Tree Transducer Composition as Deforestation Method for Functional Programs

Armin Kühnemann Janis Voigtländer*

Department of Computer Science, Dresden University of Technology

D–01062 Dresden, Germany. E-mail: {kuehne,voigt}@tcs.inf.tu-dresden.de

Abstract

We demonstrate that composition techniques for tree transducers are suitable to eliminate intermediate results in functional programs. We consider two composition techniques, which view special functional programs as (restricted) macro tree transducers: The first uses composition techniques for macro tree transducers directly. The second is indirect in that it (i) translates macro tree transducers into attributed tree transducers, (ii) uses a composition technique for attributed tree transducers, and (iii) translates the composition result back into a macro tree transducer. We informally compare these techniques with the deforestation technique of Wadler. In particular we show that the composition techniques eliminate intermediate results for certain kinds of function definitions, for which classical deforestation fails.

1 Introduction

Functional programming languages are very well suited to specify programs in a modular style, which simplifies the design and the verification of programs. Unfortunately, modular programs often have poor time- and space-complexity compared to other (sometimes less understandable) programs, which solve the same tasks. In particular function compositions, in which functions produce intermediate results, which are consumed by other functions, can cause inefficiencies, if the intermediate results are structured objects like lists or trees.

To illustrate this problem, we consider the following example program p (in Haskell notation), which contains the definition of two functions *rev* and *map*. The function *rev* reverses a polymorphic list l that is built up from *Cons*- and *Nil*-constructors (by starting with an initial expression $rev\ l\ Nil$). The list reversal is performed by accumulating the list elements in the second argument of *rev*. The function *map* applies a function f to every element of a list.

$$
\begin{array}{llcll}
rev & (Cons\ a\ x) & y & = & rev\ x\ (Cons\ a\ y) \\
rev & Nil & y & = & y \\[4pt]
map\ f & (Cons\ a\ x) & & = & Cons\ (f\ a)\ (map\ f\ x) \\
map\ f & Nil & & = & Nil
\end{array}
$$

If we compute the expression $e_{mr} = rev\ (map\ f\ l)\ Nil$ for some list l, then l is traversed twice (*map f* traverses l once and *rev* traverses the list obtained thus), although the same task could also be performed with one traversal. There are several techniques for transforming programs, which use intermediate results, into programs, which do not. The well

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

known technique *deforestation* [Wad90], which is an algorithmic instance of the *fold/unfold*-technique [BD77] and which henceforth will be referred to as *classical deforestation*, can be applied to p and e_{mr}[1] and delivers the new expression $e'_{mr} = \overline{maprev}\ f\ l\ Nil$ and the function definition:

$$\begin{array}{llllll} \overline{maprev}\ f & (Cons\ a\ x) & y & = & \overline{maprev}\ f\ x\ (Cons\ (f\ a)\ y) \\ \overline{maprev}\ f & Nil & y & = & y \end{array}$$

If we however consider the program p with the expression $e_{rm} = map\ f\ (rev\ l\ Nil)$, in which we first apply rev and then map (again giving two traversals on l), classical deforestation delivers an expression $e'_{rm} = \overline{revmap}\ f\ l\ Nil$ and the following function $\overline{revmap}$ as transformation result:

$$\begin{array}{llllll} \overline{revmap}\ f & (Cons\ a\ x) & y & = & \overline{revmap}\ f\ x\ (Cons\ a\ y) \\ \overline{revmap}\ f & Nil & y & = & map\ f\ y \end{array}$$

To evaluate the new expression e'_{rm}, again two list-traversals are necessary. The reversed list is still accumulated in the last argument of $\overline{revmap}\ f$ and afterwards traversed by $map\ f$. Thus, classical deforestation was not successful in removing the intermediate list element "inside" the recursive call of rev.

Our paper shows how results from the theory of tree transducers [FV98] can be used to tackle this problem of classical deforestation not being able to reach accumulating arguments [Chi94]: The above program p consists of two units, each of which can be seen as an extended scheme of primitive recursion, which will be called *macro tree transducer* (for short *mtt*) [Eng80, CF82, EV85] in the following. This scheme additionally allows simultaneous definitions of functions and nesting of terms in argument positions.

An mtt m translates trees into trees, where trees are built over a ranked alphabet of constructors. For this translation process, m uses a ranked alphabet of function symbols, which have at least rank one, and a set of equations. Every function f is defined by a case analysis on the root constructor c of its first argument t, called the recursion argument. The other arguments are called context arguments (also: accumulating arguments). 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 one, then m is called *top-down tree transducer* (for short *tdtt*) [Rou70, Tha70, Eng75]. Note that in our example, in order to handle polymorphic lists, we view $\underline{Cons\ a}$ and $\underline{Cons\ (f\ a)}$ as constructors and $\underline{map\ f}$ as a function symbol. Then the function rev (together with the initial expression $rev\ l\ Nil$) is an mtt m_{rev}, and the function $\underline{map\ f}$ (together with the initial expression $\underline{map\ f}\ l$) is a tdtt $m_{\underline{map\ f}}$.

The expression e_{rm}, which could not be optimized by classical deforestation, represents a composition of an mtt with a tdtt. For such compositions a single mtt can be constructed, which calculates the composition of the two functions without producing an intermediate result [EV85]. The central idea is to translate the right-hand sides of the

[1] Although map is a higher-order function, it is amenable to classical deforestation by viewing it as a *higher-order macro* [Wad90], considering $map\ f$ as a first-order function with a free occurrence of f in the right-hand side of its first rule. On the other hand, rev does not meet the requirement (the so called *treeless* condition) to apply classical deforestation directly, but can be handled after a small reformulation with let-expressions [HJ92]. See [Küh99] for a treatment with implicit let-expressions.

mtt by applying rules of the tdtt. In contrast to classical deforestation, this translation reaches the arguments of recursive function calls. In our example we get an expression $e''_{rm} = \overline{revmap}'\ f\ l\ Nil$ and a function definition for $\overline{revmap}'$ that is equivalent to the definition of $\overline{maprev}$ above. The evaluation of e''_{rm} requires only one traversal of l. In this case the transformation result of the composition technique is better than the transformation result of classical deforestation, which for its part is no better than the original expression and program. We claim that for every composition of a linear mtt with a linear tdtt, the composition technique yields at least as efficient results as classical deforestation. This claim was proven in [Küh99] for tdtts with exactly one function.

There is also a composition technique for composing a tdtt with an mtt (in this order) [Eng81]. For example, the expression e_{mr} represents such a composition. For compositions of tdtts with mtts, the transformation strategies of the composition technique and of classical deforestation correspond to each other and thus deliver equally efficient results [Küh99].

However, if we consider the composition of two mtts, none of which is a tdtt, e.g., $e_{rr} = rev\ (rev\ l\ Nil)\ Nil$, there is no general composition technique, because the class of functions, which can be realized by mtts, is not closed under composition [EV85]. In this example classical deforestation is applicable, but does not eliminate the intermediate result, due to the same problem of how to deforest in accumulating arguments. Fortunately, expression e_{rr} can be optimized, because rev belongs to a restricted class of mtts, for which the theory of tree transducers provides a composition result based on *attributed tree transducers* (for short *atts*) [Fül81]. The idea is (i) to translate mtts into atts [CF82, Höf98, EM99], (ii) to use a composition technique for atts [Gan83, GG84, Gie88], and (iii) to translate the composition result back into an mtt [Fra82, FV98].

The concept of att is a formal model for studying properties of attribute grammars [Knu68], which for their part are even considered as functional programming paradigm [Joh87]. Atts are abstractions of attribute grammars in the sense that they take trees over a ranked alphabet of constructors rather than derivation trees as argument, and that the values of the attribute instances of input trees are also constructor trees:
The set of attributes is partitioned into the set of synthesized and inherited attributes, which are associated to every node of an input tree, and which move values into the direction of the root and the leaves of the input tree, respectively. Roughly speaking, computing the value of a synthesized attribute instance at a node v of an input tree, the values of the inherited attribute instances at v and of the synthesized attribute instances at its direct descendants (if they exist) may be used and, computing the value of an inherited attribute instance at v, the values of the inherited attribute instances of its direct ancestor (if it exists) and of the synthesized attribute instances at v and at its siblings may be used. For illustration consider the att a_{rev} in Figure 1, which computes the rev-function on the input $(\underline{Cons}\ \underline{1}\ (\underline{Cons}\ \underline{2}\ Nil))$, where for technical reasons a *root*-symbol is put on top of the input tree. The att a_{rev} reverses a list l, by (i) collecting the elements of l in reversed order with an inherited attribute i, and by (ii) moving the reversed list with a synthesized attribute s to the root of l. Note that the arrows in the picture show the intended "information flow". The computed output tree is obtained as value of a designated synthesized attribute s_{in} at the *root*-symbol.

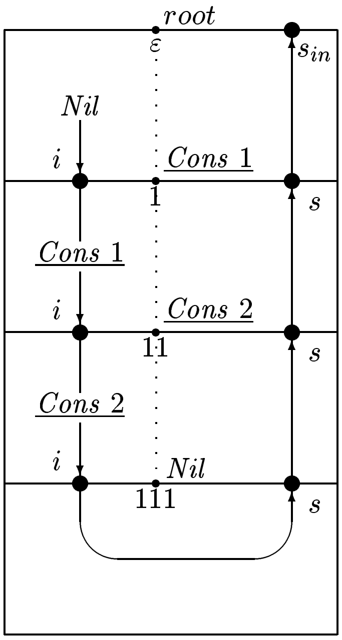

Figure 1: Information flow of a_{rev} on the tree (*root* (Cons 1 (Cons 2 *Nil*))).

As mentioned above, the composition technique for restricted mtts proceeds in three steps, the heart of the transformation being step (ii):

(ii) In [Gan83] it was shown that restricted attribute grammars, which are called language morphisms, are closed under composition. In the framework of attribute coupled grammars this composition technique was presented in [GG84, Gie88]. In the latter paper, the imposed restriction is called *syntactic single use requirement*, which essentially means that every attribute instance in a tree may be used at most once to calculate the values of other attribute instances. The corresponding condition on atts is called the *single-use* restriction [Küh97, Küh98]. Having two atts, the first of which is single-use, we can construct a new att that computes their composition. Again, the central idea is to run the second transducer on the right-hand sides of the first transducer's rules [Fül81]. Often, the resulting att can be optimized by a so called copy rules elimination [CDPR98].

(i) In order to obtain single-use atts from mtts, we have to find an appropriate restriction for mtts[2]. Therefore, we project the single-use restriction on mtts, yielding the *weakly single-use* restriction, because context arguments can still be copied. We additionally take advantage of the *noncopying* restriction for mtts. This restriction means that context arguments of functions may be used at most once in the right-hand sides of function definitions. A weakly single-use and noncopying mtt is called *single-use* and will be transformed into a single-use att. As in [CF82], the transformation of mtts into atts is based on the idea that functions and their context arguments are simulated by synthesized and inherited attributes, respectively. In

[2]The well-presented restriction for mtts and the transformation in [CF82] do not guarantee that the resulting atts are single-use.

contrast to [CF82], different sets of inherited attributes are used for the arguments of different functions [Höf98, EM99].

Since step (ii) does not require the second att to be single-use, we introduce the larger class of so called *restricted-use* mtts, which can still be handled by our transformation.

(iii) For transforming atts back into mtts we use a well known construction from [Fra82], again making use of the above mentioned correspondence of functions and their context arguments to synthesized and inherited attributes, respectively.

The whole transformation process is illustrated in Figure 2 for the composition of m_{rev} with itself. The att obtained after step (ii) is called a_{id}. The final result is the mtt m_{id} with initial expression $e'_{rr} = \overline{revrev}\ l$ and the following equations:

$$
\begin{array}{lll}
\overline{revrev}\ \ (Cons\ a\ x) & = & Cons\ a\ (\overline{revrev}\ x) \\
\overline{revrev}\ \ Nil & = & Nil
\end{array}
$$

Note that this is the expected identity function on lists[3].

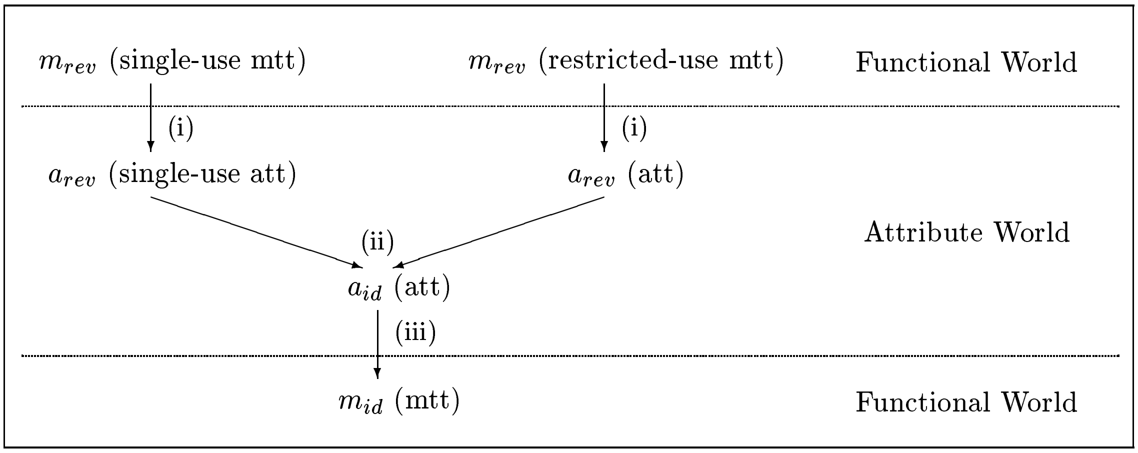

Figure 2: Transformation for the twofold composition of m_{rev}.

The elimination of intermediate results in functional programs via composition of atts and attribute grammars was first independently described in [Küh97, Küh98] and [CDPR98, CDPR99], respectively. In the former papers for step (i) no direct transformation was used, but a construction, which first splits a weakly single-use mtt into a single-use tdtt and a so called *yield function*, which handles argument substitutions [Eng80]. This yield function can be realized by an att. Finally, the composition construction of step (ii) delivers for the single-use tdtt (viewed as a single-use att without inherited attributes) and the att an att. In this paper we have chosen the direct construction, because we think that it is more intuitive. Also, it allows for more general restricted-use mtts to be transformed into atts. Though also in the latter papers [CDPR98, CDPR99] step (i) is performed directly, the relevant classes of functions, which can be treated by the three transformation steps, are not specified. A restricted instance of attribute grammar composition is also

[3] The replacement of e_{rr} with e'_{rr} is semantic-preserving only for finite lists l. With infinite data structures and lazy evaluation, the copy rules elimination can turn non-terminating programs into terminating ones.

handled in [KGF01], where a single local rule (similar to the *foldr/build*-rule of *shortcut deforestation* [GLP93]) is used to eliminate intermediate lists in compositions of *map*-style list transformers.

Recently, in [Voi01, VK01] a direct composition construction for subclasses of mtts is given, which generalizes the techniques of [Eng81, EV85], not requiring at least one mtt to be a tdtt. Compared with the indirect composition technique via atts presented here, it allows on the one hand the larger syntactic class of noncopying mtts as first composition partner, but on the other hand it can only handle the more restricted syntactic class of weakly single-use mtts as second composition partner.

Besides this introduction, the paper contains eight further sections. In Section 2 elementary notions and notations are fixed. Section 3 introduces our functional language and mtts. Section 4 presents the direct composition techniques for mtts and tdtts. Atts and their relation to mtts are considered in Section 5 and Section 6, respectively. In Section 7 the composition technique for atts is discussed. Finally, Sections 8 and 9 informally compare the techniques with classical deforestation and point to further research topics, respectively.

2 Preliminaries

We denote the set of natural numbers including 0 by $I\!N$ and the set of natural numbers without 0 by $I\!N_+$. For every $m \in I\!N$, the set $\{1,\ldots,m\}$ is denoted by $[m]$, thus $[0] = \emptyset$. We will use the sets $X = \{x_1, x_2, x_3, \ldots\}$ and $Y = \{y_1, y_2, y_3, \ldots\}$ of *recursion variables* and *context variables*, respectively. For every $k \in I\!N$, we use the abbreviations $X_k = \{x_1,\ldots,x_k\}$ and $Y_k = \{y_1,\ldots,y_k\}$, thus $X_0 = Y_0 = \emptyset$.

For an arbitrary set K, the cardinality of K is denoted by $card(K)$ and the set of all subsets of K is denoted by $\mathcal{P}(K)$. If K is a nonempty and finite subset of $I\!N$, then $max(K)$ denotes the maximum of K. Analogous for $min(K)$. Additionally, we define $max(\emptyset) = 0$. Let K and K' be two sets. A relation $f \subseteq K \times K'$ is a *partial function*, if for every $(k, k_1) \in f$ and $(k, k_2) \in f$, the elements k_1 and k_2 are equal. Such a partial function is denoted by $f : K - \rightarrow K'$. If additionally for every $k \in K$ there is a $k' \in K'$ such that $(k, k') \in f$, then f is a *total function*; this will be denoted by $f : K \longrightarrow K'$.

If A is an alphabet, then A^* denotes the set of *strings* over A. The empty string is denoted by ε. For a string w and two lists $v_1,\ldots,v_n$ and $w_1,\ldots,w_n$ of strings such that no pair v_j and $v_{j'}$ overlaps in w, we abbreviate by $w[v_1/w_1,\ldots,v_n/w_n]$ the string, which is obtained from w by substituting for every $j \in [n]$ every occurrence of v_j in w by w_j.

Let $\Rightarrow$ be a binary relation on some set K. Then, $\Rightarrow^*$ and $\Rightarrow^+$ denote the transitive, reflexive closure of $\Rightarrow$ and the transitive closure of $\Rightarrow$, respectively. 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* $\Rightarrow$. If the normal form of k with respect to $\Rightarrow$ exists and if it is unique, then it is denoted by $nf(\Rightarrow, k)$. The relation $\Rightarrow$ is called *confluent*, if for every $k, k_1, k_2 \in K$ with $k \Rightarrow^* k_1$ and $k \Rightarrow^* k_2$, there is a $k' \in K$ such that $k_1 \Rightarrow^* k'$ and $k_2 \Rightarrow^* k'$. It is called *terminating*, if there is no infinite derivation of $\Rightarrow$. If $\Rightarrow$ is terminating and confluent, then for every $k \in K$, the normal form of k exists and it is unique.

A *ranked alphabet* is a pair $(\Sigma, rank_\Sigma)$ where Σ is a finite set and $rank_\Sigma : \Sigma \longrightarrow I\!N$ is a mapping, which associates with every symbol $\sigma \in \Sigma$ a natural number $rank_\Sigma(\sigma)$, called the *rank of* σ. If $\sigma \in \Sigma$ with $rank_\Sigma(\sigma) = n$, and Σ is clear from the context, then we also write $\sigma^{(n)}$ or $rank(\sigma) = n$. If the rank function is clear from the context, then we write Σ instead of $(\Sigma, rank_\Sigma)$. The set of elements with rank n is denoted by $\Sigma^{(n)}$. We define $mx_\Sigma = max(\{n \mid \Sigma^{(n)} \neq \emptyset\})$.

For a ranked alphabet Σ and a set K of symbols, the set of *trees* (or: *terms*) *over* Σ *indexed by* K, denoted by $T_\Sigma(K)$, is the smallest subset $T \subseteq (\Sigma \cup K \cup \{(,)\})^*$ such that (i) $\Sigma^{(0)} \cup K \subseteq T$ and (ii) for every $\sigma \in \Sigma^{(n)}$ with $n \in I\!N_+$ and $t_1, \ldots, t_n \in T$: $(\sigma\ t_1 \ldots t_n) \in T$. We often omit the outermost parentheses of a tree. The set $T_\Sigma(\emptyset)$ is abbreviated by T_Σ.

