Comparison of Deforestation Techniques for Functional Programs and for Tree Transducers

Armin Kühnemann

Grundlagen der Programmierung, Institut für Softwaretechnik I,
Fakultät Informatik, Technische Universität Dresden, D–01062 Dresden, Germany,
e–mail: kuehne@orchid.inf.tu–dresden.de

Abstract. We compare transformations for the elimination of intermediate results in first-order functional programs. We choose the well known deforestation technique of Wadler and composition techniques from the theory of tree transducers, of which the implementation of functional programs yet does not take advantage. We identify syntactic classes of function definitions for which both techniques deliver equally efficient results and for which one technique is more powerful than the other. In particular, this paper offers a technique that eliminates intermediate results for certain kinds of function definitions, for which deforestation fails.

1 Introduction

Functional programs frequently use compositions of functions, where functions produce intermediate results, which are consumed by other functions. On the one hand this modular style of programming simplifies the design and the verification of programs [13]. On the other hand the production and consumption of intermediate results can cause inefficiencies, in particular, if the intermediate results are structured objects like lists or trees.

There are several techniques for transforming programs which use intermediate results into programs which do not. We compare *deforestation* [20, 5], which is a well known optimization technique for functional programs, with some *composition techniques* from the theory of tree transducers [7]. The comparison is restricted to the twofold composition of first-order functions, which are defined by an extended scheme of primitive recursion, in which additionally simultaneous definitions of functions and nesting of terms in parameter positions are allowed. This scheme is called *macro tree transducer* [3] (for short *mtt*; cf. also [2, 4]).

An mtt m translates trees over a ranked alphabet of input symbols into trees over a ranked alphabet of output symbols. For this translation process, m uses a ranked alphabet of functions which have at least rank 1, and a set of equations. Every function f is defined by a case analysis on the root symbol c of its first argument t. The right-hand side of the equation for f and c may contain recursive function calls, where the first argument of a function call is a variable that refers to a subtree of t. If every function of m has rank 1, then m is called *top-down tree transducer* [17, 19] (for short *tdtt*). Throughout this paper we will use the following mtts m_{app}, m_{rev}, m_{mir}, and m_{ex} as examples, where m_{ex} is a tdtt.

$$
\begin{array}{llll}
m_{app}\colon & app\ (A\ x_1)\ y_1 = A\ (app\ x_1\ y_1) & m_{mir}\colon & mir\ (A\ x_1)\ y_1 = A\ (mir\ x_1\ (A\ y_1)) \\
& app\ (B\ x_1)\ y_1 = B\ (app\ x_1\ y_1) & & mir\ (B\ x_1)\ y_1 = B\ (mir\ x_1\ (B\ y_1)) \\
& app\ N\ y_1 = y_1 & & mir\ N\ y_1 = y_1 \\
m_{rev}\colon & rev\ (A\ x_1)\ y_1 = rev\ x_1\ (A\ y_1) & m_{ex}\colon & ex\ (A\ x_1) = B\ (ex\ x_1) \\
& rev\ (B\ x_1)\ y_1 = rev\ x_1\ (B\ y_1) & & ex\ (B\ x_1) = A\ (ex\ x_1) \\
& rev\ N\ y_1 = y_1 & & ex\ N = N
\end{array}
$$

The mtt m_{app} appends two lists containing list elements A and B, where lists are represented by monadic trees. In particular, the empty list is represented by the symbol N with rank 0. In analogy, m_{rev} reverses a list l (starting with an application $rev\ l\ N$) by accumulating the list elements in the second argument of rev. The mtt m_{mir} combines the features of m_{app} and m_{rev} by appending a list l to a list which results from a reversal of l (by starting with an application $mir\ l\ N$). The mtt m_{ex} exchanges list elements A by B and vice versa.

If we evaluate the expression $e = (ex\ (mir\ l_0\ l_1))$, where l_0 and l_1 are lists, then mir produces an intermediate result. In particular, l_0 is traversed *three times* (mir traverses l_0 and ex traverses l_0 and l_0's reversed image). Deforestation can be applied to e and to a program p, which consists of m_{ex} and (a slight syntactic variant of) m_{mir}, and delivers an expression $e_d = (mirex\ l_0\ l_1)$ and a function

$$
\begin{array}{l}
mirex\ (A\ x_1)\ y_1 = B\ (mirex\ x_1\ (A\ y_1)) \\
mirex\ (B\ x_1)\ y_1 = A\ (mirex\ x_1\ (B\ y_1)) \\
mirex\ N\ y_1 = ex\ y_1
\end{array}
$$

For the evaluation of e_d, the list l_0 is traversed only *twice* ($mirex$ traverses l_0 and ex traverses l_0's reversed image).

In [4] it was shown that for the composition of an mtt with a tdtt (and vice versa) a single mtt can be constructed, which performs the same computation without producing an intermediate result. Since e represents a composition of the mtt m_{mir} with the tdtt m_{ex}, we can get a new expression $e_c = (\overline{mirex}\ l_0\ (ex\ l_1))$ and a function

$$
\begin{array}{l}
\overline{mirex}\ (A\ x_1)\ y_1 = B\ (\overline{mirex}\ x_1\ (B\ y_1)) \\
\overline{mirex}\ (B\ x_1)\ y_1 = A\ (\overline{mirex}\ x_1\ (A\ y_1)) \\
\overline{mirex}\ N\ y_1 = y_1
\end{array}
$$

The evaluation of e_c requires only *one* traversal of l_0.

We will show that, like in our example, for the composition of an mtt m_1 with a tdtt m_2 (in this order) deforestation successfully eliminates "intermediate symbols" which occur "outside" of the topmost function calls in right-hand sides of equations of m_1, but fails in eliminating symbols "inside" of these function calls. It turns out that the composition technique is more powerful, since it additionally eliminates the latter symbols. For compositions of tdtts with mtts (in this order), the transformation strategies of the composition technique and of deforestation correspond to each other and thus deliver equally efficient programs.

Besides this introduction, the paper contains six further sections. In Section 2 we fix elementary notions and notations. Section 3 introduces our functional language and mtts. Section 4 and Section 5 present deforestation and the composition techniques, respectively. In Section 6 the techniques are compared. Finally, Section 7 contains further research topics.

2 Preliminaries

We denote the set of natural numbers including 0 by $I\!N$. For every $m \in I\!N$, the set $\{1,\ldots,m\}$ is denoted by $[m]$. We will use the sets $X = \{x_0, x_1, x_2, \ldots\}$, $Y = \{y_0, y_1, y_2, \ldots\}$, $Z = \{z_0, z_1, z_2, \ldots\}$, and $V = X \cup Y \cup Z$ of *recursion variables*, *context variables*, *expression variables*, and *variables*, respectively. For a string w and two lists $w_1, \ldots, w_n$ and $w'_1, \ldots, w'_n$ of strings such that no pair w_i and w_j with $i \neq j$ overlaps in w, we denote by $w[w_1/w'_1, \ldots, w_n/w'_n]$ the string which is obtained from w by substituting every occurrence of w_i in w by w'_i. We abbreviate the *substitution* $[w_1/w'_1, \ldots, w_n/w'_n]$ by $[w_i/w'_i \mid w_i \in \{w_1, \ldots, w_n\}]$ or simply by $[w_i/w'_i]$, if the quantification is clear from the context.

Let $\Rightarrow$ be a binary relation on a set K. Then, $\Rightarrow^n$ and $\Rightarrow^*$ denote the n-fold composition and the transitive, reflexive closure, respectively, of $\Rightarrow$. If $k \Rightarrow^* k'$ for $k, k' \in K$ and if there is no $k'' \in K$ such that $k' \Rightarrow k''$, then k' is called a *normal form of* k *with respect to* $\Rightarrow$, which is denoted by $nf(\Rightarrow, k)$, if it exists and if it is unique. We abbreviate $nf(\Rightarrow, k)$ by $nf(k)$, if $\Rightarrow$ is clear.

