, z_1)), nil, \\
& \qquad nil, z_1)), \\
& z_1).
\end{aligned}$$

- $e_{1;2} = nf(\Rightarrow_{R_2 \cup Pair}, g_2(x, \kappa)[x \leftarrow f_1(x, \epsilon)])$, with

$$\begin{aligned}
& g_2(x, \kappa)[x \leftarrow f_1(x, \epsilon)] = g_2(f_1(x, \epsilon), \kappa) \\
& \Rightarrow_{Pair} (f_1, g_2)(x, g_1(\epsilon, (1_{f_1}, 1_{g_1})(x, nil, g_2(\epsilon, (1_{f_1}, 1_{g_2})(x, nil, nil, nil, \kappa)), \\
& \qquad\qquad nil, \kappa)), \\
& \qquad g_2(\epsilon, (1_{f_1}, 1_{g_2})(x, g_1(\epsilon, (1_{f_1}, 1_{g_1})(x, nil, nil, nil, \kappa)), nil, \\
& \qquad\qquad nil, \kappa)), \\
& \qquad \kappa) \\
& \Rightarrow_{R_2}^{*} (f_1, g_2)(x, (1_{f_1}, 1_{g_1})(x, nil, (1_{f_1}, 1_{g_2})(x, nil, nil, nil, \kappa), nil, \kappa), \\
& \qquad (1_{f_1}, 1_{g_2})(x, (1_{f_1}, 1_{g_1})(x, nil, nil, nil, \kappa), nil, nil, \kappa), \\
& \qquad \kappa).
\end{aligned}$$

- For the rules in $R_{1;2}$, we compute only two examples here:

1. We compute the rule for (f_1, g_1) at A:
   $(f_1, g_1)(A(u_1), y_{1,g_1}, y_{1,g_2}, z_1) \ \to\ nf(\Rightarrow_{R_2 \cup Pre \cup Pair}, g_1(rhs_{f_1,A}, z_1))$, with

$$\begin{aligned}
& g_1(rhs_{f_1,A}, z_1) = g_1(\alpha(f_1(u_1, y_1)), z_1) \\
& \Rightarrow_{R_2} \ g_1(f_1(u_1, y_1), g_2(f_1(u_1, y_1), z_1)) \\
& \Rightarrow_{Pair} (f_1, g_1)(u_1, g_1(y_1, (1_{f_1}, 1_{g_1})(u_1, nil, g_2(y_1, (1_{f_1}, 1_{g_2})(u_1, nil, nil, \\
& \qquad\qquad g_2(f_1(u_1, y_1), z_1), nil)), \\
& \qquad\qquad g_2(f_1(u_1, y_1), z_1), nil)), \\
& \qquad g_2(y_1, (1_{f_1}, 1_{g_2})(u_1, g_1(y_1, (1_{f_1}, 1_{g_1})(u_1, nil, nil, \\
& \qquad\qquad g_2(f_1(u_1, y_1), z_1), nil)), nil, \\
& \qquad\qquad g_2(f_1(u_1, y_1), z_1), nil)), \\
& \qquad g_2(f_1(u_1, y_1), z_1)) \\
& \Rightarrow_{Pre}^{*} \ (f_1, g_1)(u_1, y_{1,g_1}, y_{1,g_2}, g_2(f_1(u_1, y_1), z_1))
\end{aligned}$$

$$
\begin{aligned}
\Rightarrow_{Pair}\ & (f_1,g_1)(u_1,y_{1,g_1},y_{1,g_2},\\
& \quad (f_1,g_2)(u_1,g_1(y_1,(1_{f_1},1_{g_1})(u_1,nil,g_2(y_1,(1_{f_1},1_{g_2})(u_1,\\
& \qquad nil,nil,nil,z_1)),\\
& \qquad nil,z_1)),\\
& \quad g_2(y_1,(1_{f_1},1_{g_2})(u_1,g_1(y_1,(1_{f_1},1_{g_1})(u_1,\\
& \qquad nil,nil,nil,z_1)),nil,\\
& \qquad nil,z_1)),\\
& \quad z_1))\\
\Rightarrow^*_{Pre}\ & (f_1,g_1)(u_1,y_{1,g_1},y_{1,g_2},(f_1,g_2)(u_1,y_{1,g_1},y_{1,g_2},z_1)).
\end{aligned}
$$

2. We compute the rule for $(1_{f_1},1_{g_2})$ at B:

$$
\begin{aligned}
& (1_{f_1},1_{g_2})(B(u_1),y_{1,g_1},y_{1,g_2},z_{g_1,1},z_{g_2,1})\\
& \qquad \to nf(\Rightarrow_{R_2\cup Pre\cup Pair}, par_{rhs_{f_1,B}}(21,g_2,1))\text{ , with}\\
& par_{rhs_{f_1,B}}(21,g_2,1) = par_{f_1(u_1,\beta(y_1))}(21,g_2,1)\\
= \ & \omega(g_1(v_1,z_1),z_1)[v_1,z_1 \leftarrow y_1, par_{f_1(u_1,\beta(y_1))}(2,g_2,1)]\\
= \ & \omega(g_1(v_1,z_1),z_1)[v_1 \leftarrow y_1,\\
& \quad z_1 \leftarrow (1_{f_1},1_{g_2})(u, nest_{f_1}(1,g_1,\{(1,g_2)\}),\\
& \qquad nest_{f_1}(1,g_2,\{(1,g_2)\}),z_{g_1,1},z_{g_2,1})\\
& \qquad [u,y_1' \leftarrow u_1,\beta(y_1),\\
& \qquad z_{g_1,1} \leftarrow par_{f_1(u_1,\beta(y_1))}(\varepsilon,g_1,1),\\
& \qquad z_{g_2,1} \leftarrow par_{f_1(u_1,\beta(y_1))}(\varepsilon,g_2,1)]]\\
= \ & \omega(g_1(v_1,z_1),z_1)[v_1 \leftarrow y_1,\\
& \quad z_1 \leftarrow (1_{f_1},1_{g_2})(u,g_1(y_1',(1_{f_1},1_{g_1})(u,nil,nil,\\
& \qquad z_{g_1,1},z_{g_2,1})),\\
& \qquad nil,z_{g_1,1},z_{g_2,1})\\
& \qquad [u,y_1',z_{g_1,1},z_{g_2,1} \leftarrow u_1,\beta(y_1),z_{g_1,1},z_{g_2,1}]]\\
= \ & \omega(g_1(v_1,z_1),z_1)[v_1 \leftarrow y_1,\\
& \quad z_1 \leftarrow (1_{f_1},1_{g_2})(u_1,g_1(\beta(y_1),(1_{f_1},1_{g_1})(u_1,nil,nil,\\
& \qquad z_{g_1,1},z_{g_2,1})),\\
& \qquad nil,z_{g_1,1},z_{g_2,1})]\\
= \ & \omega(g_1(y_1,(1_{f_1},1_{g_2})(u_1,g_1(\beta(y_1),(1_{f_1},1_{g_1})(u_1,nil,nil,z_{g_1,1},z_{g_2,1})),\\
& \qquad nil,z_{g_1,1},z_{g_2,1})),\\
& \quad (1_{f_1},1_{g_2})(u_1,g_1(\beta(y_1),(1_{f_1},1_{g_1})(u_1,nil,nil,z_{g_1,1},z_{g_2,1})),\\
& \qquad nil,z_{g_1,1},z_{g_2,1}))\\
\Rightarrow_{Pre}\ & \omega(y_{1,g_1},(1_{f_1},1_{g_2})(u_1,g_1(\beta(y_1),(1_{f_1},1_{g_1})(u_1,nil,nil,z_{g_1,1},z_{g_2,1})),\\
& \qquad nil,z_{g_1,1},z_{g_2,1}))\\
\Rightarrow_{R_2}\ & \omega(y_{1,g_1},(1_{f_1},1_{g_2})(u_1,\gamma((1_{f_1},1_{g_1})(u_1,nil,nil,z_{g_1,1},z_{g_2,1})),\\
& \qquad nil,z_{g_1,1},z_{g_2,1})).
\end{aligned}
$$

Note the correspondences between this calculation and the informal explanations in Example 4.7.