Let Σ be a ranked alphabet. A *tree transformation on* T_Σ is a total function $\tau : T_\Sigma \longrightarrow T_\Sigma$. Let τ_1 and τ_2 be tree transformations on T_Σ, then the *composition* of τ_1 and τ_2 is denoted by $\tau_1;\tau_2$, i.e., $\tau_1;\tau_2 : T_\Sigma \longrightarrow T_\Sigma$ and for every $t \in T_\Sigma$: $(\tau_1;\tau_2)(t) = \tau_2(\tau_1(t))$. For two classes $\mathcal{T}_1$ and $\mathcal{T}_2$ of tree transformations we denote the class $\{\tau_1;\tau_2 \mid \tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2\}$ of tree transformations by $\mathcal{T}_1;\mathcal{T}_2$.

The following functions are defined inductively on the structure of trees in $T_\Sigma(K)$:

- $paths : T_\Sigma(K) \longrightarrow \mathcal{P}(I\!N^*)$ delivers the *set of paths of a tree* $t \in T_\Sigma(K)$.
  If $t = (\sigma\ t_1 \ldots t_n)$ with $\sigma \in \Sigma^{(n)}$, $n \in I\!N_+$, and $t_1, \ldots, t_n \in T_\Sigma(K)$, then
  $paths((\sigma\ t_1 \ldots t_n)) = \{\varepsilon\} \cup \{p \mid p = jp', j \in [n], p' \in paths(t_j)\}$.
  If $t \in \Sigma^{(0)} \cup K$, then $paths(t) = \{\varepsilon\}$.

- $label : T_\Sigma(K) \times I\!N^* - \to \Sigma \cup K$ delivers the *label of the node of a tree* $t \in T_\Sigma(K)$ *reached by a path* $p \in paths(t)$.
  If $t = (\sigma\ t_1 \ldots t_n)$ with $\sigma \in \Sigma^{(n)}$, $n \in I\!N_+$, and $t_1, \ldots, t_n \in T_\Sigma(K)$, then
  $label((\sigma\ t_1 \ldots t_n), p) = \sigma$, if $p = \varepsilon$,
  $label((\sigma\ t_1 \ldots t_n), p) = label(t_j, p')$, if $p = jp'$ for some $j \in [n]$.
  If $t \in \Sigma^{(0)} \cup K$, then $label(t, \varepsilon) = t$.

- $subtree : T_\Sigma(K) \times I\!N^* - \to T_\Sigma(K)$ delivers the *subtree of a tree* $t \in T_\Sigma(K)$ *reached by a path* $p \in paths(t)$.
  If $t = (\sigma\ t_1 \ldots t_n)$ with $\sigma \in \Sigma^{(n)}$, $n \in I\!N_+$, and $t_1, \ldots, t_n \in T_\Sigma(K)$, then
  $subtree((\sigma\ t_1 \ldots t_n), p) = (\sigma\ t_1 \ldots t_n)$, if $p = \varepsilon$,
  $subtree((\sigma\ t_1 \ldots t_n), p) = subtree(t_j, p')$, if $p = jp'$ for some $j \in [n]$.
  If $t \in \Sigma^{(0)} \cup K$, then $subtree(t, \varepsilon) = t$.

3 Functional Programs and Macro Tree Transducers

3.1 Syntax

We consider a simple first-order, constructor-based functional programming language P as source and target language for our transformations. Every program $p \in P$ consists of several units and every unit consists of several function definitions. The functions of a unit 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*. If in the right-hand side of a definition of a function f there is a call of a function f', then f and f' must be defined in the same unit and the first argument of f' has to be a subtree of the first argument in the corresponding left-hand side.

For simplicity we choose a unique ranked alphabet C_p of constructors, which is used to build up input trees and output trees of every function in p. We assume that every function definition is complete with respect to T_{C_p}, but in some example programs and constructions we relax this completeness condition by leaving out those equations, which are not intended to be used in evaluations. The introduction of types would have implied a more exact presentation, but would have caused a technical overhead that considerably hampers the understandability of the text.

Definition 3.1 Let C and F be ranked alphabets of *constructors* and *function symbols* (for short *functions*), respectively, such that X, Y, C, and F are pairwise disjoint. We define the classes P, U, R, and Q of *programs*, *units*, *rules* (or: *equations*), and *right-hand sides*, respectively, by the following grammar and the subsequent restrictions. We assume that p, u, r, ϱ, c, and f (also equipped with indices) range over the sets P, U, R, Q, C, and F, respectively.

$$\begin{array}{lcll} p & ::= & u_1 \dots u_l & \text{(program)} \\ u & ::= & r_1 \dots r_h & \text{(unit)} \\ r & ::= & f\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = \varrho & \text{(rule (or: equation))} \\ \varrho & ::= & y_j \mid c\ \varrho_1 \dots \varrho_k \mid f\ x_j\ \varrho_1 \dots \varrho_n & \text{(right-hand side)} \end{array}$$

The sets of constructors, functions, rules, and units that occur in $p \in P$ are denoted by C_p, F_p, R_p, and U_p, respectively. C_p and F_p are considered as ranked alphabets. We assume that $C_p^{(0)} \neq \emptyset$. The ranked alphabet of functions defined in $u \in U_p$ is denoted by F_u. The set of rules in $u \in U_p$ is denoted by R_u.

For every $u, u' \in U_p$ with $u \neq u'$: $F_u \cap F_{u'} = \emptyset$. For every $u \in U_p$, $f \in F_u^{(n+1)}$, and $c \in C_p^{(k)}$ there is exactly one equation of the form $f\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = rhs_{f,c}$ with $rhs_{f,c} \in M\text{-}RHS(F_u, C_p, X_k, Y_n)$ in R_u, where for every $u \in U_p$ and $k, n \in \mathbb{N}$ the set $M\text{-}RHS(F_u, C_p, X_k, Y_n)$ is the smallest set RHS, such that

1. For every $f \in F_u^{(h+1)}$, $j \in [k]$, and $\varrho_1, \dots, \varrho_h \in RHS$: $(f\ x_j\ \varrho_1 \dots \varrho_h) \in RHS$.
2. For every $c \in C_p^{(h)}$ and $\varrho_1, \dots, \varrho_h \in RHS$: $(c\ \varrho_1 \dots \varrho_h) \in RHS$.
3. For every $j \in [n]$: $y_j \in RHS$. □

Example 3.2 The following program $p_1 \in P$ consists of the two units u_{doub} and u_{count}, containing definitions of the function *doub* and *count*, respectively. Note that the function definitions are complete with respect to $T_{C_{tree}}$, where $C_{tree} = \{T^{(2)}, S^{(1)}, Z^{(0)}\}$.

$$\begin{array}{llcll} doub & (T\ x_1\ x_2) & & = & T\ (doub\ x_1)\ (doub\ x_2) \\ doub & (S\ x_1) & & = & S\ (S\ (doub\ x_1)) \\ doub & Z & & = & Z \\[1ex] count & (T\ x_1\ x_2) & y_1 & = & count\ x_1\ (count\ x_2\ y_1) \\ count & (S\ x_1) & y_1 & = & S\ (count\ x_1\ y_1) \\ count & Z & y_1 & = & y_1 \end{array}$$

The units u_{doub} and u_{count} double and collect, respectively, all occurrences of S in a tree. □

Now we show how also functions on, e.g., polymorphic lists and some higher-order functions can be regarded as programs in our first-order language by choosing appropriate constructors and function symbols.

Example 3.3 The following program $p_2 \in P$ consists of the four units $u_{\underline{map\ f}}$, u_{rev}, u_{app} and u_{mir}, where the respective unit u consists of those equations, which define the function mentioned in the index of u. The functions are defined for elements of $T_{C_{list}}$, where $C_{list} = \{\underline{Cons\ a}^{(1)}, \underline{Cons\ (f\ a)}^{(1)}, Nil^{(0)}\}$, however the function definitions are not complete with respect to this input domain. To fulfill the definition of a program, additional rules may be added.

$$
\begin{array}{llcll}
\underline{map\ f} & (\underline{Cons\ a}\ x_1) & & = & \underline{Cons\ (f\ a)}\ (\underline{map\ f}\ x_1) \\
\underline{map\ f} & Nil & & = & Nil \\[4pt]
rev & (\underline{Cons\ a}\ x_1) & y_1 & = & rev\ x_1\ (\underline{Cons\ a}\ y_1) \\
rev & Nil & y_1 & = & y_1 \\[4pt]
app & (\underline{Cons\ a}\ x_1) & y_1 & = & \underline{Cons\ a}\ (app\ x_1\ y_1) \\
app & Nil & y_1 & = & y_1 \\[4pt]
mir & (\underline{Cons\ a}\ x_1) & y_1 & = & \underline{Cons\ a}\ (mir\ x_1\ (\underline{Cons\ a}\ y_1)) \\
mir & Nil & y_1 & = & y_1
\end{array}
$$

The units perform the following actions: $u_{\underline{map\ f}}$ applies f to every element of a list. u_{rev} reverses an input list (given in the first argument) by accumulation in the second argument. u_{app} appends two lists. u_{mir} appends to an input list the reversed input list. □

Similarly, the well known *foldr*-function can be defined in a unit.

Note that classical deforestation, which is not considered formally in this paper, in general transforms programs of P into programs outside of P (cf. the example in the introduction, where in a right-hand side for $\overline{revmap}$ the function *map f* is applied to a context variable).

A unit can be seen as an extended scheme of primitive recursion over trees, allowing simultaneous recursion and nested function calls in argument positions. In the literature [Eng80, EV85] such schemes are called macro tree transducers.

Definition 3.4 Let $p \in P$.

- A unit $u \in U_p$ is also called *macro tree transducer unit* (for short *mtt unit*) and is characterized by the triple (F_u, C_p, R_u).
- An mtt unit u with $F_u = F_u^{(1)}$ is called *top-down tree transducer unit* (for short *tdtt unit*).
- A unit $u \in U_p$ is called linear, if for every rule $f\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = \varrho$ in R_u, ϱ is linear, i.e., every variable occurs at most once in ϱ. p is called linear, if for every $u \in U_p$, the unit u is linear. □

3.2 Semantics

In the following we will define a nondeterministic reduction semantics for our programs. We will not fix a deterministic (call-by-value, call-by-name, or call-by-need) reduction semantics, because (i) in the context of mtt units they do not differ (and thus our transformations are valid with respect to all of them) and because (ii) in this paper we do not compare the efficiency of programs formally (with respect to the number of steps of a particular reduction relation). However, note that classical deforestation and also the comparison between the mtt composition technique and classical deforestation in [Küh99, Höf99] rely on call-by-need semantics. The nondeterministic reduction relation is defined on the set $T_{C_p \cup F_p}$ of *sentential forms for* p.

Definition 3.5 Let $p \in P$ and $u \in U_p$. The *reduction relation of* p, denoted by $\Rightarrow_p$ (*reduction relation of* u, denoted by $\Rightarrow_u$, respectively), is the smallest binary relation $\Rightarrow \subseteq T_{C_p \cup F_p} \times T_{C_p \cup F_p}$, such that

1. For every $f \in F_p^{(n+1)}$, $c \in C_p^{(k)}$, $t_1, \ldots, t_k, t'_1, \ldots, t'_n \in T_{C_p \cup F_p}$, and $(f\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = rhs_{f,c}) \in R_p$ ($\in R_u$, respectively):
$$f\ (c\ t_1 \ldots t_k)\ t'_1 \ldots t'_n \Rightarrow rhs_{f,c}[x_1/t_1, \ldots, x_k/t_k, y_1/t'_1, \ldots, y_n/t'_n]$$
2. For every $c \in (C_p \cup F_p)^{(k)}$, $j \in [k]$, and $t_1, \ldots, t_k, t'_j \in T_{C_p \cup F_p}$ such that $t_j \Rightarrow t'_j$:
$$c\ t_1 \ldots t_{j-1}\ t_j\ t_{j+1} \ldots t_k \Rightarrow c\ t_1 \ldots t_{j-1}\ t'_j\ t_{j+1} \ldots t_k$$ □

From [EV85] it is known that for every $p \in P$, $u \in U_p$, $f \in F_u^{(n+1)}$, $t_0, t_1, \ldots, t_n \in T_{C_p}$ the normal form $nf(\Rightarrow_u, (f\ t_0\ t_1 \ldots t_n))$ and thus also $nf(\Rightarrow_p, (f\ t_0\ t_1 \ldots t_n))$ exists and does not contain function symbols. The proof is based on the result, that $\Rightarrow_u$ is terminating and confluent. More generally it can be shown as extension of the previous result that for every $t \in T_{C_p \cup F_p}$ the normal form $nf(\Rightarrow_p, t)$ exists and does not contain function symbols.

In the framework of two mtt units u_1 and u_2, a function $f_1 \in F_{u_1}$ produces an intermediate result, which is consumed by a function $f_2 \in F_{u_2}$, iff an application of f_1 occurs in the *first* argument of an application of f_2. Thus, we would like to optimize the evaluation of expressions of the form $(f_2\ (f_1\ t_0\ t_1 \ldots t_n)\ t'_1 \ldots t'_{n'})$, where $t_0, t_1, \ldots, t_n, t'_1, \ldots, t'_{n'} \in T_{C_p}$.

Since the particular constructor tree t_0 is not relevant for the transformations, we abstract it by a recursion variable x_1 and transform expressions of the form $(f_2\ (f_1\ x_1\ t_1 \ldots t_n)\ t'_1 \ldots t'_{n'})$, which we call *2-fold composition expressions*. Our transformations will take a program and a 2-fold composition expression as input and will deliver a new program and a new expression, the evaluation of which avoids the construction of intermediate results. It turns out that the new expression is not always of the form $(f\ x_1\ t''_1 \ldots t''_{n''})$, but has the form of a right-hand side of an mtt unit, in which the only variable x_1 occurs. Thus, we use the following more general definitions.

Definition 3.6 Let $p \in P$.

- For every $u \in U_p$ the set $M\text{-}RHS(F_u, C_p, X_1, \emptyset)$, abbreviated by IE_u, is called the set of *initial expressions for* u.
- For every $n \in I\!N$, the set of *n-fold composition expressions for* p, denoted by $n\text{-}CE_p$, is defined as follows:
  - $0\text{-}CE_p = \{x_1\}$
  - For every $n \in I\!N$, $(n+1)\text{-}CE_p = \{\varrho[x_1/e] \mid e \in n\text{-}CE_p, u \in U_p, \varrho \in IE_u\}$.
- The set $CE_p = \bigcup_{n \in I\!N} n\text{-}CE_p$ is called the set of *composition expressions for* p.

- Let $u \in U_p$ be an mtt unit (a tdtt unit, respectively) and $\varrho_u \in IE_u$. The tuple $m = (F_u, C_p, R_u, \varrho_u)$ is called *macro tree transducer* (for short *mtt*) [Eng80, EV85] (*top-down tree transducer* (for short *tdtt*) [Rou70, Tha70, Eng75], respectively). $\square$

In the following, we will also consider mtts and mtt units without explicitly mentioning their underlying programs. In this case our statements are valid for all programs containing the respective rules of the mtts.

Definition 3.7 Let $m = (F, C, R, \varrho)$ be an mtt. For every $c \in C^{(k)}$, we denote by R_c the set of all rules $f\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = rhs_{f,c}$, with $f \in F^{(n+1)}$, in R. $\square$

Example 3.8 Let p_2 be the program from Example 3.3.

- 1-CE_{p_2} contains for example $(\underline{map\ f}\ x_1)$, $(mir\ x_1\ Nil)$ and $(mir\ x_1\ (mir\ x_1\ Nil))$.
- 2-CE_{p_2} contains for example $(mir\ (\underline{map\ f}\ x_1)\ Nil)$, $(\underline{map\ f}\ (mir\ x_1\ Nil))$ and $(rev\ (rev\ x_1\ Nil)\ Nil)$.
- The tuple $m_{\underline{map\ f}} = (F_{u_{\underline{map\ f}}}, C_{list}, R_{u_{\underline{map\ f}}}, \varrho_{u_{\underline{map\ f}}})$ where $F_{u_{\underline{map\ f}}} = \{\underline{map\ f}^{(1)}\}$, $\varrho_{u_{\underline{map\ f}}} = (\underline{map\ f}\ x_1)$, and $R_{u_{\underline{map\ f}}}$ contains the "$\underline{map\ f}$-rules", is a tdtt.
- The tuple $m_{mir} = (F_{u_{mir}}, C_{list}, R_{u_{mir}}, \varrho_{u_{mir}})$ where $F_{u_{mir}} = \{mir^{(2)}\}$, $\varrho_{u_{mir}} = (mir\ x_1\ Nil)$, and $R_{u_{mir}}$ contains the "mir-rules", is an mtt. $\square$

Definition 3.9 Let $p \in P$.

- Let $e \in CE_p$. The function $\tau_{(p,e)} : T_{C_p} \longrightarrow T_{C_p}$ defined by $\tau_{(p,e)}(t) = nf(\Rightarrow_p, e[x_1/t])$ for every $t \in T_{C_p}$ is called the *tree transformation induced by p and e*.
- Let $u \in U_p$ and let $m = (F_u, C_p, R_u, \varrho_u)$ be an mtt. The function $\tau_m : T_{C_p} \longrightarrow T_{C_p}$ defined by $\tau_m = \tau_{(p,\varrho_u)}$ is called the *tree transformation induced by m*.
- We define the *classes* $MT = \{\tau_m \mid m \text{ is an mtt}\}$ and $T = \{\tau_m \mid m \text{ is a tdtt}\}$ of *macro tree transformations* and *top-down tree transformations*, respectively. $\square$

Note that the classes MT and T are equal to the usual classes of tree transformations induced by mtts and tdtts, respectively, which are defined in the literature [FV98]. This is due to the fact that for an mtt unit u and an initial expression ϱ_u, we can construct for every $c \in C^{(k)}$ a new rule $f_{in}\ (c\ x_1 \dots x_k) = nf(\Rightarrow_u, \varrho_u[x_1/(c\ x_1 \dots x_k)])$, where f_{in} is a new state (the initial state in the sense of [FV98]).

For $p \in P$, $n \geq 2$, $e \in n$-CE_p, and $t \in T_{C_p}$, the evaluation of $nf(\Rightarrow_p, e[x_1/t])$ constructs $n-1$ intermediate results. Hence, to avoid intermediate results we would like to transform the pair (p, e) into a pair (p', e') with $p' \in P$ and $e' \in$ 1-CE_p, such that $\tau_{(p,e)} = \tau_{(p',e')}$. Unfortunately, in general this transformation is not possible, because the class MT is not closed under composition [EV85]. In this paper we will only consider the case $n = 2$, for which we will give three sufficient conditions such that our desired transformation is possible.

The following observation allows to identify the tree transformation induced by a program and an n-fold composition expression with the n-fold composition of tree transformations induced by their "corresponding" mtts.

Observation 3.10 Let $p \in P$, $n \geq 2$, and for every $j \in [n]$ let $u_j \in U_p$, $\varrho_j \in IE_{u_j}$, and $m_j = (F_{u_j}, C_p, R_{u_j}, \varrho_j)$. Then, $\tau_{(p,\varrho_n[x_1/\varrho_{n-1}[...\varrho_2[x_1/\varrho_1]...]])} = \tau_{m_1}; \tau_{m_2}; \ldots; \tau_{m_n}$.
In particular, we also have $\{\tau_{(p,\varrho_2[x_1/\varrho_1])} \mid p \in P, u_1, u_2 \in U_p, \varrho_1 \in IE_{u_1}, \varrho_2 \in IE_{u_2}\} = MT; MT$ and similarly, if one or both of the occurrences of MT is replaced by T or by other subclasses of MT that will be introduced later. □

Thus in this paper we will describe our optimizations in the framework of the composition of macro tree transformations, i.e., we transform a pair $(p, \varrho_2[x_1/\varrho_1])$ with $p \in P$, $u_1, u_2 \in U_p$, $\varrho_1 \in IE_{u_1}$, $\varrho_2 \in IE_{u_2}$ by composing (if possible) the two mtts $(F_{u_1}, C_p, R_{u_1}, \varrho_1)$ and $(F_{u_2}, C_p, R_{u_2}, \varrho_2)$. Vice versa, an mtt (F_u, C_p, R_u, ϱ) resulting from this composition is considered as a pair (p', ϱ) with p' consisting of the equations in R_u.