A *ranked alphabet* is a pair $(S, rank_S)$ where S is a finite set and $rank_S$ is a mapping which associates with every symbol $s \in S$ a natural number called the *rank* of the symbol. The set of elements of S with rank n is denoted by $S^{(n)}$.

For a ranked alphabet S and a subset V' of V, the set of *trees over* S *indexed by* V', denoted by $T_S(V')$, is the smallest subset $T \subseteq (S \cup V' \cup \{(,)\})^*$ such that

- $V' \cup S^{(0)} \subseteq T$ and
- for every $s \in S^{(n)}$ with $n \geq 1$ and $t_1, \ldots, t_n \in T$: $(s\ t_1 \ldots t_n) \in T$.

$T_S(\emptyset)$ is abbreviated by T_S. Let $t \in T_S(V')$. The set of variables that occur in t is denoted by $var(t)$ and the number of occurrences of symbols from a subset $S' \subseteq S$ in t is denoted by $|t|_{S'}$ or simply by $|t|$, if $S = S'$.

3 Functional Programs and Macro Tree Transducers

We consider a simple first-order functional programming language P as source language for our transformations. Every program $p \in P$ consists of several mtts. For simplicity we choose a unique ranked alphabet C of constructor symbols, which is used to build up input trees and output trees of every mtt in p. This fact and the absence of initial functions differ from mtts in the literature. The functions of an mtt are defined by a case analysis on the first argument (*recursion argument*) via pattern matching, where only flat patterns are allowed. The other arguments are called *context arguments*.

Definition 3.1 Let C and F be ranked alphabets of *constructor symbols* (or *constructors*) and *function symbols* (or *functions*), respectively, such that X, Y, C, and F are pairwise disjoint. We define the classes P, MAC, D, and R of *programs*, *mtts*, *function definitions*, and *right-hand sides*, respectively, by the following grammar. We assume that p, m, d, r, c, and f (also equipped with indices) range over the sets P, MAC, D, R, C, and F, respectively.

$$
\begin{array}{lll}
p & ::= m_1 \dots m_l & \text{(program)} \\
m & ::= d_1 \dots d_h & \text{(macro tree transducer)} \\
d & ::= f\ (c_1\ x_1 \dots x_{k_1})\ y_1 \dots y_n = r_1 & \text{(function definition)} \\
 & \qquad \vdots & \\
 & \quad\ f\ (c_q\ x_1 \dots x_{k_q})\ y_1 \dots y_n = r_q & \\
r & ::= y_i \mid f\ x_i\ r_1 \dots r_n \mid c\ r_1 \dots r_k & \text{(right-hand side)}
\end{array}
$$

The sets of constructors and functions that occur in $p \in P$ are denoted by C_p and F_p, respectively. The set of functions that is defined in $m \in MAC$ is denoted by F_m. In addition to the grammar, the following restrictions have to be fulfilled: For every $i \in [l]$ and $f \in F_{m_i}$, the mtt m_i contains exactly one function definition for f. For every $i, j \in [l]$ with $i \neq j$: $F_{m_i} \cap F_{m_j} = \emptyset$. For every $i \in [l]$, $f \in F_{m_i}^{(n+1)}$, and $c \in C_p^{(k)}$ there is exactly one equation of the form $f\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = rhs(f, c)$, where $rhs(f, c) \in T_{C_p \cup F_{m_i}}(\{x_1, \dots, x_k, y_1, \dots, y_n\})$. □

If $F_m = F_m^{(1)}$ for an mtt m, then m is a tdtt. The classes of tdtts and of mtts which are no tdtts are denoted by TOP and by $MAC - TOP$, respectively. A tree t is called *linear*, if every variable occurs at most once in t. An mtt and a program, respectively, is called *linear*, if the right-hand side of each of its equations is linear. We add an index 1 (and l, respectively) to the class MAC and its subclasses, if mtts with only one function (and linear mtts, respectively) are considered. Note that our classes are syntactic classes and no semantic classes (classes of computed functions), which are studied in the theory of tree transducers [7].

Example 3.2 $m_{ex} \in TOP_{1,l}$ and $m_{app}, m_{rev}, m_{mir} \in (MAC - TOP)_{1,l}$. □

It turns out that deforestation in general translates a program $p \in P$ into a program p', in which in right-hand sides of an mtt m also applications of the form $f\ r_0\ r_1 \dots r_n$ with $f \in (F_{p'} - F_m)$ and without restriction "$r_0 = x_i$ for some x_i" can occur. We denote this target language by P^+ and the set of constructors and functions that occur in $p \in P^+$ by C_p and F_p, respectively.

We fix call-by-name semantics, i.e. for every $p \in P^+$ we use a call-by-name reduction relation $\Rightarrow_p$ on $T_{C_p \cup F_p}$. It can be proved that in contrast to general first-order programs, for every $p \in P$ and $e \in T_{C_p \cup F_p}$ the normal form $nf(\Rightarrow_p, e)$ exists. The proof is based on the result in [4], that for every mtt the corresponding (nondeterministic) reduction relation is terminating (and confluent).

In the framework of mtts, a function f_1 produces an intermediate result which is consumed by another function f_2, iff an application of f_1 occurs in the *first* argument of an application of f_2. Thus, in this paper we would like to optimize the evaluation of expressions of the form $(f_2\ (f_1\ t_0 \dots t_{n_1})\ t_{n_1+1} \dots t_{n_1+n_2})$, where each t_i is a tree over constructor symbols. Since the particular constructor trees are not relevant for the transformations, we abstract them by expression variables and transform expressions of the form $(f_2\ (f_1\ z_0 \dots z_{n_1})\ z_{n_1+1} \dots z_{n_1+n_2})$, which we call *composition expressions*. The transformations will deliver expressions of the form $(f\ z_0\ z_1 \dots z_n)$ or $(f\ z_0\ (f_1\ z_1) \dots (f_n\ z_n))$. All these expressions are special initial expressions for programs, which are defined as follows.

Definition 3.3 Let $p \in P^+$ and let f range over F_p. The set of *initial expressions for* p, denoted by E_p, is defined as follows, where e (also equipped with indices) ranges over E_p:

$e ::= f\ e_0\ e_1 \ldots e_n \mid z_i$ (initial expression for a program) $\square$

Our transformation techniques take elements of the class $\{(p,e) \mid p \in P, e \in E_p\}$ as input. This class contains the following subclasses with *linear* composition expressions, for which we would like to compare the transformation techniques.

Definition 3.4 For every $i \in [2]$ let M_i be the class MAC or one of its subclasses. The *composition class* $(M_1; M_2)$ is the set $\{(p,e) \mid p \in P, e \in E_p$, there are mtts m_1 and m_2 in p with $m_1 \in M_1$, $m_2 \in M_2$, and there are $f_1 \in F_{m_1}^{(n_1+1)}$ and $f_2 \in F_{m_2}^{(n_2+1)}$ such that $e = (f_2\ (f_1\ z_0\ z_1 \ldots z_{n_1})\ z_{n_1+1} \ldots z_{n_1+n_2})\}$. $\square$

Example 3.5 Let p consist of m_{ex} and m_{mir}. Then, $e_1 = (mir\ (ex\ z_0)\ z_1) \in E_p$ and $e_2 = (ex\ (mir\ z_0\ z_1)) \in E_p$. Moreover, $(p, e_1) \in (TOP_{1,l}; (MAC - TOP)_{1,l})$ and $(p, e_2) \in ((MAC - TOP)_{1,l}; TOP_{1,l})$. Let $t_0, t_1 \in T_{C_p}$ and $\sigma = [z_0/t_0, z_1/t_1]$. Then we have $e_1\sigma \Rightarrow_p^{2\cdot|t_0|} nf(e_1\sigma)$ and $e_2\sigma \Rightarrow_p^{3\cdot|t_0|+|t_1|-2} nf(e_2\sigma)$. $\square$

The transformations should preserve the semantics of pairs in $\{(p,e) \mid p \in P, e \in E_p\}$. This is formalized by the following notion of equivalence.