Altogether, we get the following set of rules:

$$
\begin{aligned}
(f_1,g_1)(A(u_1),y_{1,g_1},y_{1,g_2},z_1) &\to (f_1,g_1)(u_1,y_{1,g_1},y_{1,g_2},(f_1,g_2)(u_1,y_{1,g_1},y_{1,g_2},z_1))\\
(f_1,g_1)(B(u_1),y_{1,g_1},y_{1,g_2},z_1) &\to (f_1,g_1)(u_1,\gamma((1_{f_1},1_{g_1})(u_1,nil,y_{1,g_2},z_1,nil)),\\
&\qquad y_{1,g_2},z_1)
\end{aligned}
$$

$$
\begin{array}{ll}
(f_1,g_1)(E,y_{1,g_1},y_{1,g_2},z_1) & \to y_{1,g_1}\\
(f_1,g_2)(A(u_1),y_{1,g_1},y_{1,g_2},z_1) & \to z_1\\
(f_1,g_2)(B(u_1),y_{1,g_1},y_{1,g_2},z_1) & \to (f_1,g_2)(u_1,\gamma((1_{f_1},1_{g_1})(u_1,nil,y_{1,g_2},nil,z_1)),\\
& \qquad y_{1,g_2},z_1)\\
(f_1,g_2)(E,y_{1,g_1},y_{1,g_2},z_1) & \to y_{1,g_2}\\
(1_{f_1},1_{g_1})(A(u_1),y_{1,g_1},y_{1,g_2},z_{g_1,1},z_{g_2,1}) & \to (1_{f_1},1_{g_1})(u_1,nil,y_{1,g_2},\\
& \qquad (f_1,g_2)(u_1,y_{1,g_1},y_{1,g_2},z_{g_1,1}),z_{g_1,1})\\
(1_{f_1},1_{g_1})(B(u_1),y_{1,g_1},y_{1,g_2},z_{g_1,1},z_{g_2,1}) & \to (1_{f_1},1_{g_2})(u_1,\\
& \qquad \gamma((1_{f_1},1_{g_1})(u_1,nil,nil,z_{g_1,1},z_{g_2,1})),\\
& \qquad nil,z_{g_1,1},z_{g_2,1})\\
(1_{f_1},1_{g_1})(E,y_{1,g_1},y_{1,g_2},z_{g_1,1},z_{g_2,1}) & \to z_{g_1,1}\\
(1_{f_1},1_{g_2})(A(u_1),y_{1,g_1},y_{1,g_2},z_{g_1,1},z_{g_2,1}) & \to (1_{f_1},1_{g_2})(u_1,y_{1,g_1},nil,\\
& \qquad (f_1,g_2)(u_1,y_{1,g_1},y_{1,g_2},z_{g_1,1}),z_{g_1,1})\\
(1_{f_1},1_{g_2})(B(u_1),y_{1,g_1},y_{1,g_2},z_{g_1,1},z_{g_2,1}) & \to \omega(y_{1,g_1},(1_{f_1},1_{g_2})(u_1,\\
& \qquad \gamma((1_{f_1},1_{g_1})(u_1,nil,nil,z_{g_1,1},z_{g_2,1})),\\
& \qquad nil,z_{g_1,1},z_{g_2,1}))\\
(1_{f_1},1_{g_2})(E,y_{1,g_1},y_{1,g_2},z_{g_1,1},z_{g_2,1}) & \to z_{g_2,1}.
\end{array}
$$

The fact that the positions where *nil* was introduced above do not influence the computation—i.e., that for every possible input, the output tree computed by $M_{1;2}$ will contain no *nil*-symbols—is discussed on pages 22–24 of (Voigtländer & Kühnemann, 2001) by analysing the different types of occurrences of *nil*. ◇

5.3 Exploring Alternatives

Regarding the *nil*-symbols that are introduced by the *Pair*-rules, e.g., into expression **(***)** in Subsection 4.9, the reader may wonder why we do not instead introduce refined states $(g, k_f, l_{g'})$ with reduced ranks, and use *Pair*-rules of the form:

$$
\begin{array}{l}
g(f(u,y'_1,\ldots,y'_r),z_1,\ldots,z_s) \to\\
\quad (f,g)(u,g_1(y'_1,(g,1_f,1_{g_1})(u,nil,\ldots,g_\mu(y'_r,\ldots),\underline{z_1,\ldots,z_s}),\\
\qquad\qquad\qquad \ldots,\\
\qquad\qquad\qquad (g,1_f,s_{1_{g_1}})(u,nil,\ldots,g_\mu(y'_r,\ldots),\underline{z_1,\ldots,z_s})),\\
\qquad\quad \ldots,\\
\qquad\quad g_\mu(y'_r,(g,r_f,1_{g_\mu})(u,g_1(y'_1,\ldots),\ldots,nil,\underline{z_1,\ldots,z_s}),\\
\qquad\qquad\qquad \ldots,\\
\qquad\qquad\qquad (g,r_f,s_{\mu_{g_\mu}})(u,g_1(y'_1,\ldots),\ldots,nil,\underline{z_1,\ldots,z_s})),\\
\qquad\quad z_1,\ldots,z_s).
\end{array}
$$

The intuition might be that such a state $(g, k_f, l_{g'})$ would—in contrast to the state $(k_f, l_{g'})$ introduced at the beginning of Subsection 4.7—not need all the $z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}$-positions as context parameters, because the relevant g at the end of Subsection 4.7 would be fixed. However, we will now use an example to show that such a construction is not feasible, because determining the rules for such $(g, k_f, l_{g'})$-states would lead to new problems.

Assume that for the composition of the mtt M_{count} from Example 3.8 with a

weakly single-use mtt containing (in R_2) the rules

$$\begin{aligned} g_1(succ(v_1), z_1) &\rightarrow g_2(v_1, \gamma_2(z_1)) \\ g_2(succ(v_1), z_1) &\rightarrow g_3(v_1, \gamma_3(z_1)) \\ g_3(succ(v_1), z_1) &\rightarrow g_1(v_1, \gamma_1(z_1)), \end{aligned}$$

we want to construct the rule for state $(g_1, 1_{count}, 1_{g_3})$ at δ. Since the unique occurrence of y_1 in $rhs_{count,\delta} = succ(count(u_1, count(u_2, y_1)))$ is in the first context parameter position of a call of *count* on u_2, we would have to construct—according to the idea in Subsection 4.8—some $(g, 1_{count}, 1_{g_3})$-call on u_2, i.e., the sought rule should have the form:

$$(g_1, 1_{count}, 1_{g_3})(\delta(u_1, u_2), y_{1,g_1}, y_{1,g_2}, y_{1,g_3}, z_1) \rightarrow (?, 1_{count}, 1_{g_3})(u_2, \ldots).$$

But how can we decide which g to use in the question mark position? It should be the state with which the inner *count*-call (on u_2) in $rhs_{count,\delta}$ is reached. Clearly, since we are trying to construct a rule for $(\underline{g_1}, 1_{count}, 1_{g_3})$, we can use the knowledge that the reduction on $rhs_{count,\delta}$ is started with g_1. But still, the necessary information depends on the input and hence cannot be determined statically. For example, for the input $u_1 = \epsilon$ the $count(u_2, y_1)$-call is reached with g_3:

$$\begin{aligned} & g_1(rhs_{count,\delta}[u_1 \leftarrow \epsilon], z_1) \\ \Rightarrow_{R_{count}} & g_1(succ(succ(count(u_2, y_1))), z_1) \\ \Rightarrow^*_{R_2} & \underline{g_3}(count(u_2, y_1), \gamma_3(\gamma_2(z_1))), \end{aligned}$$

whereas for the instantiation $u_1 = \delta(\epsilon, \epsilon)$ it is reached with g_2:

$$\begin{aligned} & g_1(rhs_{count,\delta}[u_1 \leftarrow \delta(\epsilon, \epsilon)], z_1) \\ \Rightarrow^*_{R_{count}} & g_1(succ(succ(succ(succ(count(u_2, y_1))))), z_1) \\ \Rightarrow^*_{R_2} & \underline{g_2}(count(u_2, y_1), \gamma_2(\gamma_1(\gamma_3(\gamma_2(z_1))))). \end{aligned}$$

As pointed out by one referee, this problem can be solved by equipping the mtt $M_{1;2}$ with the feature of regular look-ahead (Engelfriet & Vogler, 1985) to determine the relevant g for every instantiation of the recursion variables.

To achieve this in a functional programming setting, one would either have to compute the regular look-ahead information for every subtree of the input tree once and for all and annotate the tree accordingly, or to simulate the regular look-ahead feature by recomputing the necessary information whenever needed (e.g., with *test trees*, cf. Theorem 4.21 in (Engelfriet & Vogler, 1985)). The former strategy would run contrary to our aim of *eliminating* intermediate results, while the latter strategy can necessitate a significant amount of recomputations by considering subtrees repeatedly, and thus can even worsen the runtime complexity in some cases.

In contrast, lazy evaluation ensures that no superfluous computations are performed by the mtt resulting from Construction 5.1. Even though the rules for $(k_f, l_{g'})$-states might contain recursive calls of which not only those $z_{g_1,1}, \ldots, z_{g_\mu,s_\mu}$-parameter positions associated with one particular g are non-*nil*, for every concrete input only a subset of the context parameters is relevant and hence required under call-by-need (cf. the end of Subsection 4.7).

6 Practical Aspects

After the rather artificial example in Subsection 5.2, let us begin this section with a more practical example, namely the one coming from the introduction.