Moreover, the above composition of mtts can also be viewed in a more global context, for example to transform a pair $(p, e_2[x_1/\varrho_2[x_1/\varrho_1[x_1/e_1]]])$ with $p \in P$, $e_1, e_2 \in CE_p$, $u_1, u_2 \in U_p$, $\varrho_1 \in IE_{u_1}$, $\varrho_2 \in IE_{u_2}$ into the pair $(p', e_2[x_1/\varrho[x_1/e_1]])$ where p' is constructed from p by adding the equations in R_u. This view possibly allows subsequent transformations, but will entail questions on the associativity of our composition techniques.

4 Elimination of Intermediate Results Using Composition Techniques for Macro Tree Transducers

In [Rou70] it was shown that the composition of two tdtts can be simulated by only one tdtt. This result was generalized in [Eng81, EV85], 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. We already know that there is no such construction for the composition of two mtts, because the class MT is not closed under composition [EV85].

4.1 Composition of Top-down Tree Transducers with Macro Tree Transducers

The central idea of the composition technique for the composition of a tdtt m_1 with an mtt 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 mtt m. For this purpose, m uses $F_1 \times F_2$ as function set, where F_1 and F_2 are the sets of functions of m_1 and m_2, respectively. In the following, we abbreviate every pair $(f_1, f_2) \in F_1 \times F_2$ by $\overline{f_1 f_2}$.

Transformation 4.1 Let $m_1 = (F_1, C, R_1, \varrho_1)$ be a tdtt and $m_2 = (F_2, C, R_2, \varrho_2)$ be an mtt. We construct an mtt $m = (F, C, R, \varrho)$ as follows:

1. From m_2 we produce the mtt unit $u_2' = (F_2', C', R_2')$, which is able to translate right-hand sides of equations of R_1.
   - $F_2' = \{f_2'^{(n+1)} \mid f_2 \in F_2^{(n+1)}, n \in \mathbb{N}\}$.
   - $C' = C \cup F_1 \cup \{\overline{f_1 f_2}^{(n+1)} \mid f_1 \in F_1^{(1)}, f_2 \in F_2^{(n+1)}, n \in \mathbb{N}\} \cup \{x_j^{(0)} \mid j \in [mx_C]\}$.

- – R_2' contains the rules of R_2, where every occurrence of $f_2 \in F_2$, $x_j \in X_{mx_C}$, and $y_{j'} \in Y_{mx_{F_2}-1}$ is renamed[4] to f_2', x_j', and $y_{j'}'$, respectively.
  – For every $f_2 \in F_2^{(n+1)}$ and $f_1 \in F_1^{(1)}$, we add the following rule[5] to R_2':
  $$f_2'\ (f_1\ x_1')\ y_1' \ldots y_n' = \overline{f_1 f_2}\ x_1'\ y_1' \ldots y_n'$$

2. Now we can give the mtt $m = (F, C, R, \varrho)$ as follows:
   - $F = \{\overline{f_1 f_2}^{(n+1)} \mid f_1 \in F_1^{(1)}, f_2 \in F_2^{(n+1)}, n \in I\!N\}$.
   - For every $f_2 \in F_2^{(n+1)}$, $f_1 \in F_1^{(1)}$, and $c \in C^{(k)}$, R contains the rule
   $$\overline{f_1 f_2}\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = nf(\Rightarrow_{u_2'}, f_2'\ (rhs_{f_1,c})\ y_1 \ldots y_n).$$
   - $\varrho = nf(\Rightarrow_{u_2'}, \varrho_2'[x_1/\varrho_1])$, where ϱ_2' is constructed from ϱ_2 by renaming every occurrence of $f_2 \in F_2$ in f_2'.

Then $\tau_m = \tau_{m_1}; \tau_{m_2}$ holds. We omit the proof, but mention that in [Eng81] another construction was used, which first splits the mtt into a tdtt and a *yield function*, which handles argument substitutions [Eng80], then composes the two tdtts [Rou70], and finally joins the resulting tdtt to the yield function. We get the same transformation result in one step by avoiding the explicit splitting and joining (cf. also [Küh99]). □

Corollary 4.2 $T; MT \subseteq MT$ □

We only state that even the result $T; MT = MT$ holds [Eng81, EV85]. As an example, we want to compose the $\underline{map\ f}$- and mir-functions from Example 3.3. There, the definition of mir was not complete with respect to C_{list}. Here, however, we need an instance of the $\underline{Cons\ a}$-rule, hence we consider a slightly different program.

Example 4.3 Let p_3 be the program that contains the unit $u_{\underline{map\ f}}$ from Example 3.3 and a unit u_{mir} with the following rules:

$$\begin{array}{llcl} mir & (\underline{Cons\ (f\ a)}\ x_1)\ y_1 & = & \underline{Cons\ (f\ a)}\ (mir\ x_1\ (\underline{Cons\ (f\ a)}\ y_1)) \\ mir & Nil\ \ y_1 & = & y_1 \end{array}$$

The mtts $m_{\underline{map\ f}}$ and m_{mir} are defined with initial expressions $\varrho_{u_{\underline{map\ f}}} = (\underline{map\ f}\ x_1)$ and $\varrho_{u_{mir}} = (mir\ x_1\ Nil)$, respectively. We consider $(mir\ (\underline{map\ f}\ x_1)\ Nil) \in 2\text{-}CE_{p_3}$. Thus, for the mtts $m_{\underline{map\ f}}$ and m_{mir} we construct an mtt $m_{\underline{map\ f};mir}$ with $\tau_{m_{\underline{map\ f};mir}} = \tau_{m_{\underline{map\ f}}}; \tau_{m_{mir}}$. Note that $mx_C = 1$.

The mtt unit $u_{mir}' = (\{mir'^{(2)}\}, C_{list} \cup \{\underline{map\ f}^{(1)}, \overline{\underline{map\ f}mir}^{(2)}, x_1^{(0)}\}, R')$, where R' contains the rules

$$\begin{array}{llcl} mir' & (\underline{Cons\ (f\ a)}\ x_1')\ y_1' & = & \underline{Cons\ (f\ a)}\ (mir'\ x_1'\ (\underline{Cons\ (f\ a)}\ y_1')) \\ mir' & Nil\ \ y_1' & = & y_1' \\ mir' & (\underline{map\ f}\ x_1')\ \ y_1' & = & \overline{\underline{map\ f}mir}\ x_1'\ y_1' \end{array}$$

is able to translate right-hand sides of equations of $m_{\underline{map\ f}}$.

[4] The renaming of functions is necessary, because $m_1 = m_2$ and thus $F_1 = F_2$ is possible. The renaming of variables avoids confusion with new constructors.

[5] In a strong sense, $\overline{f_1 f_2}\ x_1'\ y_1' \ldots y_n'$ is no legal right-hand side of an mtt unit, because the recursion variable x_1' does not occur as first argument of a function symbol, but of a constructor. An additional identity function id in u_2' and the right-hand side $\overline{f_1 f_2}\ (id\ x_1')\ y_1' \ldots y_n'$ would solve the problem formally. Moreover, we have omitted rules that are not used.

Now the construction yields the mtt $m_{\underline{map\ f};mir} = (F_{u_{\underline{map\ f};mir}}, C_{list}, R_{u_{\underline{map\ f};mir}}, \varrho_{u_{\underline{map\ f};mir}})$ with $F_{u_{\underline{map\ f};mir}} = \{\overline{\underline{map\ f}mir}^{(2)}\}$,

$$\begin{array}{rcl} \varrho_{u_{\underline{map\ f};mir}} & = & nf(\Rightarrow_{u'_{mir}}, (mir'\ (\underline{map\ f}\ x_1)\ Nil)) \\ & = & \overline{\underline{map\ f}mir}\ x_1\ Nil, \end{array}$$

and $R_{u_{\underline{map\ f};mir}}$ contains the following rules, where the left- and right-hand sides can be found as the first and the last elements of the equation sequences, respectively:

$$\begin{array}{rl} & \overline{\underline{map\ f}mir}\ (\underline{Cons\ a}\ x_1)\ y_1 \\ = & nf(\Rightarrow_{u'_{mir}}, (mir'\ (rhs_{\underline{map\ f}, \underline{Cons\ a}})\ y_1)) \\ = & nf(\Rightarrow_{u'_{mir}}, (mir'\ (\underline{Cons\ (f\ a)}\ (\underline{map\ f}\ x_1))\ y_1)) \\ = & nf(\Rightarrow_{u'_{mir}}, (\underline{Cons\ (f\ a)}\ (mir'\ (\underline{map\ f}\ x_1)\ (\underline{Cons\ (f\ a)}\ y_1)))) \\ = & \underline{Cons\ (f\ a)}\ (\overline{\underline{map\ f}mir}\ x_1\ (\underline{Cons\ (f\ a)}\ y_1)) \end{array}$$

$$\begin{array}{rl} & \overline{\underline{map\ f}mir}\ Nil\ y_1 \\ = & nf(\Rightarrow_{u'_{mir}}, (mir'\ (rhs_{\underline{map\ f}, Nil})\ y_1)) \\ = & nf(\Rightarrow_{u'_{mir}}, (mir'\ Nil\ y_1)) \\ = & y_1 \end{array}$$

Note that, following the idea of higher-order macros from [Wad90], $\underline{map\ f}$ was viewed as a first-order function, integrating the argument f into the function name, which leads to a "free" occurrence of f in the right-hand side of the rule for $\underline{Cons\ a}$. After the transformation, the f can again be released from the function name, giving the following more readable definition:

$$\begin{array}{llllll} \overline{mapmir}\ f & (\underline{Cons\ a}\ x_1) & y_1 & = & \underline{Cons\ (f\ a)}\ (\overline{mapmir}\ f\ x_1\ (\underline{Cons\ (f\ a)}\ y_1)) \\ \overline{mapmir}\ f & Nil & y_1 & = & y_1 \end{array}$$

□

The mtt $m_{\underline{map\ f};mir}$ of Example 4.3 corresponds to the program and expression, which is produced by a deforestation algorithm that is adjusted to mtts in order to guarantee termination [Küh99]. This adjustment assumes that context arguments of function applications in right-hand sides are implicitly abstracted by let-expressions. By a slight generalization of Lemma 20 in [Küh99], the correspondence can be formalized: For every 2-fold composition expression $\varrho_2[x_1/\varrho_1]$ where ϱ_1 and ϱ_2 are initial expressions for a linear tdtt unit and a linear mtt unit, respectively, the composition technique and classical deforestation deliver the same result (apart from the fact that sometimes classical deforestation constructs less functions than the composition technique). Thus, since classical deforestation does not deteriorate the efficiency (with respect to the number of steps in a call-by-need reduction relation) of linear programs, the same holds for the composition technique in Transformation 4.1.

4.2 Composition of Macro Tree Transducers with Top-down Tree Transducers

The composition of mtts with tdtts is handled in Transformation 4.4. We use the additional observation that for $f_1 \in F_1$, $f_2 \in F_2$, and a composition expression of the form $(f_2\ (f_1\ x_1\ t_1 \ldots t_n))$, instances of the "results" of applications of functions from F_2 to the context arguments $t_1, \ldots, t_n$, instead of instances of $t_1, \ldots, t_n$ themselves, 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 4.4 Let $m_1 = (F_1, C, R_1, \varrho_1)$ be an mtt and $m_2 = (F_2, C, R_2, \varrho_2)$ be a tdtt. Let $mx_Y = mx_{F_1} - 1$, $h = card(F_2)$, and let $f_{2,1}, \ldots, f_{2,h}$ be a fixed order of all functions in F_2. We construct an mtt $m = (F, C, R, \varrho)$ as follows:

1. From m_2 we produce the tdtt unit $u_2' = (F_2', C', R_2')$, which is able to translate right-hand sides of equations of R_1.
   - $F_2' = \{f_2'^{(1)} \mid f_2 \in F_2^{(1)}\}$.
   - $C' = C \cup F_1 \cup \{\overline{f_1 f_2}^{(hn+1)} \mid f_1 \in F_1^{(n+1)}, f_2 \in F_2^{(1)}, n \in I\!N\} \cup \{x_j^{(0)} \mid j \in [mx_C]\} \cup \{y_j^{(0)} \mid j \in [h \cdot mx_Y]\}$.
   - – R_2' contains the rules of R_2, where every occurrence of $f_2 \in F_2$ and $x_j \in X_{mx_C}$ is renamed to f_2' and x_j', respectively.
     – For every $f_{2,l} \in F_2^{(1)}$ and $f_1 \in F_1^{(n+1)}$, we add the following rule[6] to R_2':
     $$\begin{array}{rl} (*) & f_{2,l}'\ (f_1\ x_1'\ x_2' \ldots x_{n+1}') \\ = & \overline{f_1 f_{2,l}}\ x_1'\ (f_{2,1}'\ x_2') \ldots (f_{2,h}'\ x_2') \ldots\ldots (f_{2,1}'\ x_{n+1}') \ldots (f_{2,h}'\ x_{n+1}') \end{array}$$
     – For every $f_{2,l} \in F_2^{(1)}$ and $j \in [mx_Y]$, we add the following rule to R_2':
     $$f_{2,l}'\ y_j = y_{h \cdot (j-1)+l}$$
2. Now we can give the mtt $m = (F, C, R, \varrho)$ as follows:
   - $F = \{\overline{f_1 f_2}^{(hn+1)} \mid f_1 \in F_1^{(n+1)}, f_2 \in F_2^{(1)}, n \in I\!N\}$.
   - For every $f_2 \in F_2^{(1)}$, $f_1 \in F_1^{(n+1)}$, and $c \in C^{(k)}$, R contains the rule
     $$\overline{f_1 f_2}\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_h \ldots\ldots y_{h\cdot(n-1)+1} \cdots y_{h\cdot n} = nf(\Rightarrow_{u_2'}, f_2'\ (rhs_{f_1,c})).$$
   - $\varrho = nf(\Rightarrow_{u_2'}, \varrho_2'[x_1/\varrho_1])$, where ϱ_2' is constructed from ϱ_2 by renaming every occurrence of $f_2 \in F_2$ to f_2'.

Then $\tau_m = \tau_{m_1}; \tau_{m_2}$ holds (cf. the proof of Theorem 4.12 in [EV85]). □

Corollary 4.5 $MT; T \subseteq MT$ □

Again, we only state that even the result $MT; T = MT$ holds [EV85].

Example 4.6 We consider the program p_2 of Example 3.3 and $(\underline{map\ f}\ (mir\ x_1\ Nil)) \in 2\text{-}CE_{p_2}$. Thus, for the mtts m_{mir} and $m_{\underline{map\ f}}$ of Example 3.8 we construct an mtt $m_{mir;\underline{map\ f}}$ with $\tau_{m_{mir;\underline{map\ f}}} = \tau_{m_{mir}}; \tau_{m_{\underline{map\ f}}}$. Note that $mx_C = 1$, $mx_Y = 1$, $h = 1$, and $f_{2,1} = \underline{map\ f}$.

The tdtt unit $u_{\underline{map\ f}}' = (\{\underline{map\ f}'^{(1)}\}, C_{list} \cup \{mir^{(2)}, \overline{mir\underline{map\ f}}^{(2)}, x_1^{(0)}, y_1^{(0)}\}, R')$ where R' contains the rules

$$\begin{array}{lll} \underline{map\ f'}\ \ (\underline{Cons\ a}\ x_1') & = & \underline{Cons\ (f\ a)}\ (\underline{map\ f'}\ x_1') \\ \underline{map\ f'}\ \ Nil & = & Nil \\ \underline{map\ f'}\ \ (mir\ x_1'\ x_2') & = & \overline{mir\underline{map\ f}}\ x_1'\ (\underline{map\ f'}\ x_2') \\ \underline{map\ f'}\ \ y_1 & = & y_1 \end{array}$$

is able to translate right-hand sides of equations of m_{mir}.

[6] Again, in a strong sense the right-hand side of this rule is not legal.

Now the construction yields the mtt $m_{mir;\underline{map\ f}} = (F_{u_{mir;\underline{map\ f}}}, C_{list}, R_{u_{mir;\underline{map\ f}}}, \varrho_{u_{mir;\underline{map\ f}}})$ with $F_{u_{mir;\underline{map\ f}}} = \{\overline{mir\underline{map\ f}}^{(2)}\}$,

$$\begin{array}{rcl} \varrho_{u_{mir;\underline{map\ f}}} & = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\underline{map\ f'}\ (mir\ x_1\ Nil))) \\ & = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\overline{mir\underline{map\ f}}\ x_1\ (\underline{map\ f'}\ Nil))) \\ & = & \overline{mir\underline{map\ f}}\ x_1\ Nil, \end{array}$$

and $R_{u_{mir;\underline{map\ f}}}$ contains the following rules:

$$\begin{array}{rl} & \overline{mir\underline{map\ f}}\ (\underline{Cons\ a}\ x_1)\ y_1 \\ = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\underline{map\ f'}\ (rhs_{mir,\underline{Cons\ a}}))) \\ = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\underline{map\ f'}\ (\underline{Cons\ a}\ (mir\ x_1\ (\underline{Cons\ a}\ y_1))))) \\ = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\underline{Cons\ (f\ a)}\ (\underline{map\ f'}\ (mir\ x_1\ (\underline{Cons\ a}\ y_1))))) \\ = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\underline{Cons\ (f\ a)}\ (\overline{mir\underline{map\ f}}\ x_1\ (\underline{map\ f'}\ (\underline{Cons\ a}\ y_1))))) \qquad (!) \\ = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\underline{Cons\ (f\ a)}\ (\overline{mir\underline{map\ f}}\ x_1\ (\underline{Cons\ (f\ a)}\ (\underline{map\ f'}\ y_1))))) \\ = & \underline{Cons\ (f\ a)}\ (\overline{mir\underline{map\ f}}\ x_1\ (\underline{Cons\ (f\ a)}\ y_1)) \end{array}$$

$$\begin{array}{rl} & \overline{mir\underline{map\ f}}\ Nil\ y_1 \\ = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\underline{map\ f'}\ (rhs_{mir,Nil}))) \\ = & nf(\Rightarrow_{u'_{\underline{map\ f}}}, (\underline{map\ f'}\ y_1)) \\ = & y_1 \end{array}$$

Again, we can "release" f from the function name to obtain a more readable definition, which will be equivalent to the function definition of $\overline{mapmir}$ from Example 4.3. □

Clearly, the function obtained in Example 4.6 does not use intermediate results. However, classical deforestation (adjusted to mtts) applied to $map\ f\ (mir\ x_1\ Nil)$ gives the expression $\overline{mirmap}\ f\ x_1\ Nil$ and the following function $\overline{mirmap}$ as transformation result:

$$\begin{array}{lllcl} \overline{mirmap}\ f & (Cons\ a\ x_1) & y_1 & = & Cons\ (f\ a)\ (\overline{mirmap}\ f\ x_1\ (Cons\ a\ y_1)) \\ \overline{mirmap}\ f & Nil & y_1 & = & map\ f\ y_1 \end{array}$$

Thus, we see that classical deforestation could eliminate the intermediate result only partially. On the one hand, it is successful in removing the intermediate constructor "outside" the recursive call of mir by replacing $Cons\ a$ with $Cons\ (f\ a)$. On the other hand, classical deforestation cannot remove the intermediate constructor "inside" the recursive call of mir, because the function $map\ f$ does not reach the context argument of mir. In contrast to this, the step (!) in Example 4.6 shows that $\underline{map\ f}$ walks into the context argument of mir, thereby removing also the "inside" intermediate constructor.