Definition 3.6 For every $i \in [2]$ let $(p_i, e_i) \in \{(p,e) \mid p \in P^+, e \in E_p\}$. The pairs (p_1, e_1) and (p_2, e_2) are called *equivalent*, denoted by $(p_1, e_1) \equiv (p_2, e_2)$, if $C_{p_1} = C_{p_2}$ and if for every substitution $\sigma = [v/t_v \mid v \in var(e_1) \cup var(e_2), t_v \in T_{C_{p_1}}]$,

$$nf(\Rightarrow_{p_1}, e_1\sigma) = nf(\Rightarrow_{p_2}, e_2\sigma).$$ $\square$

4 Deforestation

The (classical) deforestation technique [20, 5] can be seen as algorithmic instance of the fold/unfold-technique in [1]. The presentation of deforestation in Transformation 4.1 obeys the syntax of our language P and is similar to e.g. [18]. Deforestation mimics call-by-name reduction steps (unfold-steps) on expressions with variables, i.e. with "values" which are unknown at transformation time. Therefore, deforestation defines new functions by case analysis, whenever it has to handle an unknown value v in the recursion argument of a function application. Roughly speaking, it uses an expansion of v by every possible constructor.

For *treeless* programs [20], in which essentially every occurrence of a function in the right-hand side of an equation is only applied to variables, the termination of deforestation is ensured by using already defined functions, whenever this is possible (fold-steps). The programs in P are not treeless, since mtts may use nesting of terms in parameter positions. To ensure termination of deforestation also on P, we assume that every context argument of a function application is implicitly abstracted by a let-expression [20, 12], i.e. we handle $(f\ x_i\ t_1 \ldots t_n)$ as if it were written like *let* $v_1 = t_1, \ldots, v_n = t_n$ *in* $(f\ x_i\ v_1 \ldots v_n)$. We do not choose explicit let-expressions, since this would expand the description of the composition techniques and the comparison in Sections 5 and 6, respectively.

Transformation 4.1 Let $(p, e) \in (MAC; MAC)$. We define a function $\mathcal{D}$ which takes e (and implicitly p) as input. $\mathcal{D}$ assumes an implicit abstraction with let-expressions and is specialized to handle composition expressions. It can be shown that the five rules for $\mathcal{D}$ perform a complete and disjoint case analysis on the set T of expressions over C_p, F_p, and V, which are encountered starting with a composition expression. In particular, T does not contain expressions of the forms $(f_3\ (f_2\ (f_1 \ldots) \ldots) \ldots)$ and $(f_2\ (f_1\ (c \ldots) \ldots) \ldots)$. The symbols v, c, f, and t (also with indices and apostrophes) range over the sets V, $C_p = \{c_1, \ldots, c_q\}$, F_p, and T, respectively. The notation $\widetilde{f}$ and $\widetilde{f_1 f_2}$ in rules (4) and (5), respectively, refers to a function f and $f_1 f_2$, respectively, which either is constructed as specified behind "where" ("where" does not occur in the transformation result), or which was already constructed. In this way $\mathcal{D}[\![e]\!]$ delivers a new program $p' \in P^+$ and a new initial expression e' for p'. We denote the transformation result (p', e') also by $Def(p, e)$.

$$
\begin{array}{ll}
(1)\ \mathcal{D}[\![v]\!] & = v \\
(2)\ \mathcal{D}[\![c\ t_1 \ldots t_k]\!] & = c\ \ \mathcal{D}[\![t_1]\!] \ldots \mathcal{D}[\![t_k]\!] \\
(3)\ \mathcal{D}[\![f\ (c\ t_1 \ldots t_k)\ t'_1 \ldots t'_n]\!] & = \mathcal{D}[\![rhs(f, c)[x_i/t_i][y_i/t'_i]]\!] \\
(4)\ \mathcal{D}[\![f\ v\ t'_1 \ldots t'_n]\!] & = \widetilde{f}\ v\ \mathcal{D}[\![t'_1]\!] \ldots \mathcal{D}[\![t'_n]\!] \text{ where}
\end{array}
$$

$$
\begin{array}{l}
\widetilde{f}\ (c_1\ x_1 \ldots x_{k_1})\ y_1 \ldots y_n = \mathcal{D}[\![rhs(f, c_1)]\!] \\
\quad \vdots \\
\widetilde{f}\ (c_q\ x_1 \ldots x_{k_q})\ y_1 \ldots y_n = \mathcal{D}[\![rhs(f, c_q)]\!]
\end{array}
$$

$$(5)\ \mathcal{D}[\![f_2\ (f_1\ v\ t_1 \ldots t_{n_1})\ t'_1 \ldots t'_{n_2}]\!] = \widetilde{f_1 f_2}\ v\ \mathcal{D}[\![t_1]\!] \ldots \mathcal{D}[\![t_{n_1}]\!]\ \mathcal{D}[\![t'_1]\!] \ldots \mathcal{D}[\![t'_{n_2}]\!]$$

where

$$
\begin{array}{l}
\widetilde{f_1 f_2}\ (c_1\ x_1 \ldots x_{k_1})\ y_1 \ldots y_{n_1+n_2} = \mathcal{D}[\![f_2\ (rhs(f_1, c_1))\ y_{n_1+1} \ldots y_{n_1+n_2}]\!] \\
\quad \vdots \\
\widetilde{f_1 f_2}\ (c_q\ x_1 \ldots x_{k_q})\ y_1 \ldots y_{n_1+n_2} = \mathcal{D}[\![f_2\ (rhs(f_1, c_q))\ y_{n_1+1} \ldots y_{n_1+n_2}]\!] \quad \square
\end{array}
$$

Lemma 4.2 For every $(p, e) \in (MAC; MAC)$ we have $(p, e) \equiv Def(p, e)$. $\square$

Instead of proving Lemma 4.2 formally, we argue by showing the similarities and differences of Transformation 4.1 and deforestation as it is described e.g. in [18]: $\mathcal{D}$ terminates, since for the composition of two mtts m_1 and m_2 with h_1 and h_2 functions, respectively, it constructs at most $h_1 h_2 + h_1 + h_2$ functions. The rules (4) and (5) reflect the implicit abstraction of context arguments by let-expressions and differ from the usual presentation (cf. e.g. rule (5) in [18]). In the following we show that rule (4) performs the same transformation as transformation rules in the literature, which can handle explicit let-expressions (cf. e.g. [20]). For rule (5) an analogous argument holds.

$$
\begin{array}{lr}
\quad \mathcal{D}[\![f\ v\ t'_1 \ldots t'_n]\!] & \\
= \mathcal{D}[\![let\ v_1 = t'_1, \ldots, v_n = t'_n\ in\ (f\ v\ v_1 \ldots v_n)]\!] & \text{(let-abstraction)} \\
= let\ v_1 = \mathcal{D}[\![t'_1]\!], \ldots, v_n = \mathcal{D}[\![t'_n]\!]\ in\ \mathcal{D}[\![f\ v\ v_1 \ldots v_n]\!] & (\mathcal{D} \text{ on let-expr.)} \\
= let\ v_1 = \mathcal{D}[\![t'_1]\!], \ldots, v_n = \mathcal{D}[\![t'_n]\!]\ in\ (\widetilde{f}\ v\ v_1 \ldots v_n) \text{ where } \ldots & (\mathcal{D} \text{ on function)} \\
= \widetilde{f}\ v\ \mathcal{D}[\![t'_1]\!] \ldots \mathcal{D}[\![t'_n]\!] \text{ where } \ldots & \text{(inlining of let)}
\end{array}
$$

Note that in contrast to e.g. [18] we do not introduce a new function in rule (3) for two reasons: On the one hand mtts do not allow *f-functions* in the sense of [18], which are defined without pattern matching. On the other hand, the introduction of f-functions does not contribute to the elimination of intermediate results. The omission of an f-function in rule (3) can cause a later termination of $\mathcal{D}$, but does not affect termination at all, since at the critical steps, where an expansion of variables is performed, new functions are introduced. Note also that f-functions which are not *postunfolded* later [18], increase the number of reduction steps.