Example 6.1 (*composition of the mtts for pfx and aux from the introduction*) Composition of the mtts M_{pfx} and M_{aux} from Example 3.2 yields the mtt $M_{pfx;aux} = (H, \Sigma_{term}, \Omega_{ins}, e_{pfx;aux}, R_{pfx;aux})$ with $H = \{(pfx, aux)^{(3)}, (1_{pfx}, 1_{aux})^{(3)}\}$, set of rules $R_{pfx;aux}$:

$$
\begin{array}{rlll}
\text{(i)} & : (pfx, aux)(+(u_1, u_2), y, z) & \rightarrow (pfx, aux)(u_1, (pfx, aux)(u_2, y, \\
& & \qquad (1_{pfx}, 1_{aux})(u_1, \epsilon, ADD(z))), ADD(z)) \\
\text{(ii)} & : (pfx, aux)(\times(u_1, u_2), y, z) & \rightarrow (pfx, aux)(u_1, (pfx, aux)(u_2, y, \\
& & \qquad (1_{pfx}, 1_{aux})(u_1, \epsilon, MUL(z))), MUL(z)) \\
\text{(iii)} & : (pfx, aux)(A, y, z) & \rightarrow y \\
\text{(iv)} & : (pfx, aux)(B, y, z) & \rightarrow y \\
\text{(v)} & : (1_{pfx}, 1_{aux})(+(u_1, u_2), y, z) & \rightarrow (1_{pfx}, 1_{aux})(u_2, \epsilon, \\
& & \qquad (1_{pfx}, 1_{aux})(u_1, \epsilon, ADD(z))) \\
\text{(vi)} & : (1_{pfx}, 1_{aux})(\times(u_1, u_2), y, z) & \rightarrow (1_{pfx}, 1_{aux})(u_2, \epsilon, \\
& & \qquad (1_{pfx}, 1_{aux})(u_1, \epsilon, MUL(z))) \\
\text{(vii)} & : (1_{pfx}, 1_{aux})(A, y, z) & \rightarrow LOAD_A(z) \\
\text{(viii)} & : (1_{pfx}, 1_{aux})(B, y, z) & \rightarrow LOAD_B(z)\,,
\end{array}
$$

and $e_{pfx;aux} = (pfx, aux)(x, (1_{pfx}, 1_{aux})(x, \epsilon, \epsilon), \epsilon)$.

Note that above only the last two ϵ-symbols—in $e_{pfx;aux}$—are "real" ϵs, while all the others are dummy *nil*-symbols that were replaced by ϵ. $\diamond$

6.1 Post-processing

The mtt $M_{pfx;aux}$ constructed in Example 6.1 looks rather complicated. In particular, it is not quite the optimised program that we promised in the introduction. The reason is twofold. Firstly, we observe that the construction introduces context parameters that are *superfluous*, in the sense that they will never (for no possible input tree) influence the output generated by a state. This is the case for parameter z of state (pfx, aux) and for parameter y of state $(1_{pfx}, 1_{aux})$. Secondly, the state (pfx, aux) never really performs any actual computation. No matter on which input tree it is called, it will always just project on its first context parameter. We call a state that always projects on one and the same context parameter a *copy-state*, because this phenomenon is similar to superfluous data traversals due to *copy rules* of attribute grammars (Correnson *et al.*, 1999).

The first problem is caused by the need for the composition construction to be as general as possible. There exist more complicated mtts where the additionally created context parameters are really needed. However, the presence of these superfluous context parameters does not influence the efficiency of our constructed mtts if lazy evaluation is used and we take the number of reduction steps as efficiency measure. Nevertheless, we still want to get rid of them, because they obscure the

programs and, moreover, *do* influence the efficiency if more detailed measures are used, such as taking into account the cost of allocating memory for representing function closures.

The second problem is more serious as it leads to superfluous traversals through the input data structure and so contradicts our aim of *optimising* functional programs by eliminating intermediate data structures.

In our example $M_{pfx;aux}$ it is pretty obvious how to solve the two problems, but in general both, superfluous context parameters and copy-states, are more difficult to detect. Voigtländer (2001) developed two mechanisable constructions (based on computing finite fixpoints) for post-processing and optimising mtts obtained from the composition construction, which allow to detect and remove all superfluous context parameters and copy-states, respectively.

Example 6.2 (*post-processing*)
Consider the mtt $M_{pfx;aux}$ from Example 6.1.

Construction 4.8 from (Voigtländer, 2001) detects that the second context parameter of (pfx, aux) and the first context parameter of $(1_{pfx}, 1_{aux})$ will never influence the output computed by these states, hence their ranks can be reduced (this is a kind of useless variable elimination).

Then, Construction 4.17 from (Voigtländer, 2001) detects that the state (pfx, aux) always projects on its remaining context parameter, hence it can be discarded, simplifying the initial expression.

As result we obtain the mtt $M'_{pfx;aux} = (H', \Sigma_{term}, \Omega_{ins}, e'_{pfx;aux}, R'_{pfx;aux})$ with $H' = \{(1_{pfx}, 1_{aux})'^{(2)}\}$, set of rules $R'_{pfx;aux}$:

$$\begin{array}{rll}
\text{(v)'} : & (1_{pfx}, 1_{aux})'(+(u_1, u_2), z) & \rightarrow (1_{pfx}, 1_{aux})'(u_2, (1_{pfx}, 1_{aux})'(u_1, ADD(z))) \\
\text{(vi)'} : & (1_{pfx}, 1_{aux})'(\times(u_1, u_2), z) & \rightarrow (1_{pfx}, 1_{aux})'(u_2, (1_{pfx}, 1_{aux})'(u_1, MUL(z))) \\
\text{(vii)'} : & (1_{pfx}, 1_{aux})'(A, z) & \rightarrow LOAD_A(z) \\
\text{(viii)'} : & (1_{pfx}, 1_{aux})'(B, z) & \rightarrow LOAD_B(z)\,,
\end{array}$$

and $e'_{pfx;aux} = (1_{pfx}, 1_{aux})'(x, \epsilon)$. $\diamond$

Note that the final program obtained in the previous example indeed corresponds to the optimised program of rules (x)–(xiii) in the introduction. Strictly speaking, the elimination of copy-states is sound only for computations on finite trees, because when used with infinite data structures—as they are possible in lazy functional languages—it can transform non-terminating programs into terminating ones (which we do not consider an obstacle for automation, for further discussion see Voigtländer, 2001).

6.2 Implementation in the Haskell$^+$ System

Construction 5.1 has been implemented in the Haskell$^+$ program transformation system (Lescher, 1999; Höff *et al.*, 2001). As an example for applying the system we consider the following program.

Example 6.3 (*Haskell*$^+$)

```
begindata Data
  data List = A List | B List | E
enddata

beginmag App [Mac,Mat,Su,Swp,Tl,Wp,Wsu,Xlin,Xnd,Ylin,Ynd]
  input Data
  syn app :: List -> List -> List
  app (A u) y = A (app u y)
  app (B u) y = B (app u y)
  app E     y = y
endmag
```

The above program defines a Haskell data type `List` and a function `app` that computes concatenation on the `List` type. It contains special keywords like `begindata`, `endmag`, `input` and `syn` that are used in the Haskell$^+$ language to specify ranked alphabets (in `begindata`–`enddata` blocks) and tree transducers (in `beginmag`–`endmag` blocks). The list behind the "`beginmag App`" statement is an enumeration of properties that were recognised by the analysing phase of our system. In particular, the list items `Mac`, `Ylin` and `Wsu` tell us that we have an mtt (without initial expression; see below) that is non-copying and weakly single-use.

Thus, the construction presented in Subsection 5.1 is applicable to produce the rules for the states $(\texttt{app}, \texttt{app})$ and $(1_{\texttt{app}}, 1_{\texttt{app}})$, which are denoted by `app_app` and `par_1app_1app` in our system. If instructed to perform this composition of `App` with itself, the Haskell$^+$ system introduces a new mtt `App_App` into the output program:

```
{-# RULES "COMPOSITION" forall u y1' z1.
  app (app u y1') z1 = app_app u (app y1' (par_1app_1app u E z1)) z1
#-}
beginmag App_App [Mac,Mat,Su,Wp,Wsu,Xlin,Xnd,Ylin]
  input Data
  syn app_app       :: List -> List -> List -> List
  syn par_1app_1app :: List -> List -> List -> List
  app_app       (A u1) y1 z1 = A (app_app u1 y1 z1)
  app_app       (B u1) y1 z1 = B (app_app u1 y1 z1)
  app_app       E      y1 z1 = y1
  par_1app_1app (A u1) y1 z1 = par_1app_1app u1 E z1
  par_1app_1app (B u1) y1 z1 = par_1app_1app u1 E z1
  par_1app_1app E      y1 z1 = z1
endmag
```

Also, the system outputs—in a special comment above the resulting mtt—an equation that can be used as rewrite rule in order to take advantage of the composition transformation. Note that the equation's right-hand side contains an occurrence of the state `app` (originating from the second of the two composed mtts), because its left-hand side is more general than just a composite of two mtts' initial expressions, in that it is not only variable over the input tree `u`, but also over the context parameters `y1'` and `z1`. This equation corresponds to the rewrite system *Pair* from Construction 5.1. Its correctness can be justified by appealing to statement **II**(a)**i** in Lemma A.15 of (Voigtländer & Kühnemann, 2003).

An optimising compiler would have to detect appropriate places in the program where such an equation produced by the composition construction can be applied. By using the form of a *rule pragma* for the Glasgow Haskell Compiler (Peyton Jones *et al.*, 2001), this task is left to the built-in simplifier.