In [Küh99] this observation was generalized: For every 2-fold composition expression $\varrho_2[x_1/\varrho_1]$ where ϱ_1 and ϱ_2 are initial expressions for an mtt unit and a tdtt unit, respectively, the composition technique can eliminate "more parts" of the intermediate result than classical deforestation. The different behavior of the transformations is caused by the different strategies to handle the "meeting" of two functions. Equation (∗) in Transformation 4.4 shows that the composition technique sends the functions of m_2 into the context arguments, whereas classical deforestation would generate a new function and would simply walk into the context arguments without changing them.

By a slight generalization of Theorem 23 in [Küh99], we can state that for linear programs, in which the tdtt unit contains exactly one function, the composition result is at least as efficient (with respect to the number of call-by-need reduction steps) as the classical deforestation result. We claim that the same holds for linear tdtts with more than one function. A comparison between the composition technique and classical deforestation for nonlinear programs is performed in [Höf99].

Summarizing the results we got so far, we know that we can transform a pair $(p, \varrho_2[x_1/\varrho_1])$ with $p \in P$ and $\varrho_2[x_1/\varrho_1] \in 2\text{-}CE_p$ into a pair (p', ϱ) with $p' \in P$ and $\varrho \in 1\text{-}CE_{p'}$, provided $\varrho_1 \in IE_u$ or $\varrho_2 \in IE_u$ for some tdtt unit u. In the next three sections we will show another technique (based on composition results for atts), which does not require one involved unit to be a tdtt unit. It will impose other sufficient conditions on the units, such that, e.g., also the 2-fold composition expression $(rev\ (rev\ x_1\ Nil)\ Nil)$ can be handled.

5 Attributed Tree Transducers

5.1 Syntax

Unlike the situation for mtts, where a function call on an input tree $(c\ t_1 \ldots t_k)$ causes only function calls on subtrees t_j (i.e., c is no more needed in the computation), instances of inherited attributes at t_j may call inherited attributes at $(c\ t_1 \ldots t_k)$ or synthesized attributes at $t_{j'}$. Thus it is necessary to keep the whole input tree t during a computation of an attributed tree transducer. The argument of an attribute will be no subtree t' of t, but the path of t that reaches t'. Instead of recursion variables $x_1, \ldots, x_k$, we will use a *path variable* z in the rules. If z is instantiated by a path p to a node of t, then the path zj is instantiated by the path pj to the j-th direct descendant of this node.

Like for our functional programs, we use a unique ranked alphabet C to build up input trees and output trees of an attributed tree transducer. In the formal definition we require that for every $c \in C$, a "complete" set of rules is specified. Again, this condition is relaxed in some examples and constructions.

Definition 5.1 An *attributed tree transducer* (for short *att*) is a tuple $a = (S, I, C, R, s_{in}, root)$, where

- S is a ranked alphabet of *synthesized attributes* with $S = S^{(1)}$.
- I is a ranked alphabet of *inherited attributes* with $I = I^{(1)}$.
- C is a ranked alphabet of *constructors* with $C^{(0)} \neq \emptyset$.
- s_{in} is a unary symbol, called the *initial synthesized attribute.*
- *root* is a unary symbol, called the *root symbol.*

The sets S, I, C, $\{s_{in}\}$, and $\{root\}$ are disjoint in pairs.

- $R = (R_c \mid c \in C \cup \{root\})$ is a family of sets of *rules* (or: *equations*):
  - R_{root} contains
    - exactly one rule
$s_{in}\ z = \varrho$ with $\varrho \in A\text{-}RHS(S, [1], \emptyset, C)$ and
    - for every $i \in I$, exactly one rule
$i\ z1 = \varrho$ with $\varrho \in A\text{-}RHS(S, [1], \emptyset, C)$.

- For every $c \in C^{(k)}$, R_c contains
  - for every $s \in S$, exactly one rule
    $s\ z = \varrho$ with $\varrho \in A\text{-}RHS(S,[k],I,C)$ and
  - for every $i \in I$ and $j \in [k]$, exactly one rule
    $i\ zj = \varrho$ with $\varrho \in A\text{-}RHS(S,[k],I,C)$.

For every $k \in \mathbb{N}$ and $I' \in \{I, \emptyset\}$, the set $A\text{-}RHS(S,[k],I',C)$ is the smallest set RHS with:

1. For every $s \in S$ and $j \in [k]$: $(s\ zj) \in RHS$.
2. For every $i \in I'$: $(i\ z) \in RHS$.
3. For every $c \in C^{(k)}$ and $\varrho_1, \ldots, \varrho_k \in RHS$: $(c\ \varrho_1 \ldots \varrho_k) \in RHS$. □

The original concept of att in [Fül81] incorporates the restriction that the right-hand side ϱ of every rule $i\ z1 = \varrho$ (with $i \in I$) in R_{root} may not contain synthesized attributes. In [Gie88] it was suggested to drop this restriction, and the resulting concept was called *full attributed tree transducer*. We follow this suggestion (cf. also [KV94]), but retain the name att.

Example 5.2 We define the att $a_{rev} = (S, I, C_{list}, R, s_{in}, root)$, where $S = \{s\}$, $I = \{i\}$, and R contains the following rules:

$$R_{root}: \begin{array}{rcl} s_{in}\ z &=& s\ z1 \\ i\ z1 &=& Nil \end{array} \qquad R_{\underline{Cons\ a}}: \begin{array}{rcl} s\ z &=& s\ z1 \\ i\ z1 &=& \underline{Cons\ a}\ (i\ z) \end{array} \qquad R_{Nil}: \begin{array}{rcl} s\ z &=& i\ z \end{array}$$

The rule sets of a_{rev} are illustrated in Figure 3 by "rule boxes". Note that we did not specify the rules in $R_{\underline{Cons\ (f\ a)}}$. The information flow of a_{rev} on $(root\ (\underline{Cons\ 1}\ (\underline{Cons\ 2}\ Nil)))$ is shown in Figure 1 by "gluing" rule boxes for $root$, $\underline{Cons\ 1}$, $\underline{Cons\ 2}$, and Nil and by instantiating the path variable at $root$ with ε. □

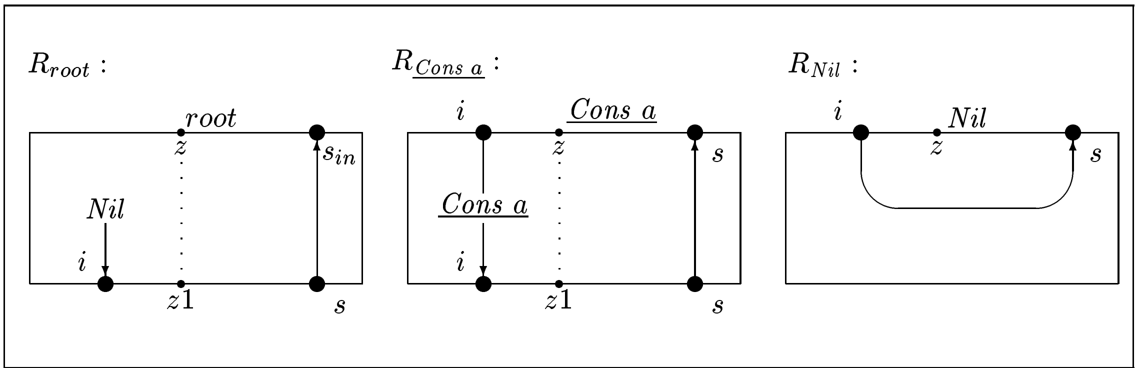

Figure 3: Rules of a_{rev}.

Example 5.3 We define the att $a_{\underline{map\ f}} = (S, I, C_{list}, R, s_{in}, root)$, where $S = \{\underline{map\ f}\}$, $I = \emptyset$, and R contains the following rules:

$$R_{root}: \quad s_{in}\ z = \underline{map\ f}\ z1 \qquad R_{\underline{Cons\ a}}: \quad \underline{map\ f}\ z = \underline{Cons\ (f\ a)}\ (\underline{map\ f}\ z1)$$
$$R_{Nil}: \quad \underline{map\ f}\ z = Nil$$

The att $a_{\underline{map\ f}}$ induces the same tree transformation (cf. Definition 5.8) as the tdtt $m_{\underline{map\ f}}$ from Example 3.8. This demonstrates that every tdtt can be easily written as att without inherited attributes. □

Let $a = (S, I, C, R, s_{in}, root)$ be an att. For every $t \in T_C$, the tree $(root\ t)$ is called *control tree*. We use the abbreviations $F = S \cup I$, $F_+ = F \cup \{s_{in}\}$, $S_+ = S \cup \{s_{in}\}$ and $C_+ = C \cup \{root\}$. For every $c \in C^{(k)}$, we define the set of *inside* and *outside attribute occurrences of* c (*with respect to* a) as $in_a(c) = \{(s\ z) \mid s \in S\} \cup \{(i\ zj) \mid i \in I, j \in [k]\}$ and $out_a(c) = \{(i\ z) \mid i \in I\} \cup \{(s\ zj) \mid s \in S, j \in [k]\}$, respectively. The sets $in_a(root) = \{(s_{in}\ z)\} \cup \{(i\ z1) \mid i \in I\}$ and $out_a(root) = \{(s\ z1) \mid s \in S\}$ are the sets of *inside* and *outside attribute occurrences of root* (*with respect to* a). For every $c \in C_+^{(k)}$ and $(f\ \eta) \in in_a(c)$, we denote the right-hand side ϱ of the rule $f\ \eta = \varrho$ in R_c by $rhs_{(f\ \eta),c}$.

Example 5.4 $in_{a_{rev}}(\underline{Cons\ a}) = \{(s\ z), (i\ z1)\}$ and $out_{a_{rev}}(\underline{Cons\ a}) = \{(i\ z), (s\ z1)\}$. □

5.2 Semantics

The set of sentential forms of an att a is the set of trees over C and over attributes, which are applied to paths of a control tree $(root\ t)$. The control tree $(root\ t)$ itself will parameterize the reduction relation (cf. Definition 5.6). With regard to the reduction relation we define the set of sentential forms for an arbitrary superset $C' \supseteq C$.

Definition 5.5 Let $a = (S, I, C, R, s_{in}, root)$ be an att. Let $t \in T_C$ and let C' be a ranked alphabet with $C \subseteq C'$. The set $SF(F, s_{in}, paths(root\ t), C')$ of *sentential forms* is the smallest set SF such that:

1. $(s_{in}\ \varepsilon) \in SF$.
2. For every $f \in F$ and $p \in (paths(root\ t) - \{\varepsilon\})$: $(f\ p) \in SF$.
3. For every $c \in C'^{(k)}$ and $t_1, \ldots, t_k \in SF$: $(c\ t_1 \ldots t_k) \in SF$.

For a control tree $(root\ t)$, the set of *attribute instances of* $(root\ t)$ (*with respect to* a) is the set $\{(s_{in}\ \varepsilon)\} \cup \{(f\ p) \mid f \in F, p \in (paths(root\ t) - \{\varepsilon\})\}$. □

Now we can define the (nondeterministic) reduction relation for atts:

Definition 5.6 Let $a = (S, I, C, R, s_{in}, root)$ be an att. Let $t \in T_C$. The *reduction relation of* a *with respect to* $(root\ t)$, denoted by $\Rightarrow_{a,(root\ t)}$, is the smallest binary relation $\Rightarrow \subseteq SF(F, s_{in}, paths(root\ t), C) \times SF(F, s_{in}, paths(root\ t), C)$, such that:

For every $t_1, t_2 \in SF(F, s_{in}, paths(root\ t), C)$, $\quad t_1 \Rightarrow t_2$, $\quad$ iff

- there is $t' \in SF(F, s_{in}, paths(root\ t), C \cup \{v\})$ with exactly one occurrence of the nullary symbol $v \notin (F_+ \cup C \cup I\!N^*)$,
- there is an attribute $f \in F_+$ and a path $p \in paths(root\ t)$ with
- $t_1 = t'[v/(f\ p)]$, and

1.
   - either $f \in S_+$,
   - there is $k \in I\!N$ and $c \in C_+^{(k)}$ with $label((root\ t), p) = c$, such that
   - $t_2 = t'[v/rhs_{(f\ z),c}[z/p]]$,
2.
   - or $f \in I$,
   - there is $p' \in paths(root\ t)$, $k \in I\!N_+$, $c \in C_+^{(k)}$, and $j \in [k]$ with $label((root\ t), p') = c$ and $p = p'j$, such that
   - $t_2 = t'[v/rhs_{(f\ zj),c}[z/p']]$. □

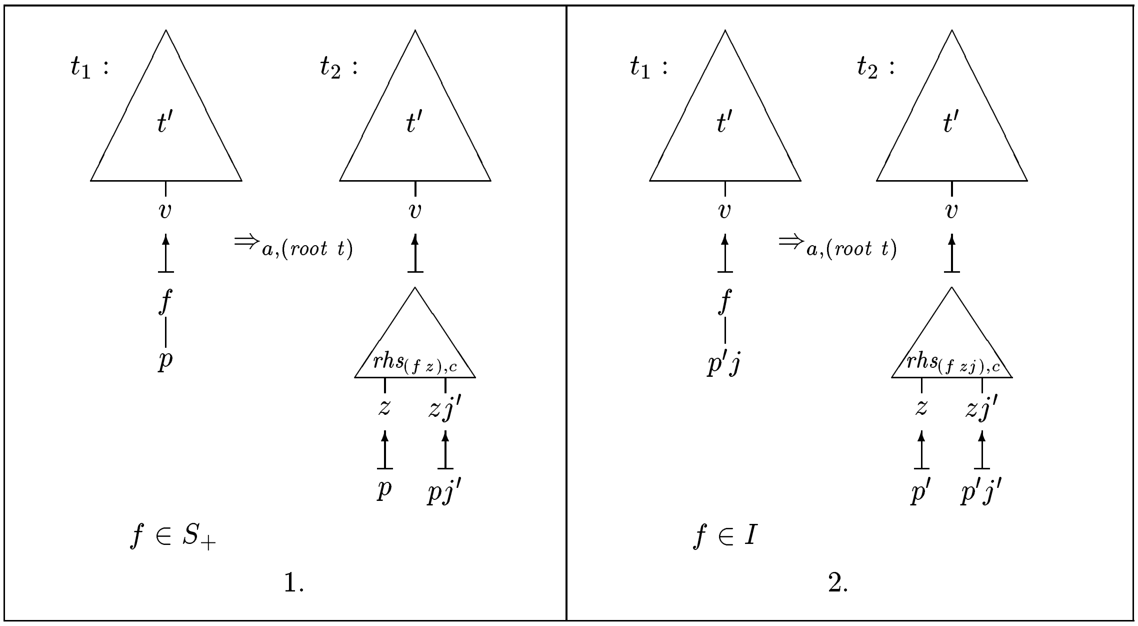

Figure 4: Reduction steps according to 1. and 2. in Definition 5.6.

If a or $(root\ t)$ are known from the context, then we drop the corresponding indices from $\Rightarrow$. The two cases of the reduction relation are illustrated in Figure 4.

Example 5.7 Let a_{rev} be the att of Example 5.2 and $(root\ (\underline{Cons\ 1}\ (\underline{Cons\ 2}\ Nil)))$ be a control tree. Then,

$$
\begin{array}{lll}
 & (s_{in}\ \varepsilon) & \\
\Rightarrow & (s\ 1) & (1.\text{ with } p = \varepsilon, \text{ and } rhs_{(s_{in}\ z),root} = s\ z1) \\
\Rightarrow & (s\ 11) & (1.\text{ with } p = 1, \text{ and } rhs_{(s\ z),\underline{Cons\ 1}} = s\ z1) \\
\Rightarrow & (s\ 111) & (1.\text{ with } p = 11, \text{ and } rhs_{(s\ z),\underline{Cons\ 2}} = s\ z1) \\
\Rightarrow & (i\ 111) & (1.\text{ with } p = 111, \text{ and } rhs_{(s\ z),Nil} = i\ z) \\
\Rightarrow & \underline{Cons\ 2}\ (i\ 11) & (2.\text{ with } p' = 11, j = 1, \text{ and } rhs_{(i\ z1),\underline{Cons\ 2}} = \underline{Cons\ 2}\ (i\ z)) \\
\Rightarrow & \underline{Cons\ 2}\ (\underline{Cons\ 1}\ (i\ 1)) & (2.\text{ with } p' = 1, j = 1, \text{ and } rhs_{(i\ z1),\underline{Cons\ 1}} = \underline{Cons\ 1}\ (i\ z)) \\
\Rightarrow & \underline{Cons\ 2}\ (\underline{Cons\ 1}\ Nil) & (2.\text{ with } p' = \varepsilon, j = 1, \text{ and } rhs_{(i\ z1),root} = Nil)
\end{array}
$$
□

Like attribute grammars, atts can be *circular*, i.e., there may exist a control tree, for which an attribute instance may depend on itself. Thus, the reduction relation of circular atts does not terminate. Nevertheless, *noncircular* atts have terminating and confluent reduction relations [Fül81] and thus every sentential form has a unique normal form. Hence, in the following we always mean noncircular atts when we talk about atts.

Definition 5.8 Let $a = (S, I, C, R, s_{in}, root)$ be an att.

- The function $\tau_a : T_C \longrightarrow T_C$ defined by $\tau_a(t) = nf(\Rightarrow_{a,(root\ t)}, (s_{in}\ \varepsilon))$, for every $t \in T_C$, is called the *tree transformation induced by a*.
- We define the class $AT = \{\tau_a \mid a \text{ is an att}\}$ of *attributed tree transformations*. □

From the literature (cf. [KV94]) we know $T \subset AT \subset MT$. The result $T \subseteq AT$ is an immediate consequence of the observation in Example 5.3. The construction for the result $AT \subseteq MT$ is provided in the next section (Transformation 6.1). The strictness of inclusions can be shown with examples.

6 Relating Functional World and Attribute World

In order to use composition techniques for atts to eliminate intermediate results in functional programs, we relate the functional world and the attribute world in this section.

6.1 From Attribute World into Functional World

The following technique that translates an att a into an mtt m is due to [Fra82, CF82]. The key idea is to simulate the synthesized attributes by functions and the inherited attributes by arguments of these functions. If a uses the inherited attributes $i_1, \ldots, i_q$ with $q \in \mathbb{N}$, then for every synthesized attribute s of a, a $(q+1)$-ary function s is used in m. The role of the inherited attribute i_l is realized by the l-th context argument of s.

For every $c \in C_+^{(k)}$, we use a recursive function $\underline{trans}_c$, which translates the right-hand side ϱ of a rule $s\ z = \varrho$ of R_c into the right-hand side $\underline{trans}_c(\varrho)$ of the new rule $s\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_q = \underline{trans}_c(\varrho)$ of m: (i) Every occurrence of $(i_l\ z)$ in ϱ is substituted by y_l. (ii) For an occurrence of $(\bar{s}\ zj)$ with $\bar{s} \in S$ in ϱ, we assume (pessimistically) that the calculation of $\bar{s}$ at the root of every input subtree t depends on every i_l at the root of t. Thus a call $(\bar{s}\ zj)$ is substituted by a call $(\bar{s}\ x_j\ t_1 \ldots t_q)$, where for every $l \in [q]$, the tree t_l is constructed by applying $\underline{trans}_c$ to the right-hand side ϱ_l of the rule $i_l\ zj = \varrho_l$ of R_c.