For initial expressions like $(app\ z_0\ (app\ z_1\ z_2))$, which are *no composition expressions*, our deforestation algorithm would behave differently from the algorithm described in [20], since $(app\ z_0\ (app\ z_1\ z_2))$ would be treated as if it were the expression $let\ v = (app\ z_1\ z_2)\ in\ (app\ z_0\ v)$.

Example 4.3 Let $p \in P$ consist of m_{ex} and m_{mir} and let $e_1 = (mir\ (ex\ z_0)\ z_1) \in E_p$. Then, $\mathcal{D}[\![mir\ (ex\ z_0)\ z_1]\!] = (exmir\ z_0\ \mathcal{D}[\![z_1]\!]) = (exmir\ z_0\ z_1)$ where

$$\begin{aligned}
&\underline{exmir\ (A\ x_1)\ y_1}\\
=\ &\mathcal{D}[\![mir\ (rhs(ex,A))\ y_1]\!]\\
=\ &\mathcal{D}[\![mir\ (B\ (ex\ x_1))\ y_1]\!]\\
=\ &\mathcal{D}[\![B\ (mir\ (ex\ x_1)\ (B\ y_1))]\!]\\
=\ &B\ \mathcal{D}[\![mir\ (ex\ x_1)\ (B\ y_1)]\!]\\
=\ &B\ (exmir\ x_1\ \mathcal{D}[\![B\ y_1]\!])\\
=\ &B\ (exmir\ x_1\ (B\ \mathcal{D}[\![y_1]\!]))\\
=\ &\underline{B\ (exmir\ x_1\ (B\ y_1))}
\end{aligned}$$

$$\begin{aligned}
&\underline{exmir\ (B\ x_1)\ y_1}\\
=\ &\ldots = \underline{A\ (exmir\ x_1\ (A\ y_1))}
\end{aligned}$$

$$\begin{aligned}
&\underline{exmir\ N\ y_1}\\
=\ &\mathcal{D}[\![mir\ (rhs(ex,N))\ y_1]\!]\\
=\ &\mathcal{D}[\![mir\ N\ y_1]\!]\\
=\ &\mathcal{D}[\![y_1]\!]\\
=\ &\underline{y_1}
\end{aligned}$$

The new initial expression is $e_{1,d} = (exmir\ z_0\ z_1)$ and the new program $p_{1,d}$ contains the above three equations for the function *exmir* (where both sides of every equation are underlined). Thus, $Def(p, e_1) = (p_{1,d}, e_{1,d})$. Let $t_0, t_1 \in T_{C_p}$ and $\sigma = [z_0/t_0, z_1/t_1]$. We have $e_{1,d}\sigma \Rightarrow^{|t_0|}_{p_{1,d}} nf(e_{1,d}\sigma)$. □

Example 4.4 Let $p \in P$ consist of m_{ex} and m_{mir} and let $e_2 = (ex\ (mir\ z_0\ z_1)) \in E_p$. Then, $\mathcal{D}[\![ex\ (mir\ z_0\ z_1)]\!] = (mirex\ z_0\ \mathcal{D}[\![z_1]\!]) = (mirex\ z_0\ z_1)$ where

$$\begin{aligned}
&\underline{mirex\ (A\ x_1)\ y_1}\\
=\ &\mathcal{D}[\![ex\ (rhs(mir,A))]\!]\\
=\ &\mathcal{D}[\![ex\ (A\ (mir\ x_1\ (A\ y_1)))]\!]\\
=\ &\mathcal{D}[\![B\ (ex\ (mir\ x_1\ (A\ y_1)))]\!] \quad (+)\\
=\ &B\ \mathcal{D}[\![ex\ (mir\ x_1\ (A\ y_1))]\!]\\
=\ &B\ (mirex\ x_1\ \mathcal{D}[\![A\ y_1]\!])\\
=\ &B\ (mirex\ x_1\ (A\ \mathcal{D}[\![y_1]\!])) \quad (-)\\
=\ &\underline{B\ (mirex\ x_1\ (A\ y_1))}
\end{aligned}$$

$$\begin{aligned}
&\underline{mirex\ (B\ x_1)\ y_1}\\
=\ &\ldots = \underline{A\ (mirex\ x_1\ (B\ y_1))}
\end{aligned}$$

$$\begin{aligned}
&\underline{mirex\ N\ y_1}\\
=\ &\mathcal{D}[\![ex\ (rhs(mir,N))]\!]\\
=\ &\mathcal{D}[\![ex\ y_1]\!]\\
=\ &\underline{ex\ y_1}
\end{aligned}$$

where *ex* is defined as in m_{ex}.

The symbols $(+)$ and $(-)$ are explained later. The new initial expression is $e_{2,d} = (mirex\ z_0\ z_1)$ and the new program $p_{2,d}$ contains the above three equations for the function *mirex* and the equations from m_{ex}. Thus, $Def(p, e_2) = (p_{2,d}, e_{2,d})$. Note that $p_{2,d} \in (P^+ - P)$, because of the equation $mirex\ N\ y_1 = ex\ y_1$. Let $t_0, t_1 \in T_{C_p}$ and $\sigma = [z_0/t_0, z_1/t_1]$. We have $e_{2,d}\sigma \Rightarrow^{2\cdot|t_0|+|t_1|-1}_{p_{2,d}} nf(e_{2,d}\sigma)$. □

5 Composition Techniques

In [17] it was shown that the composition of two tdtts can be simulated by only one tdtt. This result was generalized in [4], where composition techniques are presented which construct an mtt for the composition of a tdtt with an mtt, and of an mtt with a tdtt, respectively. In [4] it was also proved that there is no such construction for the composition of two mtts, since the class of functions which are computed by mtts is not closed under composition. The central idea of the composition techniques for the composition of two tree transducers m_1 and m_2 is the observation that, roughly speaking, intermediate results are built up from right-hand sides of m_1. Thus, instead of translating intermediate results by m_2, right-hand sides of m_1 are translated by m_2 to get the equations of a new tree transducer m. For this purpose, m uses $F_{m_1} \times F_{m_2}$ as function set. In the following, we abbreviate every pair $(f, g) \in F_{m_1} \times F_{m_2}$ by $\overline{fg}$.

The composition technique for the composition class $(TOP; MAC)$ is given by Transformation 5.1. It will be necessary to extend the call-by-name reduction relation to expressions containing variables (they are handled like 0-ary constructors) and to restrict the call-by-name reduction relation to use only equations of a certain mtt m, which will be denoted by $\Rightarrow_m$.

Transformation 5.1 Let $(p, e) \in (TOP; MAC)$. Let m_1 be a tdtt in p, let m_2 be an mtt in p, let $f_1 \in F_{m_1}^{(1)}$ and $f_2 \in F_{m_2}^{(n_2+1)}$, such that $e = (f_2\ (f_1\ z_0)\ z_1 \ldots z_{n_2})$. We construct a new program p' and a new initial expression e' for p' as follows. The transformation result (p', e') is also denoted by $Com(p, e)$.