The Haskell$^+$ system also implements post-processing steps as mentioned in the previous subsection. Applying these to the mtt `App_App` yields a simplified mtt `App_App'` and further rule pragmas:

```
{-# RULES "REMOVE SUPERFLUOUS CONTEXT PARAMETERS" forall u y1 z1.
  app_app u y1 z1 = app_app' u y1
#-}
{-# RULES "REMOVE SUPERFLUOUS CONTEXT PARAMETERS" forall u y1 z1.
  par_1app_1app u y1 z1 = par_1app_1app' u z1
#-}
beginmag App_App' [Mac,Mat,Su,Swp,Tl,Wp,Wsu,Xlin,Xnd,Ylin,Ynd]
  input Data
  syn app_app'       :: List -> List -> List
  syn par_1app_1app' :: List -> List -> List
  app_app'       (A u1) y1 = A (app_app' u1 y1)
  app_app'       (B u1) y1 = B (app_app' u1 y1)
  app_app'       E      y1 = y1
  par_1app_1app' (A u1) z1 = par_1app_1app' u1 z1
  par_1app_1app' (B u1) z1 = par_1app_1app' u1 z1
  par_1app_1app' E      z1 = z1
endmag

{-# RULES "ELIMINATE COPY-STATES" forall u z1.
  par_1app_1app' u z1 = z1
#-}
```

Applying the equations introduced in the rule pragmas from left to right, we get the following calculation:

```
app (app u y1') z1
   =  (by rule "COMPOSITION")
app_app u (app y1' (par_1app_1app u E z1)) z1
   =  (by rules "REMOVE SUPERFLUOUS CONTEXT PARAMETERS")
app_app' u (app y1' (par_1app_1app' u z1))
   =  (by rule "ELIMINATE COPY-STATES")
app_app' u (app y1' z1)
```

Since the defining equations that were constructed for `app_app'` are the same as those for `app`, this corresponds to a well-known optimising transformation, namely making use of the associativity of the concatenation function. ◇

We would also like to integrate the presented composition construction and related techniques into an optimising functional compiler, without the need for user interaction. First results of implementing an analysis phase to detect mtts in Haskell source programs (without annotations as in Haskell$^+$) and a simple transformation to compose tdtts only are promising (Reuther, 2002), but much remains to be done.

7 Related Work on Eliminating Intermediate Results

In this section we give qualitative comparisons of our mtt composition technique with classical deforestation and shortcut deforestation.

7.1 Classical Deforestation

In order to facilitate the comparison, we give a description of classical deforestation (Wadler, 1990; Chin, 1994) tailored to mtts, which has also been implemented in the Haskell$^+$ system. Note that mtts may be defined using nesting of terms in context parameter positions and hence are not *treeless* programs as required for proving termination of Wadler's original deforestation algorithm. This problem can be solved by abstracting context parameters using let-expressions explicitly (Hamilton & Jones, 1992) or implicitly (Kühnemann, 1999). We instead give a direct presentation along the lines of the *define/instantiate/unfold/fold*-strategy (Burstall & Darlington, 1977) for mtts as described in Subsections 4.1 and 4.2.

Construction 7.1 (*classical deforestation for mtts*)
Let $M_1 = (F, \Sigma, \Delta, e_1, R_1)$ and $M_2 = (G, \Delta, \Omega, e_2, R_2)$ be (unrestricted) mtts. Deforestation does not create an mtt, but a program consisting of the set of functions

$$H_{Def} = \{(f,g)^{(r+s+1)} \mid f \in F^{(r+1)}, g \in G^{(s+1)}\}\,,$$

the expression

$$e_{Def} = nf(\Rightarrow_{R_2 \cup Fold}, e_2[x \leftarrow e_1])\,,$$

and the set R_{Def}, containing for every $f \in F^{(r+1)}$, $g \in G^{(s+1)}$ and $\sigma \in \Sigma^{(p)}$ the rule:

$$(f,g)(\sigma(u_1,\ldots,u_p), y_1,\ldots,y_r,z_1,\ldots,z_s) \to nf(\Rightarrow_{R_2 \cup Fold}, g(rhs_{f,\sigma}, z_1,\ldots,z_s)).$$

The rewrite system *Fold* (over $F \cup G \cup H_{Def}$ and $\{u\} \cup Y' \cup Z$) used above is defined as follows:

$$\mathit{Fold} \;:\; g(f(u,y_1',\ldots,y_r'),z_1,\ldots,z_s) \to (f,g)(u,y_1',\ldots,y_r',z_1,\ldots,z_s)\,,\quad \forall g \in G^{(s+1)}, f \in F^{(r+1)}$$

The reduction relation $\Rightarrow_{R_2 \cup Fold}$ is confluent and terminating, hence unique normal forms exist. Now, we claim that for every $t \in T_\Sigma$:

$$nf(\Rightarrow_{R_2}, e_2[x \leftarrow nf(\Rightarrow_{R_1}, e_1[x \leftarrow t])]) = nf(\Rightarrow_{R_{Def} \cup R_1 \cup R_2}, e_{Def}[x \leftarrow t]). \quad \square$$

We do not prove the claim here, but we note that e_{Def} and R_{Def} correspond exactly to the result of classical deforestation for the expression $e_2[x \leftarrow e_1]$ and the program $R_1 \cup R_2$, except that the deforestation algorithm would construct only those functions in H_{Def} that are really needed for evaluating e_{Def} and hence sometimes delivers a smaller program.

Example 7.2 (*classical deforestation for the introductory example*)
Consider the mtts M_{pfx} and M_{aux} from Example 3.2. According to Construction 7.1, we obtain $H_{Def} = \{(pfx, aux)^{(3)}\}$ and *Fold* contains the rule:

$$aux(pfx(u, y_1'), z_1) \to (pfx, aux)(u, y_1', z_1).$$

Then, $e_{Def} = (pfx, aux)(x, \epsilon, \epsilon)$ and R_{Def} contains the rules:

$$\begin{array}{lll} (pfx, aux)(+(u_1, u_2), y_1, z_1) & \to & (pfx, aux)(u_1, pfx(u_2, y_1), ADD(z_1)) \\ (pfx, aux)(\times(u_1, u_2), y_1, z_1) & \to & (pfx, aux)(u_1, pfx(u_2, y_1), MUL(z_1)) \\ (pfx, aux)(A, y_1, z_1) & \to & aux(y_1, LOAD_A(z_1)) \\ (pfx, aux)(B, y_1, z_1) & \to & aux(y_1, LOAD_B(z_1)). \end{array}$$

Note that here deforestation removed only parts of the intermediate result, namely those that occurred "outside" topmost recursive calls in the original rules R_{pfx}. This can be seen in the following derivation of the deforested program for input $+(A, B)$:

$$\begin{array}{ll} & (pfx, aux)(+(A, B), \underline{\epsilon}, \epsilon) \\ \Rightarrow_{R_{Def}} & (pfx, aux)(A, pfx(B, \underline{\epsilon}), ADD(\epsilon)) \\ \Rightarrow_{R_{Def}} & aux(pfx(B, \underline{\epsilon}), LOAD_A(ADD(\epsilon))) \\ \Rightarrow_{R_{pfx}} & aux(\underline{B(\epsilon)}, LOAD_A(ADD(\epsilon))) \\ \Rightarrow_{R_{aux}} & aux(\underline{\epsilon}, LOAD_B(LOAD_A(ADD(\epsilon)))) \\ \Rightarrow_{R_{aux}} & LOAD_B(LOAD_A(ADD(\epsilon))), \end{array}$$

where the underlined parts have not been eliminated. ◇

The reason why classical deforestation does not reach intermediate results "inside" context parameters, whereas mtt composition does, lies in the different treatment of the $y_1', \ldots, y_r'$ by the *Fold*- and *Pair*-rules, respectively. While deforestation simply copies them without manipulation, our composition construction sends the states of M_2 into these context parameters of M_1, by *Pair*-rules of the following form:

$$g(f(u, y_1', \ldots, y_r'), z_1, \ldots, z_s) \to (f, g)(u, g_1(y_1', \ldots), \ldots, g_\mu(y_r', \ldots), z_1, \ldots, z_s).$$

On the other hand, classical deforestation (using explicit or implicit let-abstractions) is applicable to a wider class of programs than just to mtts like our construction.

A more formal comparison between the compositions $TOP; MAC \subseteq MAC$ and $MAC; TOP \subseteq MAC$ and classical deforestation is drawn by Kühnemann (1999) and Höff (1999).

7.2 Shortcut Deforestation

Shortcut deforestation achieves elimination of intermediate results by expressing producers and consumers with certain higher-order, polymorphic combinators, the composition of which can be transformed by `foldr/build-` (Gill *et al.*, 1993), `foldr/augment-` (Gill, 1996) or `cata/augment`-rules (Johann, 2001).