Our pessimistic dependency assumption can induce cycles, such that $\underline{trans}_c$ in general would not terminate. Fortunately, the construction of a right-hand side of m may be terminated, if $\underline{trans}_c$ meets a call $(\bar{s}\ zj)$ a second time, because we expect a noncircular att a. In this case we translate $(\bar{s}\ zj)$ into an arbitrary constructor. We check, whether $\underline{trans}_c$ meets a call $(\bar{s}\ zj)$ a second time, by parameterizing $\underline{trans}_c$ with the set of calls, which $\underline{trans}_c$ has already met.

Transformation 6.1 Let $a = (S, I, C, R, s_{in}, root)$ be an att. Let $I = \{i_1, \ldots, i_q\}$ with $q \in \mathbb{N}$ and let $\alpha \in C^{(0)}$ be arbitrary. We construct an mtt $m = (F, C, R', \varrho)$ as follows:

$$\begin{array}{lcl} F & = & \{s^{(q+1)} \mid s \in S\}, \\ \varrho & = & (\underline{trans}_{root,\emptyset}\ rhs_{(s_{in}\ z),root}),\ \text{and} \\ R' & = & \{s\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_q = (\underline{trans}_{c,\emptyset}\ rhs_{(s\ z),c}) \mid k \in \mathbb{N}, c \in C^{(k)},\ \text{and}\ s \in S\} \end{array}$$

For every $c \in C_+^{(k)}$, and $occ \subseteq \{(s\ zj) \mid s \in S, j \in [k]\}$, we define a function
$\underline{trans}_{c,occ} : A\text{-}RHS(S,[k],I,C) \longrightarrow M\text{-}RHS(F,C,X_k,Y_q)$.

For every $l \in [q]$,
$\underline{trans}_{c,occ}\ (i_l\ z) \quad = \quad y_l.$

For every $n \in \mathbb{N}$, $c' \in C^{(n)}$, and $\varrho_1, \ldots, \varrho_n \in A\text{-}RHS(S,[k],I,C)$,
$\underline{trans}_{c,occ}\ (c'\ \varrho_1 \ldots \varrho_n) \quad = \quad c'\ (\underline{trans}_{c,occ}\ \varrho_1) \ldots (\underline{trans}_{c,occ}\ \varrho_n).$

For every $s \in S$ and $j \in [k]$ with $(s\ zj) \notin occ$,

$$\begin{array}{lcl} \underline{trans}_{c,occ}(s\ zj) & = & s\ x_j\ (\underline{trans}_{c,occ\cup\{(s\ zj)\}}\ rhs_{(i_1\ zj),c}) \ldots \\ & & (\underline{trans}_{c,occ\cup\{(s\ zj)\}}\ rhs_{(i_q\ zj),c}). \end{array}$$

For every $s \in S$ and $j \in [k]$ with $(s\ zj) \in occ$,
$\underline{trans}_{c,occ}(s\ zj) \quad = \quad \alpha.$

Then $\tau_m = \tau_a$ holds (cf. Lemma 4.10 in [KV94]). □

Corollary 6.2 $\quad AT \subseteq MT$ □

Example 6.3 We define the att $a_{rev;rev} = (S, I, C_{list}, R, s_{in}, root)$, where $S = \{s_1, s_2\}$, $I = \{i_1, i_2\}$, and R contains the following rules:

$$\begin{array}{llcl}
R_{root}: & s_{in}\ z & = & s_1\ z1 \\
 & i_1\ z1 & = & s_2\ z1 \\
 & i_2\ z1 & = & Nil
\end{array}
\quad
\begin{array}{llcl}
R_{\underline{Cons\ a}}: & s_1\ z & = & s_1\ z1 \\
 & s_2\ z & = & \underline{Cons\ a}\ (s_2\ z1) \\
 & i_1\ z1 & = & i_1\ z \\
 & i_2\ z1 & = & i_2\ z
\end{array}
\quad
\begin{array}{llcl}
R_{Nil}: & s_1\ z & = & i_1\ z \\
 & s_2\ z & = & i_2\ z
\end{array}$$

Figure 5 illustrates the information flow of $a_{rev;rev}$ on the tree $(root\,(\underline{Cons\ 1}\ (\underline{Cons\ 2}\ Nil)))$.

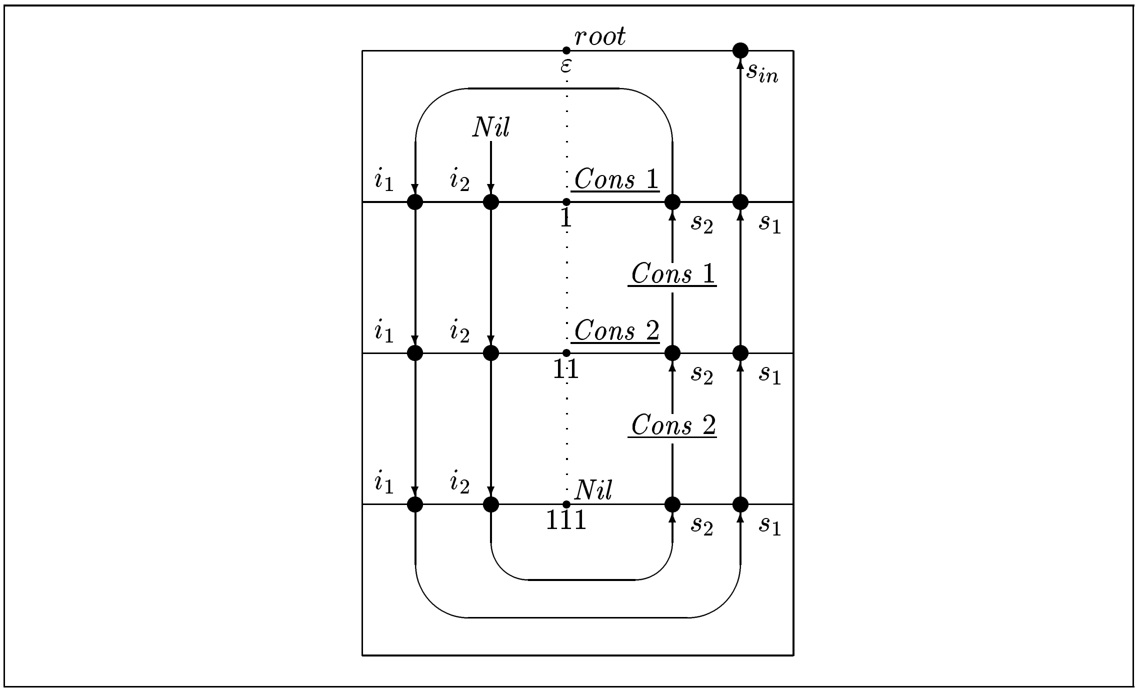

Figure 5: Information flow of $a_{rev;rev}$ on the tree $(root\ (\underline{Cons\ 1}\ (\underline{Cons\ 2}\ Nil)))$.

We translate $a_{rev;rev}$ into the mtt $m_{rev;rev} = (F, C_{list}, R', \varrho)$ with $F = \{s_1^{(3)}, s_2^{(3)}\}$ and

$$\begin{array}{rcl}
\varrho & = & \underline{trans}_{root,\emptyset}\ (s_1\ z1) \\
 & = & s_1\ x_1\ (\underline{trans}_{root,\{(s_1\ z1)\}}\ (s_2\ z1))\ (\underline{trans}_{root,\{(s_1\ z1)\}}\ Nil) \\
 & = & s_1\ x_1\ (s_2\ x_1\ (\underline{trans}_{root,\{(s_1\ z1),(s_2\ z1)\}}\ (s_2\ z1))\ (\underline{trans}_{root,\{(s_1\ z1),(s_2\ z1)\}}\ Nil))\ Nil \\
 & = & s_1\ x_1\ (s_2\ x_1\ Nil\ Nil)\ Nil.
\end{array}$$

Note here that $(\underline{trans}_{root,\{(s_1\ z1),(s_2\ z1)\}}\ (s_2\ z1))$ was replaced by the "arbitrary" output symbol Nil in order to prevent non-termination.

R' contains the following rules:

$$\begin{array}{ll}
s_1\ (\underline{Cons\ a}\ x_1)\ y_1\ y_2 = s_1\ x_1\ y_1\ y_2 & \quad s_2\ (\underline{Cons\ a}\ x_1)\ y_1\ y_2 = \underline{Cons\ a}\ (s_2\ x_1\ y_1\ y_2) \\
s_1\ Nil\ y_1\ y_2 = y_1 & \quad s_2\ Nil\ y_1\ y_2 = y_2
\end{array}$$

where the construction of the first rule is as follows:

$$\begin{array}{rcl}
s_1\ (\underline{Cons\ a}\ x_1)\ y_1\ y_2 & = & \underline{trans}_{\underline{Cons\ a},\emptyset}\ (s_1\ z1) \\
 & = & s_1\ x_1\ (\underline{trans}_{\underline{Cons\ a},\{(s_1\ z1)\}}\ (i_1\ z))\ (\underline{trans}_{\underline{Cons\ a},\{(s_1\ z1)\}}\ (i_2\ z)) \\
 & = & s_1\ x_1\ y_1\ y_2
\end{array}$$

□

Observation 6.4 Let $a = (S, I, C, R, s_{in}, root)$ be an att. We consider Transformation 6.1, which translates atts into mtts. Often the rule size of the constructed mtt can be reduced, if in a first step for every $s \in S$ a preferably small set $I_s \subseteq I$ is determined, such that for every $i \in (I - I_s)$, there is no tree $t \in T_C$, for which roughly speaking the computation of $(s\ \varepsilon)$ depends on $(i\ \varepsilon)$ (i.e., for every $i \in I_s$, there may exist a tree $t \in T_C$, on which the computation of $(s\ \varepsilon)$ depends on $(i\ \varepsilon)$). This step corresponds to the calculation of i/o-graphs in the test for *absolute* (or: *strong*) *noncircularity* of attribute grammars (cf. [KW76]; see also [CF82]). Then, in the construction of the rules for the function s, we only need $card(I_s)$ context variables. For simplicity, in Transformation 6.1 we have assumed $I_s = I$ for every $s \in S$. The improved construction is due to [CF82] and is not presented formally in our paper. □

Example 6.5 For $a_{rev;rev}$ we have $I_{s_1} = \{i_1\}$ and $I_{s_2} = \{i_2\}$ (cf., e.g., Figure 5). The improved construction delivers for $a_{rev;rev}$ the mtt $\hat{m}_{rev;rev} = (F, C_{list}, R', \varrho)$ with $F = \{s_1^{(2)}, s_2^{(2)}\}$,

$$\begin{aligned} \varrho &= \underline{trans}_{root,\emptyset}\ (s_1\ z1) \\ &= s_1\ x_1\ (\underline{trans}_{root,\{(s_1\ z1)\}}\ (s_2\ z1)) \\ &= s_1\ x_1\ (s_2\ x_1\ (\underline{trans}_{root,\{(s_1\ z1),(s_2\ z1)\}}\ Nil)) \\ &= s_1\ x_1\ (s_2\ x_1\ Nil), \end{aligned}$$

and R' contains the following rules:

$$\begin{aligned} s_1\ (\underline{Cons\ a}\ x_1)\ y_1 &= s_1\ x_1\ y_1 & \qquad s_2\ (\underline{Cons\ a}\ x_1)\ y_1 &= \underline{Cons\ a}\ (s_2\ x_1\ y_1) \\ s_1\ Nil\ y_1 &= y_1 & s_2\ Nil\ y_1 &= y_1 \end{aligned}$$

□

6.2 From Functional World into Attribute World

In [CF82] the *well-presented* restriction for mtts (which there are called primitive recursive schemes with parameters) is defined and a translation from well-presented mtts into *absolutely noncircular* atts is given. It is an open problem, whether atts and absolutely noncircular atts describe the same class of tree transformations. In [FV99] the more general *attributed-like* restriction for mtts is introduced and translations between attributed-like mtts and atts are given.

The translations in [CF82, FV99] cannot be used for our later purpose to eliminate intermediate results, because we have to construct not only atts, but atts that fulfill a special syntactic restriction in order to compose them in Section 7. Thus, we will present a refined transformation, which can handle mtts having the following restricted-use condition:

Definition 6.6 Let $m = (F, C, R, \varrho)$ be an mtt. m is called *restricted-use*, iff

(1) for every $c \in C$, $f, f', g^{(n+1)} \in F$, subtrees of the form $(g\ x_j\ \varrho_1 \dots \varrho_n)$ in $rhs_{f,c}$ and $(g\ x_j\ \varrho'_1 \dots \varrho'_n)$ in $rhs_{f',c}$, and $l \in [n]$:
   - $\varrho_l = \varrho'_l$
   - if some $y \in Y$ occurs in ϱ_l, then $f = f'$

(2) for every $g^{(n+1)} \in F$, subtrees of the form $(g\ x_1\ \varrho_1 \dots \varrho_n)$ and $(g\ x_1\ \varrho'_1 \dots \varrho'_n)$ in ϱ, and $l \in [n]$: $\varrho_l = \varrho'_l$.

We define the class $MT_{ru} = \{\tau_m \mid m \text{ is a restricted-use mtt}\}$. □

Example 6.7

- The mtt $m_{bin} = (\{bin^{(2)}\}, C_{tree}, R, (bin\ x_1\ Z))$, where R contains the rules

$$\begin{array}{llllll} bin & (S\ x_1) & y_1 & = & T\ (bin\ x_1\ (S\ y_1))\ (bin\ x_1\ (S\ y_1)) \\ bin & Z & y_1 & = & y_1 \end{array}$$

  is restricted-use, because the two occurrences of $(bin\ x_1\ \ldots)$-calls have the same context argument and occur in the same right-hand side.

- We define two variants m_{bin_1} and m_{bin_2} of the above mtt, which differ only in the rules for constructor S:

$$bin_1 \quad (S\ x_1) \quad y_1 \quad = \quad T\ (bin_1\ x_1\ (S\ y_1))\ (bin_1\ x_1\ y_1)$$

$$bin_2 \quad (S\ x_1) \quad y_1 \quad = \quad T\ (bin_2\ x_1\ (S\ y_1))\ (bin_2\ x_1\ Z)$$

  The mtts m_{bin_1} and m_{bin_2} are not restricted-use, because in both cases there are two recursive calls on the same recursion argument with different context arguments, in the right-hand side of a rule for the constructor S. □

Note that every tdtt is restricted-use.

Though we did not define well-presented mtts in this paper, we mention that the syntactic classes of restricted-use and well-presented mtts are incomparable. This is due to the facts that (i) on the one hand, well-presented mtts require the stronger condition that also context arguments in calls of different functions have to be compared, and that (ii) on the other hand, the comparison is performed by the relaxed condition $\varrho_l = \varrho'_l$. Nevertheless, a restricted-use mtt is well-presentable in the sense of [CF82], i.e., it can be transformed into a well-presented mtt. Roughly speaking, this is done by attaching to every function the context arguments of all functions in the mtt and by filling the "non-relevant" argument positions of every occurrence of a function in a new right-hand side with dummy symbols. Since only "relevant" argument positions of function calls have to be compared [CF82, FV99], the new mtt is well-presented.

We give a direct translation of a restricted-use mtt m into an att a, instead of using the above described transformation into a well-presented mtt and then the construction in [CF82]. This translation is a refinement of the translation in [CF82] and was developed independently in [Höf98] and [EM99], where it is applied to *weakly single-use* mtts (called *strongly single use restricted in the input* in the latter paper). The key idea is to simulate the functions of m and their arguments by synthesized and inherited attributes, respectively, of a. If m uses the function f with $rank(f) = n + 1$, then a synthesized attribute f and inherited attributes $y_{f,1}, \ldots, y_{f,n}$ are used in a. Note that the sets of inherited attributes for different synthesized attributes are disjoint. In contrast to this, in [CF82] the same inherited attributes $y_1, y_2, y_3, \ldots$ are used for every synthesized attribute.

Transformation 6.8 Let $m = (F, C, R, \varrho)$ be a restricted-use mtt. Let $\alpha \in C^{(0)}$ be arbitrary. We construct an att $a = (S, I, C, R', s_{in}, root)$ as follows:

$$\begin{array}{lll} S & = & \{f^{(1)} \mid f \in F\} \\ I & = & \{y_{f,l}^{(1)} \mid n \in I\!N, f \in F^{(n+1)}, l \in [n]\} \\ R' & = & (R'_c \mid c \in C_+), \end{array}$$

where for every $c \in C_+$, the set R'_c is the smallest set with:

1. For every $c \in C^{(k)}$ and $f \in F^{(n+1)}$:
$$\{f\ z = (\underline{top}_f\ rhs_{f,c})\} \cup (\underline{dec}_f\ rhs_{f,c}) \subseteq R'_c$$

2. $\{s_{in}\ z = (\underline{top}_{s_{in}}\ \varrho)\} \cup (\underline{dec}_{s_{in}}\ \varrho) \subseteq R'_{root}$

3. If there are $c \in C_+^{(k)}$, $j \in [k]$, $f \in F^{(n+1)}$, and $l \in [n]$, such that there is no rule $y_{f,l}\ zj = \ldots$ in R'_c according to 1. and 2., then add the dummy-rule $y_{f,l}\ zj = \alpha$ to R'_c.

For every $f \in F^{(q+1)}$, we define a function
$\underline{top}_f : M\text{-}RHS(F, C, X_{mx_C}, Y_q) \longrightarrow A\text{-}RHS(S, [mx_C], I, C)$.
For every $l \in [q]$,
$\underline{top}_f\ y_l \quad = \quad y_{f,l}\ z.$
For every $c \in C^{(n)}$ and $\varrho_1, \ldots, \varrho_n \in M\text{-}RHS(F, C, X_{mx_C}, Y_q)$,
$\underline{top}_f\ (c\ \varrho_1 \ldots \varrho_n) \quad = \quad c\ (\underline{top}_f\ \varrho_1) \ldots (\underline{top}_f\ \varrho_n).$
For every $g \in F^{(n+1)}$, $j \in [mx_C]$, and $\varrho_1, \ldots, \varrho_n \in M\text{-}RHS(F, C, X_{mx_C}, Y_q)$,
$\underline{top}_f\ (g\ x_j\ \varrho_1 \ldots \varrho_n) \quad = \quad g\ zj.$

For every $f \in F^{(q+1)}$, we define a function
$\underline{dec}_f : M\text{-}RHS(F, C, X_{mx_C}, Y_q) \longrightarrow \mathcal{P}(\{l = r \mid l = r \text{ is a rule of an att}\})$.
For every $l \in [q]$,
$\underline{dec}_f\ y_l \quad = \quad \emptyset.$
For every $c \in C^{(n)}$ and $\varrho_1, \ldots, \varrho_n \in M\text{-}RHS(F, C, X_{mx_C}, Y_q)$,
$\underline{dec}_f\ (c\ \varrho_1 \ldots \varrho_n) \quad = \quad \bigcup_{j \in [n]} (\underline{dec}_f\ \varrho_j).$
For every $g \in F^{(n+1)}$, $j \in [mx_C]$, and $\varrho_1, \ldots, \varrho_n \in M\text{-}RHS(F, C, X_{mx_C}, Y_q)$,
$\underline{dec}_f\ (g\ x_j\ \varrho_1 \ldots \varrho_n) \quad = \quad \{y_{g,l}\ zj = (\underline{top}_f\ \varrho_l) \mid l \in [n]\} \cup \bigcup_{l \in [n]} (\underline{dec}_f\ \varrho_l).$

a is a (well defined) att, because for every $c \in C_+^{(k)}$ by 1. and 2.:

- For every $(f\ z) \in in_a(c)$ with $f \in S_+$, exactly one rule $f\ z = \ldots$ is constructed, because there is exactly one rule for f in R_c with $c \in C$, and because ϱ is unique.
- For every $(y_{g,l}\ zj) \in in_a(c)$ with $g \in F^{(n+1)}$, $l \in [n]$, and $j \in [k]$, at most one rule $y_{g,l}\ zj = \ldots$ is constructed, because m is restricted-use, i.e., if for $c \in C$ there are occurrences of the form $(g\ x_j\ \varrho_1 \ldots \varrho_n)$ and $(g\ x_j\ \varrho'_1 \ldots \varrho'_n)$ in $rhs_{f,c}$ and $rhs_{f',c}$, respectively, for some functions f and f', then by condition (1) of Definition 6.6 we have $\varrho_l = \varrho'_l$ and, if some $y \in Y$ occurs in ϱ_l, then $f = f'$, i.e., $(\underline{top}_f\ \varrho_l) = (\underline{top}_{f'}\ \varrho'_l)$. Analogously for $c = root$, due to condition (2) of Definition 6.6. Note that here it is important that every synthesized attribute uses its own set of inherited attributes, because otherwise two occurrences $(g\ x_j\ \varrho_1 \ldots \varrho_n)$ and $(g'\ x_j\ \varrho'_1 \ldots \varrho'_{n'})$ with $g \neq g'$ would in general cause two different rules $y_l\ zj = \ldots$ for every $(y_l\ zj) \in in_a(c)$ with $l \in [min(\{n, n'\})]$.

a is noncircular, which can be shown in analogy to the proof of Claim 1.(a) following Lemma 5.12 in [EM99]. Note that there the construction is applied to *weakly single-use* mtts in the sense of Definition 6.13 below. Since we apply our construction to more general restricted-use mtts, we have to adjust the proof from [EM99] in one point.