1. From m_2 we construct an mtt $\bar{m}_2$ which is able to translate right-hand sides of equations of m_1. Note that $\bar{m}_2$ is not part of p'.
   - $\bar{m}_2$ contains the equations of m_2.
   - For every $g \in F_{m_2}^{(n+1)}$ and $f \in F_{m_1}^{(1)}$ we add the following equation to $\bar{m}_2$:
   $$g\ (f\ x_1)\ y_1 \ldots y_n = \overline{fg}\ x_1\ y_1 \ldots y_n$$
   where every $f \in F_{m_1}^{(1)}$ and $\overline{fg}$ with $f \in F_{m_1}^{(1)}$ and $g \in F_{m_2}^{(n+1)}$ is viewed as additional unary and $(n+1)$-ary constructor, respectively.
2. p' is constructed from p by replacing m_1 and m_2 by the following mtt m with $F_m^{(n+1)} = \{\overline{fg} \mid f \in F_{m_1}, g \in F_{m_2}^{(n+1)}\}$: For every $g \in F_{m_2}^{(n+1)}$, $f \in F_{m_1}^{(1)}$, and $c \in C_p^{(k)}$, such that $f\ (c\ x_1 \ldots x_k) = rhs(f, c)$ is an equation in m_1, m contains the equation
   $$\overline{fg}\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = nf(\Rightarrow_{\bar{m}_2}, g\ (rhs(f, c))\ y_1 \ldots y_n)$$
3. $e' = (\overline{f_1 f_2}\ z_0\ z_1 \ldots z_{n_2})$ □

Lemma 5.2 For every $(p, e) \in (TOP; MAC)$ we have $(p, e) \equiv Com(p, e)$. □

We omit the proof and only mention that in [4] another construction was used, which first splits the mtt into a tdtt and a device for handling parameter substitutions, then composes the two tdtts, and finally composes the resulting tdtt with the substitution device. We get the same transformation result in one step by avoiding the explicit splitting and joining of substitution devices.

Example 5.3 Let $p \in P$ consist of m_{ex} and m_{mir} and let $e_1 = (mir\ (ex\ z_0)\ z_1) \in E_p$. The mtt $\bar{m}_{mir}$ is given by the following equations:

$$\begin{array}{llll} mir\ (A\ x_1)\ y_1 = A\ (mir\ x_1\ (A\ y_1)) & & mir\ N\ y_1 & = y_1 \\ mir\ (B\ x_1)\ y_1 = B\ (mir\ x_1\ (B\ y_1)) & & mir\ (ex\ x_1)\ y_1 & = \overline{exmir}\ x_1\ y_1 \end{array}$$

The new program $p_{1,c}$ contains only one mtt with the following equations:

$$\begin{array}{ll} \quad \underline{\overline{exmir}\ (A\ x_1)\ y_1} & \quad \underline{\overline{exmir}\ (B\ x_1)\ y_1} \\ = nf(\Rightarrow_{\bar{m}_{mir}}, mir\ (rhs(ex, A))\ y_1) & = \ldots = \underline{A\ (\overline{exmir}\ x_1\ (A\ y_1))} \\ = nf(\Rightarrow_{\bar{m}_{mir}}, mir\ (B\ (ex\ x_1))\ y_1) & \\ = nf(\Rightarrow_{\bar{m}_{mir}}, B\ (mir\ (ex\ x_1)\ (B\ y_1))) & \quad \underline{\overline{exmir}\ N\ y_1} \\ = \underline{B\ (\overline{exmir}\ x_1\ (B\ y_1))} & = \ldots = \underline{y_1} \end{array}$$

The new initial expression is $e_{1,c} = (\overline{exmir}\ z_0\ z_1)$. Thus, $Com(p, e_1) = (p_{1,c}, e_{1,c})$. Let $t_0, t_1 \in T_{C_p}$ and $\sigma = [z_0/t_0, z_1/t_1]$. We have $e_{1,c}\sigma \Rightarrow_{p_{1,c}}^{|t_0|} nf(e_{1,c}\sigma)$. □

The composition class $(MAC - TOP; TOP)$ is handled by Transformation 5.4. We use the additional observation that for $f_1 \in F_{m_1}$, $f_2 \in F_{m_2}$, and an instance of an initial expression $(f_2\ \ (f_1\ \ z_0\ \ z_1 \ldots z_n))$, the results of the instances of $(f_2\ z_1), \ldots, (f_2\ z_n)$ instead of the instances of $z_1, \ldots, z_n$ will occur in the final output. Thus, the context arguments of the functions of the new mtt represent the translations of the context arguments of the functions of m_1 by m_2.

Transformation 5.4 Let $(p, e) \in (MAC - TOP; TOP)$. Let m_1 be an mtt in p, let m_2 be a tdtt in p, let $f_1 \in F_{m_1}^{(n_1+1)}$ and $f_2 \in F_{m_2}^{(1)}$, such that $e = (f_2\ (f_1\ z_0\ z_1 \ldots z_{n_1}))$. Let $g_1, \ldots, g_h$ be a fixed order of all functions in F_{m_2}. We construct a new program p' and a new initial expression e' for p' as follows. The transformation result (p', e') is also denoted by $Com(p, e)$.

1. From m_2 we construct a tdtt $\bar{m}_2$ which is able to translate right-hand sides of equations of m_1. Note that $\bar{m}_2$ is not part of p'.
   - $\bar{m}_2$ contains the equations of m_2.
   - For every $g \in F_{m_2}^{(1)}$ and $f \in F_{m_1}^{(n+1)}$ we add the following equation to $\bar{m}_2$:
     $(*_1)\qquad g\ (f\ x_1\ x_2 \ldots x_{n+1})$
     $\qquad\qquad = \overline{fg}\ x_1\ (g_1\ x_2) \ldots (g_h\ x_2) \ldots\ldots (g_1\ x_{n+1}) \ldots (g_h\ x_{n+1})$
     where every $f \in F_{m_1}^{(n+1)}$ and $\overline{fg}$ with $f \in F_{m_1}^{(n+1)}$ and $g \in F_{m_2}^{(1)}$ is viewed as additional $(n+1)$-ary and $(hn+1)$-ary constructor, respectively.
   - For every $g \in F_{m_2}^{(1)}$ and for every y_j which occurs in right-hand sides of equations of m_1 add the following equation to $\bar{m}_2$:
     $(*_2)\qquad g\ y_j = y_{j,g}$
     where every y_j and every $y_{j,g}$ are viewed as additional 0-ary constructors.
2. p' is constructed from p by keeping m_2 and by replacing m_1 by the following mtt m with $F_m^{(hn+1)} = \{\overline{fg} \mid f \in F_{m_1}^{(n+1)}, g \in F_{m_2}\}$: For every $g \in F_{m_2}^{(1)}$, $f \in F_{m_1}^{(n+1)}$, and $c \in C_p^{(k)}$ such that $f\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = rhs(f, c)$ is an equation in m_1, m contains the equation
   $$\overline{fg}\ (c\ x_1 \ldots x_k)\ y_{1,g_1} \ldots y_{1,g_h} \ldots\ldots y_{n,g_1} \ldots y_{n,g_h} = nf(\Rightarrow_{\bar{m}_2}, g\ (rhs(f, c)))$$
3. $e' = (\overline{f_1 f_2}\ z_0\ (g_1\ z_1) \ldots (g_h\ z_1) \ldots\ldots (g_1\ z_{n_1}) \ldots (g_h\ z_{n_1}))$ □

Note that e' is no composition expression. The following lemma is a consequence of a result in [4] and will not be proved here:

Lemma 5.5 For every $(p,e) \in (MAC{-}TOP; TOP)$ we have $(p,e) \equiv Com(p,e)$.$\square$

Example 5.6 Let $p \in P$ consist of m_{ex} and m_{mir} and let $e_2 = (ex\ (mir\ z_0\ z_1))$ $\in E_p$. The tdtt $\bar{m}_{ex}$ is given by the following equations:

$$\begin{array}{lll} ex\ (A\ x_1) = B\ (ex\ x_1) & ex\ N = N & ex\ (mir\ x_1\ x_2) = \overline{mirex}\ x_1\ (ex\ x_2) \\ ex\ (B\ x_1) = A\ (ex\ x_1) & ex\ y_1 = y_{1,ex} & \end{array}$$