In the framework of mtt composition, this means that the computation performed by the states of the consuming mtt M_2 needs to be expressed as a *catamorphism* that is tupled (in order to capture mutual recursion) and higher-order (in order to capture context parameters). Such a representation can be synthesized from the

rules of M_2 in a systematic way. On the other hand, all output symbols would need to be abstracted uniformly from the rules of the producing mtt M_1 in a polymorphic way. But this is problematic, because parts of the produced output can be "hidden" in the context parameters. One solution would be to prepare those parts for abstraction via an additional traversal, which is dismissed already by Gill (1996), because subsequent removal of the traversal introduced thus cannot be guaranteed. The alternative would be to use the generalisation `augment` of `build`, substituting specific values in place of nullary symbols. However, also this strategy fails, e.g., for a non-copying mtt with rules

$$\begin{array}{lcl} app'(A(u_1), y_1, y_2) & \rightarrow & A(app'(u_1, y_2, y_1)) \\ app'(B(u_1), y_1, y_2) & \rightarrow & B(app'(u_1, y_2, y_1)) \\ app'(\epsilon, y_1, y_2) & \rightarrow & y_1 \;, \end{array}$$

where the swapping of context parameters in every step prevents us from knowing beforehand which of the two is to be substituted at the end of the output list.

Hence, the effect to eliminate intermediate results in accumulating parameters as accomplished by mtt composition cannot be achieved by shortcut deforestation in general. If no context parameters at all are present—i.e., we are dealing with tdtts only—shortcut deforestation and tree transducer composition correspond to each other (Jürgensen & Vogler, 2001).

Note that Svenningsson (2002) disputes the above use of higher-order catamorphisms for shortcut deforestation, because it introduces suspended function calls. He proposes a `destroy/unfoldr`-rule, which however handles accumulating parameters only for *consumers* of intermediate lists, and hence also does not achieve deforestation *inside* accumulating parameters as mtt composition does.

On the other hand, there are also functions that form no mtts, but can be expressed using the above mentioned polymorphic combinators for the different approaches to shortcut deforestation.

8 Efficiency Considerations

We discuss efficiency aspects of our transformation technique, by motivating work on formally proving improvements with respect to abstract efficiency measures, and by comparing actual runtimes of programs transformed with mtt composition and with the methods covered in the previous section.

8.1 Motivation for Formal Efficiency Analysis

We have seen two practical examples (in the introduction and in Example 6.3) for which our approach of eliminating intermediate results yields a program that performs fewer call-by-need reduction steps to produce the final output than the original program. However, this needs not to be the case in general.

Example 8.1 (*possible loss of efficiency*)
Continuing the example from the introduction, consider the mtt $M'_{pfx;aux}$ from Example 6.2, but name the state $(1_{pfx}, 1_{aux})'$ as ins. Assume that we want to count

how many *ADD*-, how many *MUL*- and how many *LOAD*-instructions occur in an instruction sequence produced by *ins* for some term t. This can be done by computing

$$e = ic(ins(t,\epsilon), zero, zero, zero)\,,$$

with a state *ic* that uses three context parameters to accumulate the numbers, as defined in the mtt $M_{ic} = (\{ic^{(4)}\}, \Omega_{ins}, Nat \cup \{\omega^{(3)}\}, ic(x, zero, zero, zero), R_{ic})$ with the following set of rules R_{ic}:

$$\begin{array}{lcl}
ic(ADD(v_1), z_1, z_2, z_3) & \rightarrow & ic(v_1, succ(z_1), z_2, z_3)\\
ic(MUL(v_1), z_1, z_2, z_3) & \rightarrow & ic(v_1, z_1, succ(z_2), z_3)\\
ic(LOAD_A(v_1), z_1, z_2, z_3) & \rightarrow & ic(v_1, z_1, z_2, succ(z_3))\\
ic(LOAD_B(v_1), z_1, z_2, z_3) & \rightarrow & ic(v_1, z_1, z_2, succ(z_3))\\
ic(\epsilon, z_1, z_2, z_3) & \rightarrow & \omega(z_1, z_2, z_3).
\end{array}$$

Composition of $M'_{pfx;aux}$ with M_{ic}, and post-processing as in Subsection 6.1, yields that e can be replaced by

$$e' = \omega(ic_A(t, zero), ic_M(t, zero), ic_L(t, zero))\,,$$

with set of rules $R_{ins,ic}$:

$$\begin{array}{lcl}
ic_A(+(u_1, u_2), z_1) & \rightarrow & succ(ic_A(u_1, ic_A(u_2, z_1)))\\
ic_A(\times(u_1, u_2), z_1) & \rightarrow & ic_A(u_1, ic_A(u_2, z_1))\\
ic_A(A, z_1) & \rightarrow & z_1\\
ic_A(B, z_1) & \rightarrow & z_1\\
ic_M(+(u_1, u_2), z_2) & \rightarrow & ic_M(u_1, ic_M(u_2, z_2))\\
ic_M(\times(u_1, u_2), z_2) & \rightarrow & succ(ic_M(u_1, ic_M(u_2, z_2)))\\
ic_M(A, z_2) & \rightarrow & z_2\\
ic_M(B, z_2) & \rightarrow & z_2\\
ic_L(+(u_1, u_2), z_3) & \rightarrow & ic_L(u_1, ic_L(u_2, z_3))\\
ic_L(\times(u_1, u_2), z_3) & \rightarrow & ic_L(u_1, ic_L(u_2, z_3))\\
ic_L(A, z_3) & \rightarrow & succ(z_3)\\
ic_L(B, z_3) & \rightarrow & succ(z_3)\,,
\end{array}$$

where ic_A, ic_M and ic_L abbreviate $(1_{ins}, 1_{ic})'$, $(1_{ins}, 2_{ic})'$ and $(1_{ins}, 3_{ic})'$, respectively (the quote signs stem from the post-processing).

Notice that on the one hand, the transformed expression e' indeed avoids the creation of the intermediate result produced in the original expression e. But on the other hand, evaluation of e performed only one traversal over t (with *ins*) and one traversal over the intermediate result of same size (with *ic*), whereas evaluation of e' performs *three* traversals over t (with ic_A, ic_M and ic_L). ◇

In the light of the previous example it is important to develop decision procedures that determine when mtt composition should be applied, i.e., when Construction 5.1 is guaranteed to improve the efficiency of a program. For the cases that one of the involved mtts is a tdtt, such a systematic study was begun by Kühnemann (1999) and Höff (1999), and continued using a more general approach by Voigtländer (2002a). The analysis technique from the latter paper also scales for the case that both

involved mtts use context parameters. Hence, it can be used to prove sufficient conditions under which Construction 5.1 produces a program that performs fewer call-by-need reduction steps to reach normal form than the original program. The development of such criteria is work in progress. The post-processing phase never leads to an efficiency deterioration, but it would be useful to characterize cases where it enables an actual improvement.

Voigtländer (2002b) proposes an alternative composition construction that produces circular programs instead of mtts and avoids the problem of multiple traversals (as in Example 8.1) through tupling. However, this *lazy composition* algorithm does not handle mutually recursive functions, and the costs incurred by tupling make formal efficiency considerations more difficult.

8.2 Measurements

In order to demonstrate the efficiency gains realised by our technique in practice, we perform measurements for several examples. For each example, we compare execution times for multiple runs of different program versions with varying input sizes. The program versions considered are: (i) the original program, (ii) the program obtained by applying mtt composition, i.e. Construction 5.1, (iii) the program obtained from (ii) by additionally applying post-processing as discussed in Subsection 6.1, (iv) the program obtained from the original one by applying classical deforestation as presented in Construction 7.1, and (v) the program obtained by applying shortcut deforestation, where the first alternative discussed in Subsection 7.2 is used to abstract parts of the intermediate result inside context parameters via an additional traversal. To use the `augment`-alternative for (v) would make no difference for the examples considered here.

The different program versions are coded as ordinary Haskell source, compiled with the Glasgow Haskell Compiler (version 5.04.1, optimisation level `-O`) and run on a Sun Ultra 10 workstation (300MHz, 256MB). The runtimes (in seconds) shown in the measurement tables below are split into the time spent for actual expression evaluation (the first summand) and the time spent on garbage collection (the second summand) as obtained from the statistics produced using the runtime system option `-s`. The given execution times include the test frame with generation of input data and consumption of final output. This is unavoidable, because a more detailed cost centre profiling—to separate the execution times for the tested algorithms from their test frame—would corrupt the precision of the measured garbage collection times considerably.

Of the six examples that we consider, three have already occurred in the paper: Tables 1–3 contain measurements for the example from the introduction, Example 6.3 and Example 8.1. Table 4 covers an interesting variation of Example 8.1, in which the final output is only partially demanded. Tables 5 and 6 cover examples on standard Haskell lists as opposed to tree structures.