There it is shown that if a were circular "on some input tree $t \in T_C$", then there would have to exist $c \in C^{(k)}$, $n \in \mathbb{N}_+$, $f_1, \ldots, f_n \in F$ and $j_1, \ldots, j_n \in [k]$, such that (i) for every $\nu \in$

$[n-1]$, there is an occurrence of the form $(f_\nu\ x_{j_\nu}\ \ldots (f_{\nu+1}\ x_{j_{\nu+1}}\ \ldots)\ldots)$ in the right-hand side of a rule in R_c, and (ii) there is an occurrence of the form $(f_n\ x_{j_n}\ \ldots (f_1\ x_{j_1}\ \ldots)\ldots)$ in the right-hand side of a rule in R_c. Then it is argued that the weakly single-use property of m would imply that there is a rule in R_c, the right-hand side of which contains $(f_1\ x_{j_1}\ \ldots (f_2\ x_{j_2}\ \ldots (f_n\ x_{j_n}\ \ldots (f_1\ x_{j_1}\ \ldots)\ldots)\ldots)\ldots)$, which contradicts the assumption about m being weakly single-use.

Instead, we can only use the fact that m is *restricted-use*. Condition (1) of the restricted-use property (cf. Definition 6.6), however, implies that if there is an occurrence $(f\ x_j\ \ldots)$ in the right-hand side of a rule in R_c, in a context argument of which occurs a call $(f'\ x_{j'}\ \ldots)$, then *every* occurrence $(f\ x_j\ \ldots)$ in the right-hand side of a rule in R_c contains a call $(f'\ x_{j'}\ \ldots)$ in the same context argument position. Hence, we can again conclude the existence of a right-hand side as above, where a subtree of the form $(f_1\ x_{j_1}\ \ldots (f_1\ x_{j_1}\ \ldots)\ldots)$ occurs. But, this contradicts the restricted-use property, because there are two occurrences of the form $(f_1\ x_{j_1}\ \ldots)$ that cannot have the same trees in context argument positions.

Analogous reasoning using condition (2) of the restricted-use property gives that a is also noncircular "on every control tree $root(t)$" with $t \in T_C$.

Then $\tau_a = \tau_m$ holds, as can be shown in analogy to the proof of Claim 1.(b) following Lemma 5.12 in [EM99]. □

Corollary 6.9 $MT_{ru} \subseteq AT$ □

Example 6.10 We consider the restricted-use mtt $m_{rev} = (F_{u_{rev}}, C_{list}, R_{u_{rev}}, \varrho)$, where u_{rev} is the unit from Example 3.3 (with rules $R_{u_{rev}}$ as repeated below) and $\varrho = (rev\ x_1\ Nil)$.

$$\begin{array}{lllcl} rev & (\underline{Cons\ a}\ x_1) & y_1 & = & rev\ x_1\ (\underline{Cons\ a}\ y_1) \\ rev & Nil & y_1 & = & y_1 \end{array}$$

We translate m_{rev} into the att $a'_{rev} = (S, I, C_{list}, R', s_{in}, root)$ with $S = \{rev\}$, $I = \{y_{rev,1}\}$ and R' contains the following rules according to 1. and 2. in Transformation 6.8:

$$\begin{array}{lcl} R'_{root} & \supseteq & \{s_{in}\ z = (\underline{top}_{s_{in}}\ (rev\ x_1\ Nil))\} \cup (\underline{dec}_{s_{in}}\ (rev\ x_1\ Nil)) \\ & = & \{s_{in}\ z = rev\ z1\} \cup \{y_{rev,1}\ z1 = (\underline{top}_{s_{in}}\ Nil)\} \cup (\underline{dec}_{s_{in}}\ Nil) \\ & = & \{s_{in}\ z = rev\ z1, \quad y_{rev,1}\ z1 = Nil\} \\ R'_{\underline{Cons\ a}} & \supseteq & \{rev\ z = (\underline{top}_{rev}\ (rev\ x_1\ (\underline{Cons\ a}\ y_1)))\} \cup (\underline{dec}_{rev}\ (rev\ x_1\ (\underline{Cons\ a}\ y_1))) \\ & = & \{rev\ z = rev\ z1\} \cup \{y_{rev,1}\ z1 = (\underline{top}_{rev}\ (\underline{Cons\ a}\ y_1))\} \cup (\underline{dec}_{rev}\ (\underline{Cons\ a}\ y_1)) \\ & = & \{rev\ z = rev\ z1, \quad y_{rev,1}\ z1 = \underline{Cons\ a}\ (y_{rev,1}\ z)\} \\ R'_{Nil} & \supseteq & \{rev\ z = (\underline{top}_{rev}\ y_1)\} \cup (\underline{dec}_{rev}\ y_1) \\ & = & \{rev\ z = y_{rev,1}\ z\} \end{array}$$

No rules are added according to 3. in Transformation 6.8. Thus the above occurrences of $\supseteq$ can be replaced by $=$. Note that the att a_{rev} of Example 5.2 can be obtained from a'_{rev} by a renaming of attributes. □

The resulting att a'_{rev} of Example 6.10 has a particular property: every outside attribute occurrence of a constructor c is used at most once in the right-hand sides of the rules in R'_c. This restriction is called *single-use* and is our desired syntactic restriction, which is sufficient in order to compose atts.

Definition 6.11 Let $a = (S, I, C, R, s_{in}, root)$ be an att. a is called *single-use*, iff for every $c \in C_+$, and $(f\ \eta) \in out_a(c)$, there is at most one occurrence of $(f\ \eta)$ in the right-hand sides of rules in R_c. We define the class $AT_{su} = \{\tau_a \mid a \text{ is a single-use att}\}$. □

Example 6.12 The atts a_{rev}, $a_{\underline{map\ f}}$, and $a_{rev;rev}$ of Examples 5.2, 5.3, and 6.3, respectively, are single-use. If we apply Transformation 6.8 to the restricted-use mtt m_{bin} of Example 6.7, then the resulting att a_{bin} is not single-use, because there will be two uses of the synthesized attribute *bin* at the direct descendant of the constructor S. $\square$

If we want to obtain single-use atts by Transformation 6.8, also a single-use restriction for mtts is required before translating them into atts:

Definition 6.13 Let $m = (F, C, R, \varrho)$ be an mtt.

- m is called *noncopying* (or *linear in context variables*), iff for every $c \in C$, $f \in F^{(n+1)}$, and $j \in [n]$, there is at most one occurrence of y_j in $rhs_{f,c}$.
- m is called *weakly single-use*, iff
  1. for every $c \in C^{(k)}$, $f \in F$, and $j \in [k]$, there is at most one occurrence of the form $(f\ x_j \dots)$ in the right-hand sides of rules in R_c
  2. for every $f \in F$, there is at most one occurrence of the form $(f\ x_1 \dots)$ in ϱ.
- m is called *single-use*, iff m is weakly single-use and noncopying.

We define the classes $MT_{nc} = \{\tau_m \mid m \text{ is a noncopying mtt}\}$, $MT_{wsu} = \{\tau_m \mid m \text{ is a weakly single-use mtt}\}$, and $MT_{su} = \{\tau_m \mid m \text{ is a single-use mtt}\}$. $\square$

Note that every weakly single-use mtt is restricted-use, thus $MT_{su} \subseteq MT_{wsu} \subseteq MT_{ru}$.

Example 6.14 The mtt $m_{count} = (\{count^{(2)}\}, C_{tree}, R, (count\ x_1\ Z))$, where R contains the *count*-rules from Example 3.2, is single-use, because it is noncopying and weakly single-use. So are the mtts $m_{\underline{map\ f}}$ and m_{mir} of Example 3.8, and m_{rev} of Example 6.10. The mtt m_{bin} of Example 6.7 is not weakly single-use, because there are two calls of *bin* on x_1 in $rhs_{bin,S}$. $\square$

Now we can argue that Transformation 6.8 translates every single-use mtt into a single-use att. This was already observed in [EM99], where the noncopying property is called *single use restricted in the parameters.*

Observation 6.15 Let $m = (F, C, R, \varrho)$ be a single-use mtt, i.e., weakly single-use and noncopying. Since m is in particular restricted-use, we get by Transformation 6.8 an att $a = (S, I, C, R', s_{in}, root)$.

The att a is single use, because for every $c \in C_{+}^{(k)}$:

- For every $(g\ zj) \in out_a(c)$ with $g \in S$ and $j \in [k]$, there is at most one occurrence of $(g\ zj)$ in the right-hand sides of rules in R'_c, because m is weakly single-use, i.e., there is at most one occurrence of the form $(g\ x_j\ \varrho_1 \dots \varrho_n)$ in the right-hand sides of R_c, respectively in ϱ for the case $c = root$.
- For every $(y_{f,l}\ z) \in out_a(c)$ with $f \in F^{(n+1)}$ and $l \in [n]$, there is at most one occurrence of $(y_{f,l}\ z)$ in the right-hand sides of rules in R'_c, because m is noncopying, i.e., there is at most one occurrence of y_l in $rhs_{f,c}$. Note that also here it is important that every synthesized attribute uses its own set of inherited attributes, because otherwise occurrences of y_l in $rhs_{f,c}$ and $rhs_{f',c}$ with $f \neq f'$ would cause two occurrences of $(y_l\ z)$ in the right-hand sides of rules in R'_c. $\square$

Corollary 6.16 $MT_{su} \subseteq AT_{su}$ $\square$

In order to illustrate some subtle points in Transformation 6.8, we perform another example construction.

Example 6.17 According to Transformation 6.8, the single-use mtt $\hat{m}_{rev;rev}$ of Example 6.5 can be translated into a single-use att, which can be obtained from $a_{rev;rev}$ by a renaming of attributes. We show the computation of the rules for *root* from $\varrho = s_1\ x_1\ (s_2\ x_1\ Nil)$:

$$\begin{aligned}
& \{s_{in}\ z = (\underline{top}_{s_{in}}\ (s_1\ x_1\ (s_2\ x_1\ Nil)))\} \cup (\underline{dec}_{s_{in}}\ (s_1\ x_1\ (s_2\ x_1\ Nil)))\\
= \;& \{s_{in}\ z = s_1\ z1\} \cup \{y_{s_1,1}\ z1 = (\underline{top}_{s_{in}}\ (s_2\ x_1\ Nil))\} \cup (\underline{dec}_{s_{in}}\ (s_2\ x_1\ Nil))\\
= \;& \{s_{in}\ z = s_1\ z1\} \cup \{y_{s_1,1}\ z1 = s_2\ z1\} \cup \{y_{s_2,1}\ z1 = (\underline{top}_{s_{in}}\ Nil)\} \cup (\underline{dec}_{s_{in}}\ Nil)\\
= \;& \{s_{in}\ z = s_1\ z1, \quad \underline{y_{s_1,1}\ z1} = s_2\ z1, \quad \underline{y_{s_2,1}\ z1} = Nil\}
\end{aligned}$$

Note that it is important that we have different inherited attributes $y_{s_1,1}$ and $y_{s_2,1}$ in order to prevent two different rules with the same left-hand side.

Additionally, we show the computation of the rules for *Nil* from $s_1\ Nil\ y_1 = y_1$ and $s_2\ Nil\ y_1 = y_1$:

$$\begin{aligned}
& \{s_1\ z = (\underline{top}_{s_1}\ y_1)\} \cup (\underline{dec}_{s_1}\ y_1) \cup \{s_2\ z = (\underline{top}_{s_2}\ y_1)\} \cup (\underline{dec}_{s_2}\ y_1)\\
= \;& \{s_1\ z = \underline{y_{s_1,1}\ z}, \quad s_2\ z = \underline{y_{s_2,1}\ z}\}
\end{aligned}$$

Note that it is again important that we have different inherited attributes $y_{s_1,1}$ and $y_{s_2,1}$, because otherwise the resulting att would not be single-use. □

7 Elimination of Intermediate Results Using Composition Techniques for Attributed Tree Transducers

In Section 6 we have studied the relation between the functional world and the attribute world. Here we will use the techniques from Section 6 in order to eliminate intermediate results in some cases, where classical deforestation and the composition technique for mtts (cf. Section 4) both fail. The general idea is illustrated in Figure 6 for the twofold composition of m_{rev}: In a first step we translate the two incarnations of m_{rev} into two incarnations of the att a_{rev} (Transformation 6.8 and Observation 6.15), then we compose these atts to one att $a_{rev;rev}$ (Transformation 7.1), which finally will be translated back into the mtt $\hat{m}_{rev;rev}$ (Transformation 6.1 and Observation 6.4). Note that Figure 6 does not exactly correspond to Figure 2, because the copy rules elimination (cf. Subsection 7.2) was integrated into step (ii) in Figure 2.

Unfortunately, like for mtts, there is no construction for the composition of two atts, because the class *AT* is not closed under composition [Fül81]. Thus, again we have to impose restrictions.

7.1 Composition of Restricted Attributed Tree Transducers

Two restrictions are known, under which two atts $a_1 = (S_1, I_1, C, R_1, s_{in,1}, root)$ and $a_2 = (S_2, I_2, C, R_2, s_{in,2}, root)$ can be composed to one att $a = (S, I, C, R, s_{in}, root)$: The construction in [Fül81] uses a translation technique similar to Section 4 in our paper and requires that a_2 is a tdtt. The other construction restricts a_1 to be single-use [Gan83,

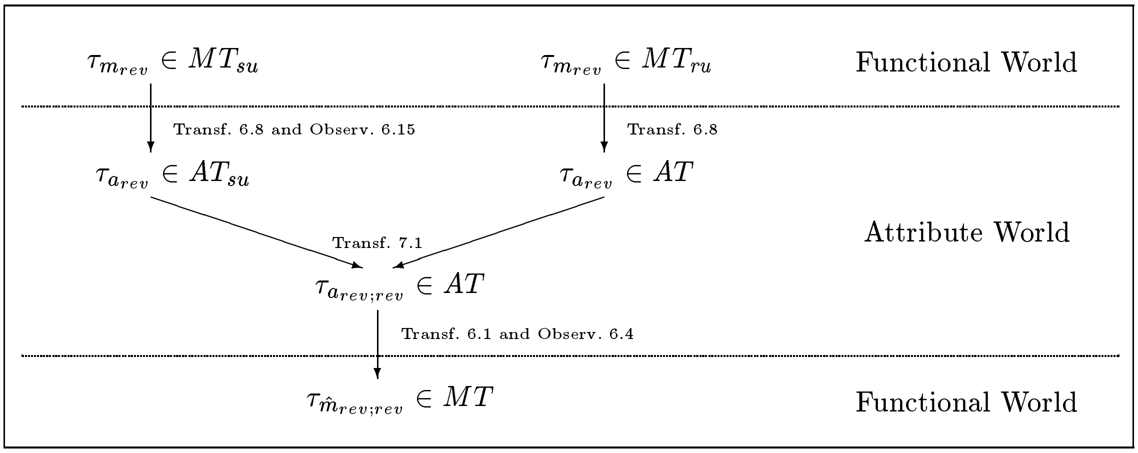

Figure 6: Transformation for the twofold composition of m_{rev}.

GG84, Gie88] and was presented in the framework of tree transducers in [Küh97] by generalizing the technique of [Fül81].

Before giving the formal construction in Transformation 7.1, we informally explain it. For the att a we take the cartesian products $S = (S_1 \times S_2) \cup (I_1 \times I_2)$ and $I = (S_1 \times I_2) \cup (I_1 \times S_2)$ as sets of attributes. For every $c \in C_+^{(k)}$, the rules in R_c are constructed by "translating" the right-hand sides of rules in $(R_1)_c$ with the help of a_2, i.e., for every rule $s_1\ z = \varrho$ (and $i_1\ zj = \varrho$, respectively) in $(R_1)_c$ with $s_1 \in S_1$ ($i_1 \in I_1$ and $j \in [k]$, respectively), ϱ serves as input tree for a_2:

- If a reduction of a_2 is started with a synthesized attribute $s_2 \in S_2$ at the root of ϱ (i.e., the right-hand side of the rule defining $(s_1\ z) \in in_{a_1}(c)$ (and $(i_1\ zj) \in in_{a_1}(c)$, respectively)), then its result is the right-hand side of the rule in R_c with left-hand side $((s_1, s_2)\ z)$ (and $((i_1, s_2)\ zj)$, respectively). Note that this corresponds to the cases (1) and (4) in Transformation 7.1 (cf. also Figures 7 and 8).

- If a reduction of a_2 is started with an inherited attribute $i_2 \in I_2$ at a leaf of ϱ, which is labeled by $(s_1'\ zj') \in out_{a_1}(c)$ with $s_1' \in S_1$ and $j' \in [k]$ (and $(i_1'\ z) \in out_{a_1}(c)$ with $i_1' \in I_1$, respectively), then its result is the right-hand side of the rule in R_c with left-hand side $((s_1', i_2)\ zj')$ (and $((i_1', i_2)\ z)$, respectively). This corresponds to the cases (2) and (3) in Transformation 7.1 (cf. also Figures 7 and 8).

If during a reduction of a_2:

- a synthesized attribute $s_2' \in S_2$ hits $(s_1'\ zj') \in out_{a_1}(c)$ with $s_1' \in S_1$ and $j' \in [k]$ (and $(i_1'\ z) \in out_{a_1}(c)$ with $i_1' \in I_1$, respectively), then it is substituted by $((s_1', s_2')\ zj')$ (and $((i_1', s_2')\ z)$, respectively). This corresponds to the cases (a) and (b) in Transformation 7.1 (cf. also Figures 7 and 8).

- an inherited attribute $i_2' \in I_2$ hits the root of ϱ (i.e., the right-hand side of the rule defining $(s_1\ z)$ (and $(i_1\ zj)$, respectively)), then it is substituted by $((s_1, i_2')\ z)$ (and $((i_1, i_2')\ zj)$, respectively). This corresponds to the cases (c) and (d) in Transformation 7.1 (cf. also Figures 7 and 8).