The new program $p_{2,c}$ contains m_{ex} and a new mtt with the following equations:

$$\begin{array}{ll} \quad \overline{mirex}\ (A\ x_1)\ y_{1,ex} & \quad \overline{mirex}\ (B\ x_1)\ y_{1,ex} \\ = nf(\Rightarrow_{\bar{m}_{ex}}, ex\ (rhs(mir, A))) & = \ldots \\ = nf(\Rightarrow_{\bar{m}_{ex}}, ex\ (A\ (mir\ x_1\ (A\ y_1)))) & = \underline{A\ (\overline{mirex}\ x_1\ (A\ y_{1,ex}))} \\ = nf(\Rightarrow_{\bar{m}_{ex}}, B\ (ex\ (mir\ x_1\ (A\ y_1)))) & \\ = nf(\Rightarrow_{\bar{m}_{ex}}, B\ (\overline{mirex}\ x_1\ (ex\ (A\ y_1))))\quad (\,!\,) & \quad \overline{mirex}\ N\ y_{1,ex} \\ = nf(\Rightarrow_{\bar{m}_{ex}}, B\ (\overline{mirex}\ x_1\ (B\ (ex\ y_1)))) & = \ldots \\ = \underline{B\ (\overline{mirex}\ x_1\ (B\ y_{1,ex}))} & = \underline{y_{1,ex}} \end{array}$$

The new initial expression is $e_{2,c} = (\overline{mirex}\ z_0\ (ex\ z_1))$. Thus, $Com(p, e_2) = (p_{2,c}, e_{2,c})$. Let $t_0, t_1 \in T_{C_p}$ and $\sigma = [z_0/t_0, z_1/t_1]$. We have $e_{2,c}\sigma \Rightarrow_{p_{2,c}}^{|t_0|+|t_1|} nf(e_{2,c}\sigma)$. $\square$

6 Comparison

Although the transformations of Sections 4 and 5 work on nonlinear programs as well, we restrict the following comparison to linear mtts, since nonlinear function definitions can cause a deterioration by deforestation [20] and also by composition. We will use the following notions to compare efficiencies.

Definition 6.1 For every $i \in [2]$ let $(p_i, e_i) \in \{(p,e) \mid p \in P^+, e \in E_p\}$ such that $(p_1, e_1) \equiv (p_2, e_2)$. We call (p_1, e_1) *as efficient as* (*at least as efficient as*, respectively) (p_2, e_2), denoted by $\mathcal{E}(p_1, e_1) = \mathcal{E}(p_2, e_2)$ ($\mathcal{E}(p_1, e_1) \geq \mathcal{E}(p_2, e_2)$, respectively), if for every $\sigma = [v/t_v \mid v \in var(e_1) \cup var(e_2), t_v \in T_{C_{p_1}}]$ and for every $\alpha_{1,\sigma}, \alpha_{2,\sigma} \in I\!N$ with $e_i\sigma \Rightarrow_{p_i}^{\alpha_{i,\sigma}} nf(e_i\sigma)$ for every $i \in [2]$,

$$\alpha_{1,\sigma} = \alpha_{2,\sigma}\ (\alpha_{1,\sigma} \leq \alpha_{2,\sigma}, \text{respectively}).$$

We call (p_1, e_1) *more efficient than* (p_2, e_2), denoted by $\mathcal{E}(p_1, e_1) > \mathcal{E}(p_2, e_2)$, if $\mathcal{E}(p_1, e_1) \geq \mathcal{E}(p_2, e_2)$ and there is $\sigma = [v/t_v \mid v \in var(e_1) \cup var(e_2), t_v \in T_{C_{p_1}}]$ and $\alpha_{1,\sigma}, \alpha_{2,\sigma} \in I\!N$ with $e_i\sigma \Rightarrow_{p_i}^{\alpha_{i,\sigma}} nf(e_i\sigma)$ for every $i \in [2]$ and

$$\alpha_{1,\sigma} < \alpha_{2,\sigma}. \qquad \square$$

We will compare the transformations on the composition class $(MAC_l; MAC_l)$. It turns out that there are different results for its three disjoint subclasses $(TOP_l; MAC_l)$, $((MAC-TOP)_l; TOP_l)$, and $((MAC-TOP)_l; (MAC-TOP)_l)$.

6.1 Where Deforestation and Composition behave similarly

Example 6.2 Let $p \in P$ consist of m_{ex} and m_{mir} and let $e_1 = (mir\ (ex\ z_0)\ z_1)$. Examples 4.3 and 5.3 show that deforestation and the composition technique transform (p, e_1), apart from function renaming, into the same program and initial expression. In particular, $\mathcal{E}(Def(p, e_1)) = \mathcal{E}(Com(p, e_1)) > \mathcal{E}(p, e_1)$. □

This observation can be generalized as follows.

Lemma 6.3 For every $(p, e) \in (TOP_{1,l}; MAC_{1,l})$: $Def(p, e) = Com(p, e)$ (apart from function renaming).

Proof Idea. Let $(p_d, e_d) = Def(p, e)$ and $(p_c, e_c) = Com(p, e)$. Let m_1 be a tdtt in p, m_2 be an mtt in p, $f_1 \in F_{m_1}^{(1)}$, and $f_2 \in F_{m_2}^{(n+1)}$ such that $e = (f_2\ (f_1\ z_0)\ z_1 \dots z_n)$. Let $c \in C_p^{(k)}$. Deforestation generates in the body of the where-clause in rule (5) the equation

$$f_1 f_2\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = \mathcal{D}[\![f_2\ (rhs(f_1, c))\ y_1 \dots y_n]\!]$$

for p_d and composition generates by 2. in Transformation 5.1 the equation

$$\overline{f_1 f_2}\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = nf(\Rightarrow_{\bar{m}_2}, f_2\ (rhs(f_1, c))\ y_1 \dots y_n)$$

for p_c. Identifying $f_1 f_2$ with $\overline{f_1 f_2}$, these two equations are equal, since

$$\mathcal{D}[\![f_2\ r\ r_1 \dots r_n]\!] = nf(\Rightarrow_{\bar{m}_2}, f_2\ r\ r_1 \dots r_n),$$

for every right-hand side r of m_1 and for every right-hand sides $r_1, \dots, r_n$ of m_2 in which recursion variables are instantiated by right-hand sides of m_1, can be proved by an induction on $r \in R$. Additionally, an inner induction proof on the set of in this way instantiated right-hand sides of m_2 has to be performed. □

If mtts with more than one function are considered, then Lemma 6.3 remains in principle valid, but the composition technique sometimes constructs new functions which are never used to evaluate instances of the new initial expression. Deforestation prevents the construction of superfluous functions, since it mimics the call-by-name evaluation of an arbitrarily instantiated initial expression.

Lemma 6.4 For every $(p, e) \in (TOP_l; MAC_l)$: $Def(p, e) = Com(p, e)$ (apart from superfluous functions and apart from function renaming). □

Since deforestation does not deteriorate the efficiency (cf. e.g.[20, 18]), we get:

Corollary 6.5 For every $(p, e) \in (TOP_l; MAC_l)$:

$$\mathcal{E}(Def(p, e)) = \mathcal{E}(Com(p, e)) \geq \mathcal{E}(p, e).$$ □

6.2 Where Composition is at least as good as Deforestation

Example 6.6 Let $p \in P$ consist of m_{ex} and m_{mir} and let $e_2 = (ex\ (mir\ z_0\ z_1))$. We get the result $\mathcal{E}(Com(p, e_2)) > \mathcal{E}(Def(p, e_2)) > \mathcal{E}(p, e_2)$ from Examples 3.5, 4.4, and 5.6.