The measurements for the introductory example $aux(pfx(t,\epsilon),\epsilon)$ on fully balanced binary trees of different heights h in Table 1 show a considerable runtime improvement by mtt composition, in particular after post-processing has been applied

Table 1. $aux(pfx(t,\epsilon),\epsilon)$, *n runs with* $size(t) = 2^{h+1} - 1$

$n \times h$:	60000×5	2000×10	500×12	60×15	2×20
original	2.6+0.1=2.7	2.8+0.3=3.1	2.7+1.1=3.8	2.6+6.0=8.6	3.0+10.1=13.1
compos.	2.4+0.1=2.5	2.5+0.1=2.6	2.5+0.1=2.6	2.5+2.2=4.7	2.6+ 1.9= 4.5
+post-p.	1.8+0.0=1.8	1.8+0.1=1.9	1.8+0.1=1.9	1.8+1.9=3.7	1.9+ 1.9= 3.8
deforest.	2.5+0.1=2.6	2.8+0.3=3.1	2.8+0.9=3.7	2.8+5.7=8.5	3.0+10.1=13.1
shortcut	1.6+0.1=1.7	1.6+0.2=1.8	1.7+0.4=2.1	1.7+3.3=5.0	1.8+ 6.2= 8.0

Table 2. `app (app l l) l`, *n runs with* $size($`l`$) = s$

$n \times s$:	40000×100	4000×1000	2000×2000	1000×4000	800×5000
original	6.5+0.1=6.6	6.5+0.2=6.7	6.6+0.3=6.9	6.7+0.2= 6.9	6.9+3.8=10.7
compos.	5.6+0.1=5.7	5.7+0.1=5.8	5.7+0.1=5.8	5.8+0.1= 5.9	5.9+0.1= 6.0
+post-p.	5.2+0.1=5.3	5.4+0.1=5.5	5.3+0.2=5.5	5.4+0.1= 5.5	5.4+0.1= 5.5
deforest.	5.3+0.1=5.4	5.4+0.1=5.5	5.4+0.1=5.5	5.5+0.1= 5.6	5.7+0.1= 5.8
shortcut	8.1+0.2=8.3	8.3+0.2=8.5	8.5+0.2=8.7	8.2+5.7=13.9	8.0+5.7=13.7

(yielding the function *ins* from the introduction). As indicated in Example 7.2, only minimal parts of the intermediate tree can be eliminated by classical deforestation, resulting in the observation of almost no runtime improvement. The performance of the program produced by shortcut deforestation is on a par with that of the program produced by our techniques for relatively small input trees, but for larger input trees the shortcut deforested program has a considerably higher garbage collection overhead.

The measurements in Table 2, where the original expression is a left-associative concatenation of three identical lists using the function `app` from Example 6.3, show about the same runtime improvement by our approach and by classical deforestation, whereas the shortcut deforestation technique decreases the performance in this example.

Table 3 gives the runtimes measured for the differently transformed versions of $ic(ins(t,\epsilon), zero, zero, zero)$ from the previous subsection on fully balanced binary trees of varying heights h. While no significant change in the runtime behaviour is observed for classical and shortcut deforestation, our technique increases the time spent in expression evaluation for the reasons indicated in Example 8.1. Interestingly though, for large input trees the garbage collection times become dominant, such that then the elimination of the intermediate result pays off, even at the price of introducing an additional traversal.

An interesting variation of Example 8.1 can be obtained by considering the case that the final output needs not to be computed to its full normal form, but instead only a part of this output is demanded by the program context in which it occurs.

Table 3. $ic(ins(t, \epsilon), zero, zero, zero)$, *n runs with* $size(t) = 2^{h+1} - 1$

$n \times h$:	60000×5	2000×10	500×12	60×15	2×20
original	2.3+0.1=2.4	2.4+0.3=2.7	2.5+0.6=3.1	2.3+4.6=6.9	2.7+9.2=11.9
compos.	5.3+0.2=5.5	5.7+0.2=5.9	5.7+2.9=8.6	5.7+2.9=8.6	6.3+3.1= 9.4
+post-p.	4.0+0.1=4.1	4.2+0.1=4.3	4.4+1.5=5.9	4.4+2.2=6.6	4.9+2.4= 7.3
deforest.	2.3+0.1=2.4	2.4+0.2=2.6	2.6+0.5=3.1	2.5+4.4=6.9	2.6+9.3=11.9
shortcut	2.3+0.1=2.4	2.5+0.2=2.7	2.6+0.6=3.2	2.5+4.5=7.0	2.7+9.3=12.0

Table 4. $pr_2(ic(ins(t, \epsilon), zero, zero, zero))$, *n runs with* $size(t) = 2^{h+1} - 1$

$n \times h$:	60000×5	2000×10	500×12	60×15	2×20
original	1.6+0.0=1.6	1.6+0.2=1.8	1.6+0.7=2.3	1.5+4.2=5.7	1.6+9.3=10.9
compos.	1.8+0.0=1.8	1.9+0.1=2.0	2.0+0.1=2.1	2.3+0.1=2.4	2.4+0.1= 2.5
+post-p.	0.9+0.0=0.9	0.9+0.0=0.9	1.1+0.0=1.1	1.3+0.0=1.3	1.6+0.0= 1.6
deforest.	1.5+0.1=1.6	1.6+0.2=1.8	1.6+0.7=2.3	1.5+4.1=5.6	1.6+9.2=10.8
shortcut	1.7+0.0=1.7	1.7+0.2=1.9	1.7+0.6=2.3	1.6+4.1=5.7	1.7+9.4=11.1

This can be simulated by consuming the output with a projection function pr_2 that has the following defining rule:

$$pr_2(\omega(x_1, x_2, x_3)) \to x_2.$$

Table 4 contains the runtime measurements for the thus adapted example and shows that then the program obtained by mtt composition plus post-processing outperforms all the other program versions, because it needs only one traversal over the input to compute the demanded part of the final output.

As an example for applying our technique on standard Haskell lists—as opposed to trees over ranked alphabets—consider the following function definitions:

```
enum :: Int -> [Int] -> [Int]
enum 0     ys = ys
enum (x+1) ys = enum x (x:ys)

even :: [Int] -> [Int]              odd :: [Int] -> [Int]
even []     = []                    odd []     = []
even (x:xs) = x:(odd xs)            odd (x:xs) = even xs
```

The initial expression `enum m []` can be used to enumerate in ascending order all the non-negative integers that are smaller than a given one, by accumulating them in the second parameter. By treating “`x:`” as a special constructor symbol, the function `enum` together with this initial expression can be regared as an mtt (cf. Kühnemann & Voigtländer, 2001). Likewise, the mutually recursive unary functions `even` and `odd`—selecting every other element of a list—form a tdtt. The

Table 5. `even (enum m [])`, *n runs*

$n \times$ `m`:	50000×100	10000×500	5000×1000	2000×2500	1000×5000
original	2.7+0.1=2.8	2.7+0.2=2.9	2.5+0.4=2.9	2.6+0.8=3.4	2.7+1.5=4.2
compos.	1.4+0.1=1.5	1.4+0.1=1.5	1.4+0.2=1.6	1.4+0.5=1.9	1.6+0.8=2.4
+post-p.	1.4+0.1=1.5	1.4+0.1=1.5	1.4+0.2=1.6	1.4+0.5=1.9	1.6+0.8=2.4
deforest.	2.7+0.1=2.8	2.7+0.2=2.9	2.6+0.4=3.0	2.7+0.7=3.4	2.6+1.5=4.1
shortcut	1.4+0.1=1.5	1.4+0.1=1.5	1.3+0.3=1.6	1.3+0.6=1.9	1.3+1.0=2.3

modular program `even (enum m [])`—enumerating all the non-negative even integers smaller than `m`—thus represents the composition of an mtt with a tdtt, using an intermediate list as "glue". The Haskell$^+$ system currently does not implement the transformation of programs on standard lists, hence the measurements in Table 5 were obtained from hand-transformed program versions. They show that our technique and shortcut deforestation achieve a comparable efficiency improvement over the original program and the result of classical deforestation. Note that the second and the third line of the table contain identical measurements, because post-processing as described in Subsection 6.1 is usually only necessary in cases where none of the two mtts involved in the composition is a tdtt.

Such a case emerges by a variation of the previous example, the aim being to again enumerate the non-negative even integers smaller than a given `m`, but to additionally assemble the integers previously discarded by `odd` towards the end of the output list. This is achieved by enriching each of the functions `even` and `odd` with an accumulating parameter, yielding an mtt consisting of the following two functions:

```
even' :: [Int] -> [Int] -> [Int]    odd' :: [Int] -> [Int] -> [Int]
even' []     zs = zs                odd' []     zs = zs
even' (x:xs) zs = x:(odd' xs zs)    odd' (x:xs) zs = even' xs (x:zs)
```

The measurements for the differently transformed versions of the new program `even' (enum m []) []` in Table 6 show a solid improvement by our technique after post-processing, whereas classical deforestation has almost no effect and shortcut deforestation even leads to an efficiency deterioration. This failure of shortcut deforestation is mainly due to garbage collection overheads, probably caused by the introduction of a sequence of suspended function calls as discussed by Svenningsson (2002).

9 Tree Transducer Theory Results

In Subsection 5.1 we have presented a construction that composes a non-copying mtt and a weakly single-use one into a single mtt. It is natural to ask whether these two restrictions also work the other way round, i.e., whether we also have $MAC_{wsu}; MAC_{nc} \subseteq MAC$. We will show that this is not the case by giving a counterexample. In fact, we will even show the stronger result $MAC_{su}; MAC_{nc} \not\subseteq MAC$.