Transformation 7.1 Let $a_1 = (S_1, I_1, C, R_1, s_{in,1}, root)$ and $a_2 = (S_2, I_2, C, R_2, s_{in,2}, root)$ be two atts, where a_1 is single-use or a_2 does not contain inherited attributes (roughly speaking: a_2 is a tdtt). We construct an att $a = (S, I, C, R, s_{in}, root)$ as follows:

$$\begin{array}{rcl} S & = & (S_1 \times S_2) \cup (I_1 \times I_2) \\ I & = & (S_1 \times I_2) \cup (I_1 \times S_2) \\ s_{in} & = & (s_{in,1}, s_{in,2}) \\ R & = & (R_c \mid c \in C_+) \text{ with:} \end{array}$$

- For every $c \in C^{(k)}$, R_c contains the following rules:
  - For every $s_1 \in S_1$ and $(s_1\ z = \varrho) \in (R_1)_c$:

    Let $sub = [\,(s_2'\ p)/((s_1', s_2')\ zj')\ ;\ s_2' \in S_2, p \in paths(\varrho), s_1' \in S_1$, and $j' \in [k]$ with $subtree(\varrho, p) = (s_1'\ zj')$, (a)

    $(s_2'\ p)/((i_1', s_2')\ z)\ ;\ s_2' \in S_2, p \in paths(\varrho)$, and $i_1' \in I_1$ with $subtree(\varrho, p) = (i_1'\ z)$, (b)

    $(i_2'\ \varepsilon)/((s_1, i_2')\ z)\ ;\ i_2' \in I_2]$.[7] (c)

    (1) For every $s_2 \in S_2$:

    $$((s_1, s_2)\ z = nf(\Rightarrow_{a_2,\varrho}, (s_2\ \varepsilon))sub) \in R_c.^{8}$$

    (2) For every $p \in paths(\varrho)$, $s_1' \in S_1$ and $j' \in [k]$ with $subtree(\varrho, p) = (s_1'\ zj')$, and for every $i_2 \in I_2$:

    $$((s_1', i_2)\ zj' = nf(\Rightarrow_{a_2,\varrho}, (i_2\ p))sub) \in R_c.$$

    (3) For every $p \in paths(\varrho)$ and $i_1' \in I_1$ with $subtree(\varrho, p) = (i_1'\ z)$, and for every $i_2 \in I_2$:

    $$((i_1', i_2)\ z = nf(\Rightarrow_{a_2,\varrho}, (i_2\ p))sub) \in R_c.$$

    The cases (1), (2), and (3) and the substitutions according to (a), (b), and (c) are illustrated in Figure 7. Roughly speaking, the translations of a_2 on the right-hand side ϱ of a_1 start at the positions marked (1), (2), and (3) and stop at the positions marked (a), (b), and (c), because they want to leave ϱ.
  - For every $i_1 \in I_1$, $j \in [k]$ and $(i_1\ zj = \varrho) \in (R_1)_c$:

    Let $sub = [\,(s_2'\ p)/((s_1', s_2')\ zj')\ ;\ s_2' \in S_2, p \in paths(\varrho), s_1' \in S_1$, and $j' \in [k]$ with $subtree(\varrho, p) = (s_1'\ zj')$, (a)

    $(s_2'\ p)/((i_1', s_2')\ z)\ ;\ s_2' \in S_2, p \in paths(\varrho)$, and $i_1' \in I_1$ with $subtree(\varrho, p) = (i_1'\ z)$, (b)

    $(i_2'\ \varepsilon)/((i_1, i_2')\ zj)\ ;\ i_2' \in I_2]$. (d)

    (4) For every $s_2 \in S_2$:

    $$((i_1, s_2)\ zj = nf(\Rightarrow_{a_2,\varrho}, (s_2\ \varepsilon))sub) \in R_c.$$

[7] Instead of listing every "substitution pair", a compacter notation with constraints on substitution pairs is used here.

[8] Note that in a strong sense a_2 is not completely defined to work on right-hand sides of a_1, because the rules for outside attribute occurrences with respect to a_1 are missing. In [Küh97] a more formal construction is given, which uses a modified att a_2' (similar to Section 4). Further, note that in a strong sense $\Rightarrow_{a_2,\varrho}$ is not defined, because ϱ does not have the symbol $root$ at its root. These two deficiencies are overcome by the substitution sub.

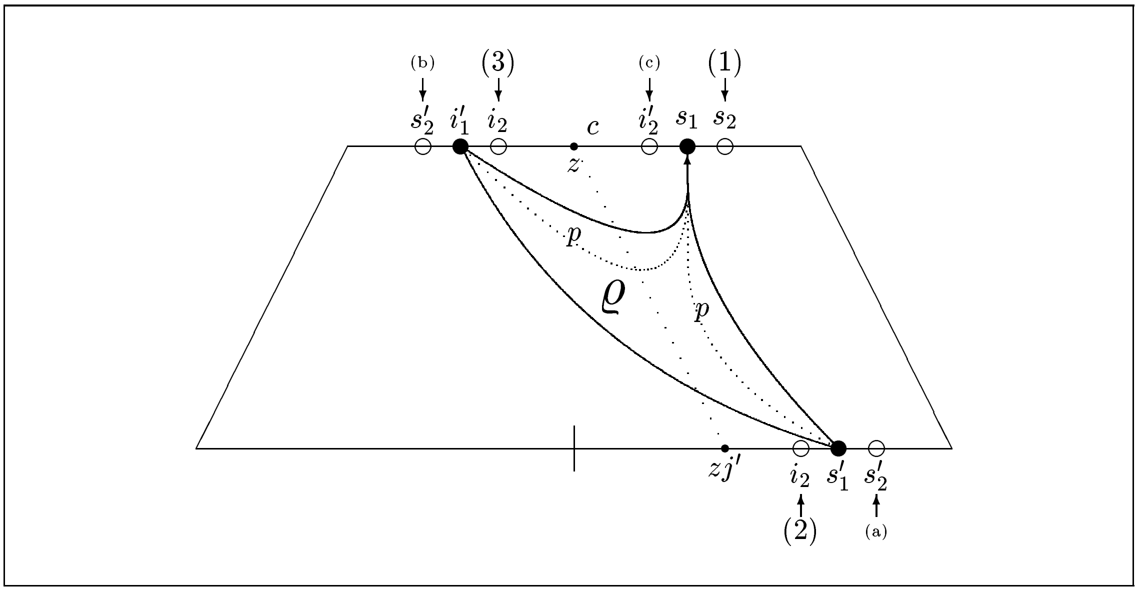

Figure 7: Situation in the cases (1), (2), and (3).

(2) For every $p \in paths(\varrho)$, $s_1' \in S_1$ and $j' \in [k]$ with $subtree(\varrho, p) = (s_1'\ zj')$, and for every $i_2 \in I_2$:

$$((s_1', i_2)\ zj' = nf(\Rightarrow_{a_2,\varrho}, (i_2\ p))sub) \in R_c.$$

(3) For every $p \in paths(\varrho)$ and $i_1' \in I_1$ with $subtree(\varrho, p) = (i_1'\ z)$, and for every $i_2 \in I_2$:

$$((i_1', i_2)\ z = nf(\Rightarrow_{a_2,\varrho}, (i_2\ p))sub) \in R_c.$$

The cases (4), (2), and (3) and the substitutions according to (a), (b), and (d) are illustrated in Figure 8.

- R_{root} contains the following rules:
  - For $(s_{in,1}\ z = \varrho) \in (R_1)_{root}$:

    Let $sub = [\,(s_2'\ p)/((s_1', s_2')\ z1)\ ;\ s_2' \in S_2, p \in paths(root\ \varrho)$, and $s_1' \in S_1$ with $subtree((root\ \varrho), p) = (s_1'\ z1)]$. (a)

    (1) $$((s_{in,1}, s_{in,2})\ z = nf(\Rightarrow_{a_2,(root\ \varrho)}, (s_{in,2}\ \varepsilon))sub) \in R_{root}.$$

    (2) For every $p \in paths(root\ \varrho)$ and $s_1' \in S_1$ with $subtree((root\ \varrho), p) = (s_1'\ z1)$, and for every $i_2 \in I_2$:

    $$((s_1', i_2)\ z1 = nf(\Rightarrow_{a_2,(root\ \varrho)}, (i_2\ p))sub) \in R_{root}.$$

    Note that a_2 has to calculate on $(root\ \varrho)$ instead of ϱ, because a_2 would also calculate on $(root\ t)$, if t is an output of a_1, and because ϱ is the "topmost part" of t.
  - For every $i_1 \in I_1$ and $(i_1\ z1 = \varrho) \in (R_1)_{root}$:

    Let $sub = [\,(s_2'\ p)/((s_1', s_2')\ z1)\ ;\ s_2' \in S_2, p \in paths(\varrho)$, and $s_1' \in S_1$ with $subtree(\varrho, p) = (s_1'\ z1)$, (a)
    $(i_2'\ \varepsilon)/((i_1, i_2')\ z1)\ ;\ i_2' \in I_2]$. (d)

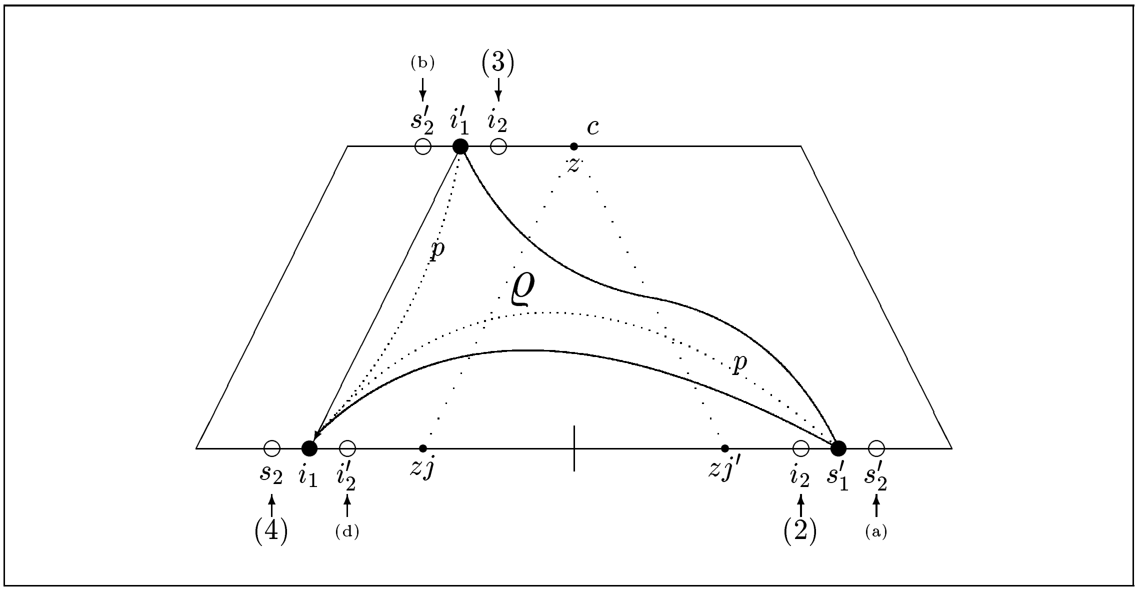

Figure 8: Situation in the cases (4), (2), and (3).

(4) For every $s_2 \in S_2$:

$$((i_1, s_2)\ z1 = nf(\Rightarrow_{a_2,\varrho}, (s_2\ \varepsilon))sub) \in R_{root}.$$

(2) For every $p \in paths(\varrho)$ and $s_1' \in S_1$ with $subtree(\varrho, p) = (s_1'\ z1)$, and for every $i_2 \in I_2$:

$$((s_1', i_2)\ z1 = nf(\Rightarrow_{a_2,\varrho}, (i_2\ p))sub) \in R_{root}.$$

- Let $\alpha \in C^{(0)}$ be arbitrary. For every $c \in C_+^{(k)}$ and for every $(f\ \eta) \in out_{a_1}(c)$, such that for every $(\lambda = \varrho) \in (R_1)_c$ and $p \in paths(\varrho)$ the condition $subtree(\varrho, p) \neq (f\ \eta)$ holds, R_c contains for every $i_2 \in I_2$ a (dummy) rule $(f, i_2)\ \eta = \alpha$.

a is a (well defined) att, because for every $c \in C_+^{(k)}$:

- For every $s_2 \in S_2$ and $(f_1\ \eta) \in in_{a_1}(c)$, exactly one rule with left-hand side $(f_1, s_2)\ \eta$ is in R_c, because there is exactly one rule with left-hand side $f_1\ \eta$ in $(R_1)_c$.
- For unrestricted atts there may exist $i_2 \in I_2$ and $(f_1\ \eta) \in out_{a_1}(c)$, such that there are different rules with left-hand side $(f_1, i_2)\ \eta$ in R_c. This is caused by the fact that an outside attribute occurrence of $out_{a_1}(c)$ may occur more than once in $(R_1)_c$. For every such occurrence at a path p of a right-hand side ϱ and for every $i_2 \in I_2$, a rule with left-hand side $(f_1, i_2)\ \eta$ is constructed, where its right-hand side is the normal form of $(i_2\ p)$ with respect to $\Rightarrow_{a_2,\varrho}$. Clearly, for different occurrences the normal form may be different. But, since a_1 is single-use or a_2 is a tdtt, this phenomenon is impossible.

a is noncircular as reasoned informally in the following by using the noncircularity of a_1 and a_2, respectively. Since a_1 is noncircular, for every input tree t an intermediate result $\bar{t}$ is calculated, which roughly speaking is built up from occurrences of right-hand sides (like ϱ in Figures 7 and 8) of a_1. Since a_2 is noncircular as well, no attribute instance of $(root\ \bar{t})$ with respect to a_2 depends on itself. Since, roughly speaking, exactly those attribute instances of $(root\ \bar{t})$ with respect to a_2 will show up as attribute instances of $(root\ t)$ with respect to a, which occur at the root or the leafs (like the hollow circles in Figures 7 and 8) of occurrences of right-hand sides of a_1 in $(root\ \bar{t})$, the set of attribute instances of $(root\ t)$ with respect to a can be considered as a subset of the set of attribute instances of $(root\ \bar{t})$ with respect to a_2, obeying the same dependencies. Hence, also no attribute instance of $(root\ t)$ with respect to a depends on itself. A more formal reasoning can be found in the proof of Lemma 6.4 in [Küh97].

Then $\tau_a = \tau_{a_1}; \tau_{a_2}$ holds (cf. the proofs of Theorem 4.3 in [Fül81] and Lemma 6.4 in [Küh97]). □

Corollary 7.2 $AT_{su}; AT \subseteq AT$ and $AT; T \subseteq AT$ □

We only state that even the results $AT = AT_{su}; AT = AT; T$ and $AT_{su} = AT_{su}; AT_{su}$ hold [Fül81, Gie88, Küh97].

Example 7.3 We recall the single-use att $a_{rev} = (S, I, C_{list}, R, s_{in}, root)$ from Example 5.2, where $S = \{s\}$, $I = \{i\}$, and R contains the following rules:

$$R_{root}: \begin{array}{rcl} s_{in}\ z & = & s\ z1 \\ i\ z1 & = & Nil \end{array} \qquad R_{\underline{Cons\ a}}: \begin{array}{rcl} s\ z & = & s\ z1 \\ i\ z1 & = & \underline{Cons\ a}\ (i\ z) \end{array} \qquad R_{Nil}: \begin{array}{rcl} s\ z & = & i\ z \end{array}$$

The composition of a_{rev} with itself (for consistency to Transformation 7.1 we use indices 1 and 2 for the attributes of the first and second incarnation of a_{rev}, respectively) delivers the att $a'_{rev;rev} = (S', I', C_{list}, R', (s_{in,1}, s_{in,2}), root)$, where $S' = \{(s_1, s_2), (i_1, i_2)\}$, $I' = \{(i_1, s_2), (s_1, i_2)\}$, and R' contains the following rules. The numbers behind the calculations refer to the cases in Transformation 7.1.

$$\begin{array}{lcll}
R'_{root}: & & (s_{in,1}, s_{in,2})\ z & \\
& = & nf(\Rightarrow_{a_{rev},(root\ (s_1\ z1))}, (s_{in,2}\ \varepsilon))[(s_2\ 1)/((s_1, s_2)\ z1)] & (1) \\
& = & (s_2\ 1)[(s_2\ 1)/((s_1, s_2)\ z1)] & \\
& = & (s_1, s_2)\ z1 & \\
& & (s_1, i_2)\ z1 & \\
& = & nf(\Rightarrow_{a_{rev},(root\ (s_1\ z1))}, (i_2\ 1))[(s_2\ 1)/((s_1, s_2)\ z1)] & (2) \\
& = & Nil & \\
& & (i_1, s_2)\ z1 & \\
& = & nf(\Rightarrow_{a_{rev},Nil}, (s_2\ \varepsilon))[(i_2\ \varepsilon)/((i_1, i_2)\ z1)] & (4) \\
& = & (i_2\ \varepsilon)[(i_2\ \varepsilon)/((i_1, i_2)\ z1)] & \\
& = & (i_1, i_2)\ z1 &
\end{array}$$

$$\begin{array}{lcll}
R'_{Nil}: & & (s_1, s_2)\ z & \\
& = & nf(\Rightarrow_{a_{rev},(i_1\ z)}, (s_2\ \varepsilon))[(s_2\ \varepsilon)/((i_1, s_2)\ z), (i_2\ \varepsilon)/((s_1, i_2)\ z))] & (1) \\
& = & (s_2\ \varepsilon)[(s_2\ \varepsilon)/((i_1, s_2)\ z), (i_2\ \varepsilon)/((s_1, i_2)\ z))] & \\
& = & (i_1, s_2)\ z & \\
& & (i_1, i_2)\ z & \\
& = & nf(\Rightarrow_{a_{rev},(i_1\ z)}, (i_2\ \varepsilon))[(s_2\ \varepsilon)/((i_1, s_2)\ z), (i_2\ \varepsilon)/((s_1, i_2)\ z))] & (3) \\
& = & (i_2\ \varepsilon)[(s_2\ \varepsilon)/((i_1, s_2)\ z), (i_2\ \varepsilon)/((s_1, i_2)\ z))] & \\
& = & (s_1, i_2)\ z &
\end{array}$$

$$
\begin{array}{rll}
R'_{\underline{Cons\ a}}: & (s_1, s_2)\ z & \\
= & nf(\Rightarrow_{a_{rev},(s_1\ z1)}, (s_2\ \varepsilon))[(s_2\ \varepsilon)/((s_1, s_2)\ z1), (i_2\ \varepsilon)/((s_1, i_2)\ z))] & (1)\\
= & (s_2\ \varepsilon)[(s_2\ \varepsilon)/((s_1, s_2)\ z1), (i_2\ \varepsilon)/((s_1, i_2)\ z))] & \\
= & (s_1, s_2)\ z1 & \\
 & (s_1, i_2)\ z1 & \\
= & nf(\Rightarrow_{a_{rev},(s_1\ z1)}, (i_2\ \varepsilon))[(s_2\ \varepsilon)/((s_1, s_2)\ z1), (i_2\ \varepsilon)/((s_1, i_2)\ z))] & (2)\\
= & (i_2\ \varepsilon)[(s_2\ \varepsilon)/((s_1, s_2)\ z1), (i_2\ \varepsilon)/((s_1, i_2)\ z))] & \\
= & (s_1, i_2)\ z & \\
 & (i_1, s_2)\ z1 & \\
= & nf(\Rightarrow_{a_{rev},(\underline{Cons\ a}\ (i_1\ z))}, (s_2\ \varepsilon))[(s_2\ 1)/((i_1, s_2)\ z), (i_2\ \varepsilon)/((i_1, i_2)\ z1))] & (4)\\
= & (s_2\ 1)[(s_2\ 1)/((i_1, s_2)\ z), (i_2\ \varepsilon)/((i_1, i_2)\ z1))] & \\
= & (i_1, s_2)\ z & \\
 & (i_1, i_2)\ z & \\
= & nf(\Rightarrow_{a_{rev},(\underline{Cons\ a}\ (i_1\ z))}, (i_2\ 1))[(s_2\ 1)/((i_1, s_2)\ z), (i_2\ \varepsilon)/((i_1, i_2)\ z1))] & (3)\\
= & (\underline{Cons\ a}\ (i_2\ \varepsilon))[(s_2\ 1)/((i_1, s_2)\ z), (i_2\ \varepsilon)/((i_1, i_2)\ z1))] & \\
= & \underline{Cons\ a}\ ((i_1, i_2)\ z1) &
\end{array}
$$

Note that $a_{rev;rev}$ of Example 6.3, which was illustrated in Figure 5, can be obtained from $a'_{rev;rev}$ by a renaming of attributes. Now we have closed the gap between the Examples 6.10 and 6.5. The resulting mtt $\hat{m}_{rev;rev}$ represents a functional program that does not construct intermediate results! □

The statement $AT_{su}; AT \subseteq AT$ of Corollary 7.2 is the heart of transformations like in Figure 6. We have seen in Example 7.3 that sometimes it can eliminate intermediate results, when classical deforestation and the composition technique for mtts fail. On the other hand, the statement $AT; T \subseteq AT$ of Corollary 7.2 is not relevant with respect to the "pure" elimination of intermediate results, because $AT \subseteq MT$ and hence in this case the direct construction of the result $MT; T \subseteq MT$ can be used. It can become relevant, if efficiencies are compared formally, because the transformation results of the direct and the indirect constructions may be syntactically different.