Example 4.4 shows that on the one hand deforestation is successful in removing the intermediate constructor "outside" the recursive call of mir by replacing A by B (cf. step marked (+) in Example 4.4).

On the other hand, deforestation cannot remove the intermediate constructor "inside" the recursive call of mir (cf. step marked $(-)$ in Example 4.4), since the function ex does not reach the context argument of mir. In contrast to this, the step (!) in Example 5.6 shows, that ex moves into the context argument of mir, thereby removing also the"inside" intermediate constructor. □

The different behaviour of the transformations is caused by the different strategies to handle the "meeting" of two functions. Rule (5) in Transformation 4.1 shows that deforestation generates a new function and simply moves into the context arguments without changing them, whereas equation $(*_1)$ in Transformation 5.4 shows that composition additionally sends the functions of m_2 into the context arguments. Maybe this observation allows an improvement of deforestation by integrating the behaviour of $(*_1)$ into rule (5). The equation $(*_1)$ also has one negative aspect: Since it copies context arguments according to the number of functions of m_2, the produced program can be nonlinear. We do not know yet, how the efficiency (counting call-by-need reduction steps) of programs, which are constructed in this way, is related to the original programs. Hence, we restrict the statement of the following lemma to tdtts with only one function.

Theorem 6.7 For every $(p, e) \in ((MAC - TOP)_l; TOP_{1,l})$:

$$\mathcal{E}(Com(p,e)) \geq \mathcal{E}(Def(p,e)) \geq \mathcal{E}(p,e).$$ □

Before we give the proof idea, we would like to discuss, for which composition classes we obtain strict improvements in Theorem 6.7.

Example 6.8 Let $p \in P$ consist of m_{ex}, m_{app}, and m_{rev}.
- For $e = (ex\ (app\ z_0\ z_1))$ we have $\mathcal{E}(Com(p,e)) = \mathcal{E}(Def(p,e)) > \mathcal{E}(p,e)$, since already deforestation completely removes the intermediate result.
- For $e = (ex\ (rev\ z_0\ z_1))$ we have $\mathcal{E}(Com(p,e)) > \mathcal{E}(Def(p,e)) = \mathcal{E}(p,e)$, since deforestation fails in removing the intermediate result, whereas composition completely removes it. □

The different results in Example 6.8 are caused by the different structures of right-hand sides of app and rev. For example, $rhs(app, A)$ has a constructor outside and no constructor inside the recursive function call, whereas in $rhs(rev, A)$ it is vice versa. Roughly speaking, deforestation can only eliminate intermediate constructors, which occur outside the topmost function applications of right-hand sides, whereas composition additionally eliminates intermediate constructors inside the topmost function applications. Following this idea, we define a decomposition of right-hand sides into a top-part and bottom-parts, which will be important to state a refinement of Theorem 6.7 and to prove Theorem 6.7.

Definition 6.9 Let $r \in R$. Define $top(r) \in T_C(y_1, \ldots, y_n, u_1, \ldots, u_m)$, for every $i \in [m]$ define $f_{k_i} \in F^{(n_i+1)}$ and $x_{k_i} \in X$, and for every $j \in [n_i]$ define $bot_{i,j}(r) \in R$, such that $r = top(r)[u_i/(f_{k_i}\ x_{k_i}\ bot_{i,1}(r) \ldots bot_{i,n_i}(r))]$. □

Example 6.10 If $r = (A\ (mir\ x_1\ (A\ y_1)))$, then $top(r) = (A\ u_1)$, $f_{k_1} = mir$, $x_{k_1} = x_1$, and $bot_{1,1}(r) = (A\ y_1)$. Thus, $|top(r)|_C = 1$ and $|bot_{1,1}(r)|_C = 1$. □

Theorem 6.11 Let $(p, e) \in ((MAC - TOP)_l; TOP_{1,l})$. Let m_1 be an mtt in p, m_2 be a tdtt in p, $f_1 \in F_{m_1}^{(n_1+1)}$, and $f_2 \in F_{m_2}$, such that $e = (f_2\ (f_1\ z_0 \ldots z_{n_1}))$.

- If there is $c \in C_p$ with $|top(rhs(f_1, c))|_{C_p} > 0$, then
$$\mathcal{E}(Def(p, e)) > \mathcal{E}(p, e).$$
- If there are $c \in C_p$, and $i, j > 0$ with $|bot_{i,j}(rhs(f_1, c))|_{C_p} > 0$, and if $var(l) = var(r)$ for every equation $l = r$ of m_1 and m_2 (nondeleting condition), then
$$\mathcal{E}(Com(p, e)) > \mathcal{E}(Def(p, e)). \qquad \square$$

Proof Idea of Theorems 6.7 and 6.11. Let $(p, e) \in ((MAC-TOP)_l; TOP_{1,l})$, $(p_d, e_d) = Def(p, e)$, and $(p_c, e_c) = Com(p, e)$. Let m_1 be an mtt in p, m_2 be a tdtt in p, $f_1 \in F_{m_1}^{(n_1+1)}$, $g \in F_{m_2}^{(1)}$ such that $e = (g\ (f_1\ z_0 \ldots z_{n_1}))$.

Let $f \in F_{m_1}^{(n+1)}$ and $c \in C_p^{(k)}$. We abbreviate $top(rhs(f, c))$ by top and $bot_{i,j}(rhs(f, c))$ by $bot_{i,j}$. Thus, $rhs(f, c) = top[u_i/(f_{k_i}\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i})]$. Deforestation generates (if demanded) the following equation for p_d:

$$\begin{aligned}
&\underline{fg\ (c\, x_1 \ldots x_k)\ y_1 \ldots y_n} \\
&= \mathcal{D}[\![g\ (top[u_i/(f_{k_i}\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i})])]\!] && \text{(Body of where in (5))} \\
&= nf(\Rightarrow_{m_2}, g\ top)[(g\ y_j)/\mathcal{D}[\![g\ y_j]\!]][(g\ u_i)/\mathcal{D}[\![g\ (f_{k_i}\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i})]\!]] \\
& && \text{(Rules (2) and (3))} \\
&= nf(\Rightarrow_{m_2}, g\ top)[(g\ y_j)/(g\ y_j)][(g\ u_i)/\mathcal{D}[\![g\ (f_{k_i}\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i})]\!]] \\
& && (\mathcal{D}[\![g\ y_j]\!] = (g\ y_j)) \\
&= nf(\Rightarrow_{m_2}, g\ top)[(g\ u_i)/(f_{k_i}g\ x_{k_i}\ \ \mathcal{D}[\![bot_{i,1}]\!] \ldots \mathcal{D}[\![bot_{i,n_i}]\!])] \\
& && \text{(Rule (5))} \\
&= \underline{nf(\Rightarrow_{m_2}, g\ top)[(g\ u_i)/(f_{k_i}g\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i})]} && (\mathcal{D}[\![r]\!] = r \text{ for } r \in R)
\end{aligned}$$

The statements $\mathcal{D}[\![g\ y_j]\!] = (g\ y_j)$ and $\mathcal{D}[\![r]\!] = r$ do not reflect that here functions of m_2 and m_1, respectively, are "reproduced" in general. $\mathcal{D}[\![r]\!] = r$ can be proved by an induction on $r \in R$. Composition generates the following equation for p_c:

$$\begin{aligned}
&\underline{\overline{fg}\ (c\, x_1 \ldots x_k)\ y_{1,g} \ldots y_{n,g}} \\
&= nf(\Rightarrow_{\bar{m}_2}, g\ (top[u_i/(f_{k_i}\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i})])) && \text{(2. in Transformation 5.4)} \\
&= nf(\Rightarrow_{\bar{m}_2}, g\ top)[(g\ u_i)/nf(\Rightarrow_{\bar{m}_2}, g\ (f_{k_i}\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i}))] \\
& && \text{(Definition of } \bar{m}_2\text{)} \\
&= \underline{nf(\Rightarrow_{m_2}, g\ top)[(g\ y_j)/y_{j,g}]} \\
&\quad \underline{[(g\ u_i)/(\overline{f_{k_i}g}\ x_{k_i}\ \ nf(\Rightarrow_{\bar{m}_2}, g\ bot_{i,1}) \ldots nf(\Rightarrow_{\bar{m}_2}, g\ bot_{i,n_i}))]} \\
& && ((*_1) \text{ and } (*_2) \text{ in Tr. 5.4})
\end{aligned}$$