Table 6. `even' (enum m []) []`, n *runs*

$n \times$ `m`:	50000×100	10000×500	5000×1000	2000×2500	1000×5000
original	3.8+0.2=4.0	3.8+0.3=4.1	3.9+0.5=4.4	4.0+1.2=5.2	3.9+2.3=6.2
compos.	4.3+0.2=4.5	4.3+0.2=4.5	4.2+0.4=4.6	4.2+0.9=5.1	4.2+1.7=5.9
+post-p.	3.4+0.1=3.5	3.2+0.2=3.4	3.3+0.3=3.6	3.3+0.5=3.8	3.3+1.0=4.3
deforest.	3.9+0.1=4.0	3.8+0.3=4.1	3.9+0.4=4.3	4.1+1.0=5.1	4.1+2.1=6.2
shortcut	3.9+0.2=4.1	3.9+0.5=4.4	3.9+0.8=4.7	4.0+2.0=6.0	4.3+3.7=8.0

Firstly, we quote a classical result on mtts (Theorem 3.24 in (Engelfriet & Vogler, 1985)), namely that the heights of their output trees are exponentially bounded by the heights of their input trees.

Lemma 9.1 (*exponential height-height bound for mtts*)
Let $M = (F, \Sigma, \Delta, e, R)$ be an mtt. There exists a constant $c \in \mathbb{N}$ such that $height(\tau(M)(t)) \leq c^{height(t)}$ for every $t \in T_\Sigma$. $\square$

The negative result can now be shown by counterexample.

Theorem 9.2 (*a symmetric composition construction cannot exist*)
$MAC_{su}; MAC_{nc} \nsubseteq MAC$

Proof
Consider the mtts M_{count} and M_{exp} from Example 3.8, which are single-use and non-copying, respectively, and assume the existence of an mtt M such that $\tau(M) = \tau(M_{count}); \tau(M_{exp})$. Then, for every $t \in T_{\Delta_{bin}}$, $height(\tau(M)(t)) = 2^{size(t)}$ holds by the statements in items 2 and 3 of Example 3.8. Taking for every $h \in \mathbb{N}$, for t the fully balanced binary tree of height h over the ranked alphabet Δ_{bin}, we get $size(t) = 2^{h+1} - 1$ and thus $height(\tau(M)(t)) = 2^{(2^{h+1}-1)}$. Hence, the height of the output tree is double exponential in the height of the input tree, contradicting Lemma 9.1. $\square$

Results of the same style as Lemma 9.1—relating input- and output-height or -size—are also available for restricted mtts (and will be summarised in a table below), but not yet for the class of tree transductions induced by non-copying mtts. It would be useful to know such a bound, because—as demonstrated in the proof of Theorem 9.2—these kinds of results can help in reasoning about the expressiveness of various classes of tree transducers.

By using the composition result $MAC_{nc}; MAC_{wsu} \subseteq MAC$, we are going to show that for non-copying mtts the output-*size* is exponentially bounded by the input-*height*. In order to do so, we construct for every ranked alphabet Δ a weakly single-use mtt M_Δ that computes the sizes of trees over Δ as natural numbers in monadic representation over the ranked alphabet Nat.

Table 7. *Input-Output Boundedness for Classes of Tree Transductions*

MAC :	output-*height*	*exponentially*	bounded by input-*height*
MAC_{nc} :	output-*size*	*exponentially*	bounded by input-*height*
MAC_{wsu} :	output-*height*	*linearly*	bounded by input-*size*
MAC_{su} :	output-*size*	*linearly*	bounded by input-*size*
TOP :	output-*height*	*linearly*	bounded by input-*height*

Construction 9.3 (*counting symbols in a tree using a weakly single-use mtt*)
Let Δ be a ranked alphabet. We construct the weakly single-use mtt $M_\Delta = (\{count^{(2)}\}, \Delta, Nat, count(x, zero), R_\Delta)$, where for every $\delta \in \Delta^{(p)}$ the set R_Δ contains the following rule:

$$count(\delta(u_1, \ldots, u_p), y_1) \to succ(count(u_1, count(u_2, \ldots count(u_p, y_1) \ldots))).$$

By a straightforward induction—to be found in (Voigtländer, 2001)—we can show that for every $t \in T_\Delta$: $height(\tau(M_\Delta)(t)) = size(t)$. □

Example 9.4
For the ranked alphabet Δ_{bin}, we obtain the mtt M_{count} in Example 3.8. ◇

Theorem 9.5 (*exponential size-height bound for non-copying mtts*)
Let $M_{nc} = (F, \Sigma, \Delta, e, R)$ be a non-copying mtt. There exists a constant $c \in \mathbb{N}$ such that $size(\tau(M_{nc})(t)) \leq c^{height(t)}$ for every $t \in T_\Sigma$.

Proof
Since M_{nc} is non-copying and M_Δ is weakly single-use, there exists—by Construction 5.1 and Theorem 5.2—an mtt M such that $\tau(M) = \tau(M_{nc}); \tau(M_\Delta)$. By Lemma 9.1 there exists a constant $c \in \mathbb{N}$ such that $height(\tau(M)(t)) \leq c^{height(t)}$ for every $t \in T_\Sigma$. Furthermore, $height(\tau(M)(t)) = height(\tau(M_\Delta)(\tau(M_{nc})(t))) = size(\tau(M_{nc})(t))$ by the statement in Construction 9.3. This proves the claim. □

Thus, we obtain the summary of input-output boundedness for classes of tree transductions shown in Table 7. The results for TOP and MAC have been proven by Engelfriet & Vogler (1985). The bound for MAC_{wsu} follows from Theorem 7.1 of (Kühnemann, 1998) and the corresponding boundedness property for attributed tree transducers (Fülöp, 1981). The result for MAC_{su} is a consequence of Theorem 7.2 of (Kühnemann, 1998) and a boundedness property for single-use attributed tree transducers (Giegerich, 1988; Kühnemann, 1997).

As pointed out by one referee, these results can be used to prove that MAC_{nc} and MAC_{wsu} are incomparable with respect to inclusion, and that MAC_{nc} is not included in ATT.

Theorem 9.6
$MAC_{nc} \not\subseteq MAC_{wsu}$, $MAC_{wsu} \not\subseteq MAC_{nc}$ and $MAC_{nc} \not\subseteq ATT$

Proof
Consider the mtts M_{exp} and M_{bin} from Example 3.8 and the statements in items 3 and 4 of that example.

- $\tau(M_{exp}) \in MAC_{nc} \setminus (MAC_{wsu} \cup ATT)$, because for every $t \in T_{Nat}$ we have $height(\tau(M_{exp})(t)) = 2^{height(t)} = 2^{size(t)-1}$, which is inconsistent with the linear height-size bound for weakly single-use mtts and for attributed tree transducers (Fülöp, 1981).
- $\tau(M_{bin}) \in MAC_{wsu} \setminus MAC_{nc}$, because for every fully balanced binary tree $t \in T_{\Delta_{bin}}$ we have $size(\tau(M_{bin})(t)) = 2^{(2^{height(t)}+1)} - 1$, which is inconsistent with the exponential size-height bound for non-copying mtts. □

10 Conclusion

We have presented a direct construction that composes a non-copying mtt and a weakly single-use mtt, which are special functional programs. Thus, we have broadened the applicability of a technique—first proposed by Kühnemann (1997; 1998)—for eliminating intermediate data structures in functional programs, including those built up in accumulating parameters.

Central to understanding under what conditions mtts can be composed, was the question **Q** raised in Subsection 4.5. We think that this question is also the key to relax further the restrictions needed to compose two mtts or to find other restrictions that enable such a construction. In particular, we believe that a direct composition construction for the setting of attributed-like mtts (Fülöp & Vogler, 1999) can be given based on a similar idea as the one exploited in the main construction of the present paper.

Also, we consider it fruitful to further investigate composition constructions for macro attributed tree transducers (Kühnemann & Vogler, 1994). These could deliver interesting transformation techniques for higher-order functional programs. The basic idea is that—under appropriate restrictions—functions that use context parameters of higher-order type can be transformed into macro attributed tree transducers, similarly to direct translations of functions with context parameters of first-order type into attributed tree transducers (Courcelle & Franchi-Zannettacci, 1982; Höff, 1998; Engelfriet & Maneth, 1999). Then, decomposition results of Kühnemann (1998) can be applied, thus *introducing* new intermediate results, but creating new opportunities for compositions, which altogether can still lead to an optimisation.

Acknowledgements

We have benefited from many useful suggestions and comments by the anonymous referees that helped to improve the paper a lot. In particular, we would like to thank one referee who made insightful proposals for adding more formal details to the proof appendix (Voigtländer & Kühnemann, 2003). We also thank Andreas Maletti for providing an implementation of the composition construction in the Haskell$^+$ system.

References

Baader, F., & Nipkow, T. (1998). *Term Rewriting and All That.* Cambridge University Press.

Bird, R.S., & de Moor, O. (1997). *Algebra of Programming.* International Series in Computer Science. Prentice Hall.

Burstall, R.M., & Darlington, J. (1977). A transformation system for developing recursive programs. *J. ACM*, **24**, 44–67.

Chin, W.N. (1994). Safe fusion of functional expressions II: Further improvements. *J. Funct. Prog.*, **4**, 515–555.

Correnson, L., Duris, E., Parigot, D., & Roussel, G. (1998). *Symbolic composition.* Tech. rept. 3348. Unité de recherche INRIA Rocquencourt.