By putting the statements of Corollaries 6.16, 6.9, 7.2 and 6.2 together, we can formulate the theoretical background for the elimination of intermediate results using composition techniques for atts as follows.

Theorem 7.4 $MT_{su}; MT_{ru} \subseteq MT$ □

Since also well-presented mtts can be translated into atts (cf. Subsection 6.2), we additionally get the following result, where MT_{wp} denotes the class of tree transformations induced by well-presented mtts.

Theorem 7.5 $MT_{su}; MT_{wp} \subseteq MT$ □

7.2 Elimination of Copy Rules

In this short subsection we would like to demonstrate a problem of the composition technique and its possible solution only by means of an example:

Example 7.6 The mtt $\hat{m}_{rev;rev}$ obtained in Example 6.5 is not as efficient as we expected. It performs unnecessary reduction steps to compute the identity function on lists. They are caused by the so called *copy rules* of $a_{rev;rev}$, which do not produce output symbols (cf. Figure 5). In our example a *copy rules elimination* [CDPR98] can detect that for every list l,

1. $(s_1\ 1) \Rightarrow^*_{a_{rev;rev},(root\ l)} (i_1\ 1)$, and

2. $(i_2\ p) \Rightarrow^*_{a_{rev;rev},(root\ l)} Nil$, if $label((root\ l), p) = Nil$.

Hence, we have $(s_{in}\ \varepsilon) \Rightarrow^*_{a_{rev;rev},(root\ l)} (s_2\ 1)$ and $(s_2\ p) \Rightarrow^*_{a_{rev;rev},(root\ l)} Nil$, if $label((root\ l), p) = Nil$, and the copy rules elimination produces the att $a_{id} = (S, \emptyset, C_{list}, R, s_{in}, root)$ with $S = \{s_2^{(1)}\}$, and R contains the following rules:

$$R_{root}: \quad s_{in}\ z \;=\; s_2\ z1 \qquad R_{\underline{Cons\ a}}: \quad s_2\ z \;=\; \underline{Cons\ a}\ (s_2\ z1) \qquad R_{Nil}: \quad s_2\ z \;=\; Nil$$

Note that a_{id} "is" a tdtt m_{id} that represents the expected identity function id on lists (cf. Figure 2).

Note that the copy rules elimination can be unsound in the framework of a lazy functional programming language when in particular infinite data objects are handled: if for instance l represents a list with infinitely many elements and $h\ (Cons\ a\ x) = Nil$ is an additional function definition, then the evaluation of $h\ (rev\ (rev\ l\ Nil)\ Nil)$ does not terminate, whereas the evaluation of $h\ (id\ l)$ delivers the output Nil after two reduction steps. $\square$

In [Voi01] the related concept of *copy-states* on the level of mtts is considered, and a decision algorithm (based on a fixpoint construction) is given that allows the elimination of all copy-states of an mtt. There, also the practical implications of integrating such a program transformation, which may turn non-terminating programs into terminating ones, into an optimizing compiler are discussed.

8 Comparison of Deforestation Methods

In this section we want to compare the transformation techniques presented in this paper and classical deforestation on examples. Our comparison criterion is the ability to eliminate intermediate data structures.

Note that when applicable, the mtt and att composition techniques always eliminate the intermediate result. In the case of classical deforestation, it can also happen that the intermediate data structure is only partially eliminated. Hence, we consider 12 different cases as illustrated in Figure 9. We say that classical deforestation does *not* eliminate the intermediate data structure, if with growing size of the input tree, the ratio between the size of the eliminated part and the overall size of the intermediate result tends asymptotically towards zero. We say that classical deforestation *does* eliminate the intermediate result, if the non-eliminated part of the intermediate data structure has a constant size, independent of the input tree.

Example composition expressions corresponding to the different cases in Figure 9 are given in the following table, where $doub'$ is a variant of $doub$ from Example 3.2, without the rule at T.

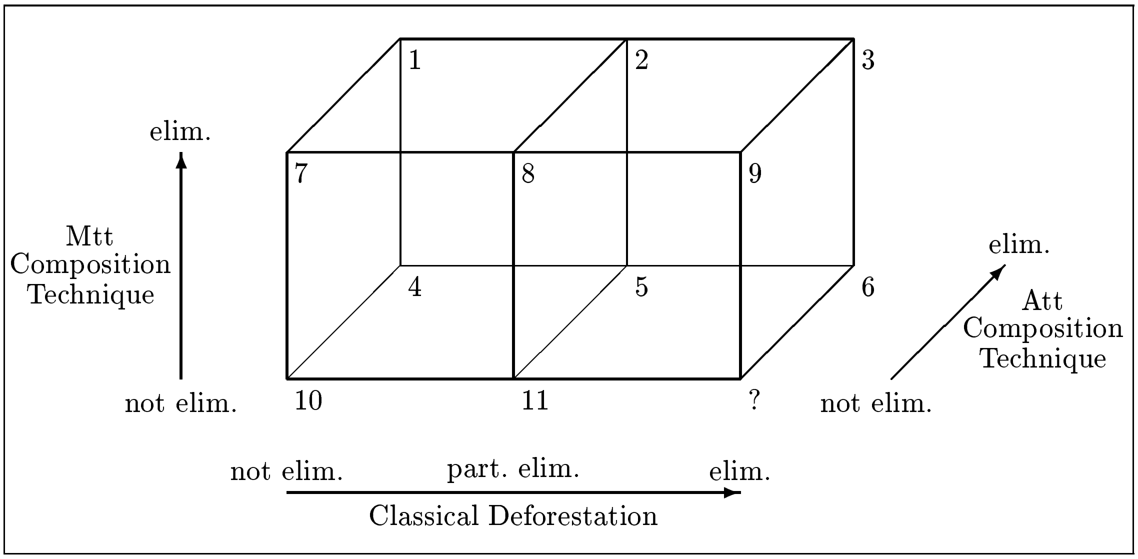

Figure 9: Different elimination abilities by different transformation techniques.

1 : $\underline{map\ f}\ (rev\ x_1\ Nil)$	(cf. the introduction)
2 : $\underline{map\ f}\ (mir\ x_1\ Nil)$	(cf. Example 4.6)
3 : $mir\ (\underline{map\ f}\ x_1)\ Nil$	(cf. Example 4.3)
4 : $rev\ (rev\ x_1\ Nil)\ Nil$	(cf. Examples 6.10, 7.3 and 7.6)
5 : $rev\ (mir\ x_1\ Nil)\ Nil$	
6 : $rev\ (app\ x_1\ (\underline{Cons\ 1}\ Nil))\ Nil$	
7 : $doub\ (bin_1\ x_1\ Z)$	
8 : $doub\ (bin_2\ x_1\ Z)$	
9 : $bin_1\ (doub'\ x_1)\ Z$	
10 : $count\ (bin_1\ x_1\ Z)\ Z$	
11 : $count\ (bin_2\ x_1\ Z)\ Z$	

For the examples 7, 8, 10 and 11, note that classical deforestation does not reach accumulating arguments. Hence, only the T-constructors of the intermediate data structures produced by bin_1 and bin_2, respectively, will be eliminated (cf. Example 6.7). Since for every $n \in \mathbb{N}$, the intermediate result of $bin_1\ (S^n\ Z)\ Z$ (and $bin_2\ (S^n\ Z)\ Z$, respectively) contains $2^n - 1$ occurrences of T, 2^n occurrences of Z, and $n2^{n-1}$ occurrences of S (and $2^n - 1$ occurrences of S, respectively), the asymptotic behavior for these examples with classical deforestation is as classified in Figure 9.

A question mark remains in Figure 9, namely for a composition expression, for which classical deforestation is successful and our proposed techniques are not. We cannot provide such a composition expression, which however does not mean that there are no realistic programs, for which classical deforestation is superior. Besides cases, where our techniques are not applicable, because the functions to be composed are not in the classes of tree transducers that can be handled, consider the classical example $e = app\ (app\ x_1\ y_1)\ z_1$ with three free variables. Classical deforestation transforms this into a more efficient expression, essentially proving the associativity of app. With the att composition technique we cannot handle such expressions, because they do not fit into our framework of initial and composition expressions (cf. Definition 3.6). In [Küh99] for compositions of mtts and tdtts (cf. Transformations 4.1 and 4.4), a more general framework is used, in which also expressions with more than one free variable, such as $\underline{map\ f}$ $(rev\ x_1\ y_1)$, can be handled. Similarly, the direct composition construction developed in [Voi01] can be used to transform the expression e into a form equally efficient as the classical deforestation result.

9 Future Work

We have presented informal comparisons of classical deforestation and composition techniques for certain classes of mtts, where we have considered the pure elimination of intermediate results as success. Instead of giving examples, it would be nice to specify classes of programs for the vertices in Figure 9. Also, a more formal analysis (counting call-by-need reduction steps or using more exact efficiency measures) would be useful, in order to determine, when transformations should be applied, and in order to compare speedups realized by different transformations. For the direct composition techniques of mtts and tdtts, such a study has begun in [Küh99] for linear and in [Höf99] for nonlinear programs, respectively. For the composition technique by indirection via atts, work is in progress [Mal01]. The direct composition construction for restricted mtts in the case that neither of the two composition partners is a tdtt [Voi01] might also favor the efficiency comparison with classical deforestation. Furthermore, it would be interesting to integrate other techniques for the elimination of intermediate results into such a comparison, e.g., *shortcut deforestation* [GLP93, Gil96], *program calculation* techniques based on *recursion operators* [Mal89, MFP91, SF93, BdM96, HIT96, Jür00], and *supercompilation* [Tur86, SGJ96, SS99].

The theory provides other classes of tree transducers that might be employed for the elimination of intermediate results. For example, *modular tree transducers* [EV91] are used in [KGK01] to transform non-accumulative functional programs into accumulative ones. It could be fruitful to consider *macro attributed tree transducers* [KV94] in connection with higher-order functional programs. Similar to Transformation 6.8 in the present paper, where functions with context arguments of first-order type are translated into atts, certain functions that use context parameters of higher-order type can be transformed into macro attributed tree transducers. Together with composition constructions for macro attributed tree transducers [Küh98], we might obtain interesting transformation techniques for higher-order functions.

The transformations considered in this paper have been implemented in a rudimentary program transformation system [HVM$^+$01]. In order to benefit from these transformations in a real programming environment, we would like to integrate them into a full-scale functional compiler. This would require the overhead of an analysis phase to detect tree transducers in source programs, but advantageously the transformations themselves are local and do not involve a bookkeeping of the transformation history that classical deforestation needs to ensure termination.

Acknowledgment

We would like to thank Matthias Höff (June 22, 1974 – February 21, 2000) for his contribution to the results in Section 6 of this paper.

References

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. Assoc. Comput. Mach.*, 24:44–67, 1977.

[BdM96] R.S. Bird and O. de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1996.

[CDPR98] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Symbolic composition. Technical Report 3348, Unité de recherche INRIA Rocquencourt, 1998.

[CDPR99] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: a deforestation case-study. In G. Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming – PPDP'99, Paris, France, Proceedings*, volume 1702 of *LNCS*, pages 360–377. Springer-Verlag, September 1999.

[CF82] B. Courcelle and P. Franchi–Zannettacci. Attribute grammars and recursive program schemes. *Theoret. Comput. Sci.*, 17:163–191, 235–257, 1982.

[Chi94] W.N. Chin. Safe fusion of functional expressions II: Further improvements. *J. Funct. Prog.*, 4:515–555, 1994.

[EM99] J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inform. and Comput.*, 154:34–91, 1999. Also: Leiden University Technical Report, 98-09, August 1998.

[Eng75] J. Engelfriet. Bottom–up and top–down tree transformations — a comparison. *Math. Syst. Theory*, 9:198–231, 1975.

[Eng80] 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.

[Eng81] J. Engelfriet. Tree transducers and syntax directed semantics. Technical Report Memorandum 363, Technische Hogeschool Twente, 1981.

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31:71–145, 1985.

[EV91] J. Engelfriet and H. Vogler. Modular tree transducers. *Theoret. Comput. Sci.*, 78:267–304, 1991.

[Fra82] P. Franchi–Zannettacci. *Attributs semantiques et schemas de programmes.* PhD thesis, Universite de Bordeaux I, 1982.

[Fül81] Z. Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981.

[FV98] 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.

[FV99] Z. Fülöp and H. Vogler. A characterization of attributed tree transformations by a subclass of macro tree transducers. *Theory of Comput. Syst.*, 32:649–676, 1999.

[Gan83] H. Ganzinger. Increasing modularity and language–independency in automatically generated compilers. *Sci. of Comput. Prog.*, 3:223–278, 1983.

[GG84] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Symposium on Compiler Construction, Montreal, Canada, Proceedings*, volume 19 of *ACM SIGPLAN Notices*, pages 157–170, June 1984.

[Gie88] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.

[Gil96] A. Gill. *Cheap deforestation for non-strict functional languages.* PhD thesis, University of Glasgow, 1996.

[GLP93] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture – FPCA'93, Copenhagen, Denmark, Proceedings*, pages 223–231. ACM Press, June 1993.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *International Conference on Functional Programming, Philadelphia, Pennsylvania, Proceedings*, pages 73–82. ACM Press, May 1996.

[HJ92] G.W. Hamilton and S.B. Jones. Extending deforestation for first order functional programs. In R. Heldal, C. K. Holst, and P. L. Wadler, editors, *1991 Glasgow Workshop on Functional Programming*, Series of Workshops in Computing, pages 134–145. Springer-Verlag, 1992.

[Höf98] M. Höff. Vergleich der Berechnungsstärken von Klassen eingeschränkter Tree Transducer. Großer Beleg, Dresden University of Technology, 1998.

[Höf99] M. Höff. Vergleich von Verfahren zur Elimination von Zwischenergebnissen bei funktionalen Programmen. Master's thesis, Dresden University of Technology, Department of Computer Science, September 1999.

[HVM+01] M. Höff, R. Vater, A. Maletti, A. Kühnemann, and J. Voigtländer. Tree transducer based program transformations for Haskell+, March 2001. Progress report, Dresden University of Technology.
http://wwwtcs.inf.tu-dresden.de/~voigt/hpgreport.ps.gz.

[Joh87] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture – FPCA'87, Portland, Proceedings*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, September 1987.

[Jür00] C. Jürgensen. A formalization of hylomorphism based deforestation with an application to an extended typed λ-calculus. Technical Report TUD-FI00-13, Dresden University of Technology, 2000.
http://wwwtcs.inf.tu-dresden.de/~claus/publications/tud00-13.ps.gz.

[KGF01] K. Kakehi, R. Glück, and Y. Futamura. A note on deforesting parameters of accumulating maps. Manuscript, Waseda University, Tokyo, Japan, 2001.

[KGK01] A. Kühnemann, R. Glück, and K. Kakehi. Relating accumulative and non-accumulative functional programs. In A. Middeldorp, editor, *12th International Conference on Rewriting Techniques and Applications – RTA'01, Utrecht, The Netherlands, Proceedings*, volume 2051 of *LNCS*, pages 154–168. Springer-Verlag, May 2001.

[Knu68] D.E. Knuth. Semantics of context–free languages. *Math. Syst. Theory*, 2:127–145, 1968. Corrections in *Math. Syst. Theory*, 5:95-96, 1971.

[Küh97] A. Kühnemann. *Berechnungsstärken von Teilklassen primitiv–rekursiver Programmschemata.* PhD thesis, Dresden University of Technology, 1997. Shaker Verlag, Aachen.

[Küh98] A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In V. Arvind and R. Ramanujam, editors, *Foundations of Software Technology & Theoretical Computer Science – FST & TCS'98, 18th Conference, Chennai, India, Proceedings*, volume 1530 of *LNCS*, pages 146–157. Springer-Verlag, December 1998.

[Küh99] A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In A. Middeldorp and T. Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming – FLOPS'99, Tsukuba, Japan, Proceedings*, volume 1722 of *LNCS*, pages 114–130. Springer-Verlag, November 1999.

[KV94] A. Kühnemann and H. Vogler. Synthesized and inherited functions — a new computational model for syntax–directed semantics. *Acta Informatica*, 31:431–477, 1994.

[KW76] K. Kennedy and S.K. Warren. Automatic generation of efficient evaluators for attribute grammars. In *Principles of Programming Languages, Atlanta, Georgia, Proceedings*, pages 32–49, January 1976.

[Mal89] G. Malcolm. Homomorphisms and promotability. In *Mathematics of Program Construction, Groningen, The Netherlands, Proceedings*, volume 375 of *LNCS*, pages 335–347. Springer-Verlag, June 1989.

[Mal01] A. Maletti. Efficiency analysis for the elimination of intermediate results in functional programs by compositions of attributed tree transducers. Bachelor's thesis, Dresden University of Technology, in progress, 2001.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture – FPCA'91, 5th ACM Conference, Cambridge, MA, USA, Proceedings*, volume 523 of *LNCS*, pages 124–244. Springer–Verlag, August 1991.

[Rou70] W.C. Rounds. Mappings and grammars on trees. *Math. Syst. Theory*, 4:257–287, 1970.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Functional Programming Languages and Computer Architecture – FPCA'93, Copenhagen, Denmark, Proceedings*, pages 233–242. ACM Press, June 1993.

[SGJ96] M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. *J. Funct. Prog.*, 6:811–838, 1996.

[SS99] J.P. Secher and M.H. Sørensen. On perfect supercompilation. In *Perspectives of System Informatics, Novosibirsk, Russia, Proceedings*, volume 1755 of *LNCS*, pages 113–127. Springer-Verlag, July 1999.

[Tha70] J.W. Thatcher. Generalized^2 sequential machine maps. *J. Comput. Syst. Sci.*, 4:339–367, 1970.

[Tur86] V.F. Turchin. The concept of a supercompiler. *ACM Trans. on Prog. Lang. and Systems*, 8:292–325, 1986.

[VK01] J. Voigtländer and A. Kühnemann. Composition of functions with accumulating parameters. Technical Report TUD-FI01-08, Dresden University of Technology, 2001.
http://wwwtcs.inf.tu-dresden.de/~voigt/TUD-FI01-08.ps.gz.

[Voi01] J. Voigtländer. Composition of restricted macro tree transducers. Master's thesis, Dresden University of Technology, March 2001.
http://wwwtcs.inf.tu-dresden.de/~voigt/mscthesis.ps.gz (revised version).

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoret. Comput. Sci.*, 73:231–248, 1990.