Let $t_1, \ldots, t_k, t'_1, \ldots, t'_n \in T_{C_p}$. Let a reduction of p start with the step

$$g\ (f\ (c\, t_1 \ldots t_k)\ t'_1 \ldots t'_n) \Rightarrow_p g\ (top[u_i/(f_{k_i}\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i})])[x_l/t_l][y_l/t'_l].$$

The continuation of this reduction will in particular force the evaluation of $(g\ top)$ and of some $(g\ bot_{i,j})$. The corresponding first reduction step of p_d is

$$\begin{aligned}
&fg\ (c\, t_1 \ldots t_k)\ t'_1 \ldots t'_n \quad \Rightarrow_{p_d} \\
&nf(\Rightarrow_{m_2}, g\ top)[(g\ y_j)/(g\ t'_j)][(g\ u_i)/((f_{k_i}g\ x_{k_i}\ \ bot_{i,1} \ldots bot_{i,n_i})[x_l/t_l][y_l/t'_l])]
\end{aligned}$$

It saves the steps for the evaluation of $(g\ top)$. The corresponding first reduction step of p_c also saves the steps for the evaluation of the $(g\ bot_{i,j})$:

$\overline{fg}\ (c\ t_1 \dots t_k)\ (g\ t'_1) \dots (g\ t'_n) \quad \Rightarrow_{p_c}$
$nf(\Rightarrow_{m_2}, g\ top)[(g\ y_j)/(g\ t'_j)]$
$\quad [(g\ u_i)/((\overline{f_{k_i} g}\ x_{k_i}\ nf(\Rightarrow_{\bar{m}_2}, g\ bot_{i,1}) \dots nf(\Rightarrow_{\bar{m}_2}, g\ bot_{i,n_i}))[x_l/t_l][y_{l,g}/(g\ t'_l)])]$

This also holds for corresponding instances of e, e_d, and e_c. The informal explanation can be seen as heart of an induction proof on T_{C_p}, for which additionally an inner induction proof on the set of right-hand sides of m_1 is needed. The proof also shows that, if subtrees are deleted, then this affects the three reductions in the same way. If for some $c \in C_p$ we have $|top(rhs(f_1, c))|_{C_p} > 0$ (and $|bot_{i,j}(rhs(f_1, c))|_{C_p} > 0$ for some $i, j > 0$ such that $bot_{i,j}(rhs(f_1, c))$ is not deleted by $\Rightarrow_p$), then the precomputation of $(g\ \ top(rhs(f_1, c)))$ (and of $(g\ \ bot_{i,j}(rhs(f_1, c)))$, respectively) really saves reduction steps. This holds at least for substitutions of z_0 in e by a tree which has the symbol c at its root. □

6.3 Where Deforestation is at least as good as Composition

Since for every $(p, e) \in ((MAC - TOP)_l; (MAC - TOP)_l)$ the composition techniques are not applicable, we define $Com(p, e) = (p, e)$ and get as consequence:

Theorem 6.12 For every $(p, e) \in ((MAC - TOP)_l; (MAC - TOP)_l)$:
$$\mathcal{E}(Def(p, e)) \geq \mathcal{E}(Com(p, e)) = \mathcal{E}(p, e).$$ □

Also here we could distinguish composition classes, which are either improved or not improved by deforestation. Since this leads to an analysis of deforestation and does not contribute to our intended comparison of the two techniques, we only give representative examples for these classes:

Example 6.13 Let $p \in P$ consist of m_{app} and m_{rev}.

- For $e = (app\ (app\ z_0\ z_1)\ z_2)$ we have $\mathcal{E}(Def(p, e)) > \mathcal{E}(p, e)$.
- For $e = (rev\ (rev\ z_0\ z_1)\ z_2)$ we have $\mathcal{E}(Def(p, e)) = \mathcal{E}(p, e)$. □

7 Future Work

We have presented qualitative comparisons of deforestation and composition techniques for certain composition classes. A quantitative analysis (also by using more exact efficiency measures) would be useful, in order to calculate speedups, which are realized by the transformations. Moreover, we would like to extend our research to nonlinear programs and to specify composition classes, for which nonlinearity is harmless with respect to deforestation and composition.

Other methods for the elimination of intermediate results should be integrated into the comparison. Therefore, on the one hand we want to analyze *Short Cut* (or: *Cheap*) *Deforestation* [11]. In [10] already an informal comparison between classical deforestation and cheap deforestation is performed. On the other hand we would like to inspect composition methods for *attributed tree transducers* [6], which are abstractions of *attribute grammars* [14]. Such composition results were presented e.g. in [6, 8, 9]. They can also be used for the composition of a restricted class of mtts [15]. For the intended comparison, we consider *macro attributed tree transducers* [16] as suitable integration formalism.

Acknowledgment

I would like to thank Morten Heine Sørensen and Heiko Vogler for stimulating discussions on deforestation and on tree transducers, respectively.

References

1. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. Assoc. Comput. Mach.*, 24:44–67, 1977.
2. B. Courcelle and P. Franchi–Zannettacci. Attribute grammars and recursive program schemes. *Theoret. Comput. Sci.*, 17:163–191, 235–257, 1982.
3. J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal language theory; perspectives and open problems*, pages 241–286. New York, Academic Press, 1980.
4. J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31:71–145, 1985.
5. A.B. Ferguson and P. Wadler. When will deforestation stop? In *1988 Glasgow Workshop on Functional Programming*, pages 39–56, 1988.
6. Z. Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981.
7. Z. Fülöp and H. Vogler. *Syntax-directed semantics – Formal models based on tree transducers.* Monographs in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 1998.
8. H. Ganzinger. Increasing modularity and language–independency in automatically generated compilers. *Science of Computer Programming*, 3:223–278, 1983.
9. R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.
10. A. Gill. *Cheap deforestation for non-strict functional languages.* PhD thesis, University of Glasgow, 1996.
11. A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *FPCA'93, Copenhagen, Denmark, Proceedings*, pages 223–231. ACM Press, 1993.
12. G. W. Hamilton and S. B. Jones. Extending deforestation for first order functional programs. In *1991 Glasgow Workshop on Functional Programming*, Series of Workshops in Computing, pages 134–145. Springer-Verlag, 1992.
13. J. Hughes. Why functional programming matters. In D. Turner, editor, *Research Topics in Functional Programming.* Addison-Wesley, 1990.
14. D.E. Knuth. Semantics of context–free languages. *Math. Syst. Th.*, 2:127–145, 1968. Corrections in *Math. Syst. Th.*, 5:95-96, 1971.
15. A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In *FST & TCS'98, Chennai, India, Proceedings*, volume 1530 of *LNCS*, pages 146–157. Springer-Verlag, December 1998.
16. A. Kühnemann and H. Vogler. Synthesized and inherited functions – a new computational model for syntax-directed semantics. *Acta Informatica*,31:431–477, 1994.
17. W.C. Rounds. Mappings and grammars on trees. *Math. Syst. Th.*,4:257–287, 1970.
18. M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *ESOP'94, Edinburgh, U.K., Proceedings*, volume 788 of *LNCS*, pages 485–500. Springer–Verlag, 1994.
19. J.W. Thatcher. Generalized^2 sequential machine maps. *J. Comput. Syst. Sci.*, 4:339–367, 1970.
20. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comp. Sci.*, 73:231–248, 1990.