Correnson, L., Duris, E., Parigot, D., & Roussel, G. (1999). Declarative program transformation: A deforestation case-study. *Pages 360–377 of: Principles and Practice of Declarative Programming, Paris, France, Proceedings.* LNCS, vol. 1702. Springer-Verlag.

Courcelle, B., & Franchi-Zannettacci, P. (1982). Attribute grammars and recursive program schemes. *Theoret. Comput. Sci.*, **17**, 163–191, 235–257.

Dershowitz, N., & Jouannaud, J.P. (1990). Rewrite systems. *Chap. 6, pages 243–320 of:* van Leeuwen, J. (ed), *Handbook of Theoretical Computer Science*, vol. B. Elsevier Science Publishers B.V.

Engelfriet, J. (1975). Bottom-up and top-down tree transformations — a comparison. *Math. Syst. Theory*, **9**, 198–231.

Engelfriet, J. (1980). Some open questions and recent results on tree transducers and tree languages. *Pages 241–286 of:* Book, R.V. (ed), *Formal language theory; perspectives and open problems.* Academic Press.

Engelfriet, J. (1981). *Tree transducers and syntax directed semantics.* Tech. rept. 363. Technische Hogeschool Twente.

Engelfriet, J., & Maneth, S. (1999). Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inform. and Comput.*, **154**, 34–91.

Engelfriet, J., & Vogler, H. (1985). Macro tree transducers. *J. Comput. Syst. Sci.*, **31**, 71–145.

Franchi-Zannettacci, P. (1982). *Attributs sémantiques et schémas de programmes.* Ph.D. thesis, Université de Bordeaux I.

Fülöp, Z. (1981). On attributed tree transducers. *Acta Cybernetica*, **5**, 261–279.

Fülöp, Z., & Vogler, H. (1998). *Syntax-Directed Semantics — Formal Models Based on Tree Transducers.* Monographs in Theoretical Computer Science, An EATCS Series. Springer-Verlag.

Fülöp, Z., & Vogler, H. (1999). A characterization of attributed tree transformations by a subclass of macro tree transducers. *Theory of Comput. Syst.*, **32**, 649–676.

Ganzinger, H. (1983). Increasing modularity and language-independency in automatically generated compilers. *Sci. of Comput. Prog.*, **3**, 223–278.

Ganzinger, H., & Giegerich, R. (1984). Attribute coupled grammars. *Pages 157–170 of: Symposium on Compiler Construction, Montreal, Canada, Proceedings.* SIGPLAN Notices, vol. 19. ACM Press.

Giegerich, R. (1988). Composition and evaluation of attribute coupled grammars. *Acta Informatica*, **25**, 355–423.

Gill, A. (1996). *Cheap deforestation for non-strict functional languages.* Ph.D. thesis, University of Glasgow.

Gill, A., Launchbury, J., & Peyton Jones, S.L. (1993). A short cut to deforestation. *Pages*

223–232 of: Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, Proceedings. ACM Press.

Hamilton, G.W., & Jones, S.B. (1992). Extending deforestation for first order functional programs. *Pages 134–145 of: 1991 Glasgow Workshop on Functional Programming, Portree, Scotland, Proceedings.* Series of Workshops in Computing. Springer-Verlag.

Höff, M. (1998). *Vergleich der Berechnungsstärken von Klassen eingeschränkter Tree Transducer.* Students project, Dresden University of Technology.

Höff, M. (1999). *Vergleich von Verfahren zur Elimination von Zwischenergebnissen bei funktionalen Programmen.* M.Sc. thesis, Dresden University of Technology.

Höff, M., Vater, R., Maletti, A., Kühnemann, A., & Voigtländer, J. (2001). *Tree transducer based program transformations for Haskell$^+$.* Progress report, Dresden University of Technology.

Hu, Z., Iwasaki, H., & Takeichi, M. (1996). Deriving structural hylomorphisms from recursive definitions. *Pages 73–82 of: International Conference on Functional Programming, Philadelphia, Pennsylvania, Proceedings.* SIGPLAN Notices, vol. 31. ACM Press.

Johann, P. (2001). Short cut fusion: Proved and improved. *Pages 47–71 of: Semantics, Applications, and Implementation of Program Generation, Florence, Italy, Proceedings.* LNCS, vol. 2196. Springer-Verlag.

Jürgensen, C., & Vogler, H. (2001). *Syntactic composition of top-down tree transducers is short cut fusion.* Tech. rept. TUD-FI01-10. Dresden University of Technology.

Kakehi, K., Glück, R., & Futamura, Y. (2001). On deforesting parameters of accumulating maps. *Pages 46–56 of: Logic Based Program Synthesis and Transformation, Paphos, Cyprus, Proceedings.* LNCS, vol. 2372. Springer-Verlag.

Knuth, D.E. (1968). Semantics of context-free languages. *Math. Syst. Theory*, **2**, 127–145. Corrections *Ibid.*, **5**, 95–96 (1971).

Kühnemann, A. (1997). *Berechnungsstärken von Teilklassen primitiv-rekursiver Programmschemata.* Ph.D. thesis, Dresden University of Technology.

Kühnemann, A. (1998). Benefits of tree transducers for optimizing functional programs. *Pages 146–157 of: Foundations of Software Technology* & *Theoretical Computer Science, Chennai, India, Proceedings.* LNCS, vol. 1530. Springer-Verlag.

Kühnemann, A. (1999). Comparison of deforestation techniques for functional programs and for tree transducers. *Pages 114–130 of: Functional and Logic Programming, Tsukuba, Japan, Proceedings.* LNCS, vol. 1722. Springer-Verlag.

Kühnemann, A., & Vogler, H. (1994). Synthesized and inherited functions — a new computational model for syntax-directed semantics. *Acta Informatica*, **31**, 431–477.

Kühnemann, A., & Voigtländer, J. (2001). *Tree transducer composition as deforestation method for functional programs.* Tech. rept. TUD-FI01-07. Dresden University of Technology.

Lescher, C. (1999). *Entwurf und Implementierung einer Eingabesprache für ein System zur Erzeugung syntaxgesteuerter Editoren.* Students project, Dresden University of Technology.

Malcolm, G. (1989). Homomorphisms and promotability. *Pages 335–347 of: Mathematics of Program Construction, Groningen, The Netherlands, Proceedings.* LNCS, vol. 375. Springer-Verlag.

Meijer, E., Fokkinga, M., & Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Pages 124–144 of: Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, Proceedings.* LNCS, vol. 523. Springer-Verlag.

Peyton Jones, S.L., Tolmach, A., & Hoare, T. (2001). Playing by the rules: Rewriting

as a practical optimisation technique in GHC. *Pages 203–233 of: Haskell Workshop, Florence, Italy, Proceedings.*

Reuther, S. (2002). *Adding a tree transducer recognition/transformation pass to the Glasgow Haskell Compiler.* Students project, Dresden University of Technology.

Rounds, W.C. (1970). Mappings and grammars on trees. *Math. Syst. Theory*, **4**, 257–287.

Secher, J.P., & Sørensen, M.H. (1999). On perfect supercompilation. *Pages 113–127 of: Perspectives of System Informatics, Novosibirsk, Russia, Proceedings.* LNCS, vol. 1755. Springer-Verlag.

Sheard, T., & Fegaras, L. (1993). A fold for all seasons. *Pages 233–242 of: Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, Proceedings.* ACM Press.

Sørensen, M.H., Glück, R., & Jones, N.D. (1996). A positive supercompiler. *J. Funct. Prog.*, **6**, 811–838.

Svenningsson, J. (2002). Shortcut fusion for accumulating parameters & zip-like functions. *Pages 124–132 of: International Conference on Functional Programming, Pittsburgh, Pennsylvania, Proceedings.* SIGPLAN Notices, vol. 37. ACM Press.

Thatcher, J.W. (1970). Generalized^2 sequential machine maps. *J. Comput. Syst. Sci.*, **4**, 339–367.

Turchin, V.F. (1986). The concept of a supercompiler. *ACM Trans. on Prog. Lang. and Systems*, **8**, 292–325.

Voigtländer, J. (2001). *Composition of restricted macro tree transducers.* M.Sc. thesis, Dresden University of Technology.

Voigtländer, J. (2002a). Conditions for efficiency improvement by tree transducer composition. *Pages 222–236 of: Rewriting Techniques and Applications, Copenhagen, Denmark, Proceedings.* LNCS, vol. 2378. Springer-Verlag.

Voigtländer, J. (2002b). Using circular programs to deforest in accumulating parameters. *Pages 126–137 of: Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Aizu, Japan, Proceedings.* ACM Press.

Voigtländer, J., & Kühnemann, A. (2001). *Composition of functions with accumulating parameters.* Tech. rept. TUD-FI01-08. Dresden University of Technology.

Voigtländer, J., & Kühnemann, A. (2003). *Proof appendix: Composition of functions with accumulating parameters.*
Available at: `http://wwwtcs.inf.tu-dresden.de/~voigt/JFP-appendix.pdf`.

Wadler, P. (1990). Deforestation: Transforming programs to eliminate trees. *Theoret. Comput. Sci.*, **73**, 231–248.