Relating Accumulative and Non-Accumulative Functional Programs

Armin Kühnemann[1,⋆], Robert Glück[2,⋆⋆], and Kazuhiko Kakehi[2,⋆⋆⋆]

[1]Institute for Theoretical Computer Science, Department of Computer Science,
Dresden University of Technology, D–01062 Dresden, Germany
`kuehne@orchid.inf.tu-dresden.de`

[2]Institute for Software Production Technology,
Waseda University, 3-4-1 Okubo, Shinjuku, Tokyo 169-8555, Japan
`glueck@acm.org` and `kaz@futamura.info.waseda.ac.jp`

Abstract. We study the problem to transform functional programs, which intensively use append functions (like inefficient list reversal), into programs, which use accumulating parameters instead (like efficient list reversal). We give an (automatic) transformation algorithm for our problem and identify a class of functional programs, namely restricted 2-modular tree transducers, to which it can be applied. Moreover, since we get macro tree transducers as transformation result and since we also give the inverse transformation algorithm, we have a new characterization for the class of functions induced by macro tree transducers.

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. As running example we consider the following program p_{non} that contains functions *app* and *rev*, which append and reverse lists, respectively. For simplicity we only consider lists with elements A and B, where lists are represented by monadic trees. In particular, the empty list is represented by the symbol N.[1] The program p_{non} is a straightforward (but naive) solution for list reversal, since the definition of *rev* simply uses the function *app*.

$$
\begin{array}{llll}
\mathit{rev}\ (A\ x_1) = \mathit{app}\ (\mathit{rev}\ x_1)\ (A\ N) & \qquad & \mathit{app}\ (A\ x_1)\ y_1 = A\ (\mathit{app}\ x_1\ y_1) \\
\mathit{rev}\ (B\ x_1) = \mathit{app}\ (\mathit{rev}\ x_1)\ (B\ N) & & \mathit{app}\ (B\ x_1)\ y_1 = B\ (\mathit{app}\ x_1\ y_1) \\
\mathit{rev}\ N \qquad\ = N & & \mathit{app}\ N\ y_1 \qquad = y_1
\end{array}
$$

⋆ Research supported in part by the German Academic Exchange Service (DAAD).
⋆⋆ Research supported in part by the Japan Science and Technology Corporation (JST).
⋆⋆⋆ Research fellow of the Japan Society for the Promotion of Science (JSPS).

[1] Analogous functions for the usual list representation with (polymorphic) binary *Cons*-operator can be handled by our transformations just as well, but in order to describe such functions and transformations technically clean, an overhead to introduce types is necessary.

For list reversal, the program p_{non} has quadratic time-complexity in the length of an input list l, since it produces *intermediate results*, where the number and the maximum length of intermediate results depends on the length of l. Therefore we would prefer the following program p_{acc} with linear time-complexity in the length of an input list, which uses a binary auxiliary function rev'.

$$\begin{array}{ll} rev\ (A\ x_1) = rev'\ x_1\ (A\ N) & rev'\ (A\ x_1)\ y_1 = rev'\ x_1\ (A\ y_1) \\ rev\ (B\ x_1) = rev'\ x_1\ (B\ N) & rev'\ (B\ x_1)\ y_1 = rev'\ x_1\ (B\ y_1) \\ rev\ N \qquad = N & rev'\ N\ y_1 \qquad = y_1 \end{array}$$

Since p_{acc} reverses lists by accumulating their elements in the second argument of rev', we call p_{acc} an *accumulative* program, whereas we call p_{non} *non-accumulative*. Already in [1] it is shown in the context of transforming programs into iterative form, how non-accumulative programs can be transformed (non-automatically) into their more efficient accumulative versions. An algorithm which removes append functions in many cases, e.g. also in p_{non}, was presented in [23]. In comparison to [23], our transformation technique is more general in three aspects (though so far we only have artificial examples to demonstrate these generalizations): we consider (i) arbitrary tree structures (instead of lists), (ii) functions defined by simultaneous recursion, and (iii) substitution functions on trees (instead of append) which may replace different designated symbols by different trees. On the other hand, our technique is restricted to unary functions (apart from substitutions), though also in [23] the only example program involving a non-unary function could not be optimized. Moreover, since we formally describe the two different program paradigms and since we also present a transformation of accumulative into non-accumulative programs, we obtain the equality of the classes of functions computed by the two paradigms.

Well-known techniques for eliminating intermediate results cannot improve p_{non}: (i) *deforestation* [24] and *supercompilation* [22, 20] suffer from the phenomenon of the *obstructing function call* [2] and (ii) *shortcut deforestation* [14, 13] is hampered by the unknown number of intermediate results. In [3] an extension of shortcut deforestation was developed which is based on *type-inference* and splits function definitions into *workers* and *wrappers*. It successfully transforms p_{non} into p_{acc}, but is also less general in the above mentioned three aspects with respect to the transformation of non-accumulative into accumulative programs.

In [17, 18] it was demonstrated that sometimes *composition* and *decomposition techniques* [8, 11, 6, 12] for *attribute grammars* [16] and *tree transducers* [10] can help, when deforestation fails. For this purpose we have considered special functional programs as compositions of *macro tree transducers* (for short *mtts*) [5, 4, 6]. Every function f of an mtt is defined by a case analysis on the root symbol c of its first argument t. The right-hand side of the equation for f and c may only contain (*extended*) *primitive-recursive function calls*, i.e. the first argument of a function call has to be a variable that refers to a subtree of t. Under certain restrictions, compositions of mtts can be transformed into a single mtt. The function *app* in p_{non} is an mtt, whereas the function *rev* in p_{non} is not an mtt (since it calls *app* with a first argument that differs from x_1), such that these techniques cannot be applied directly.

In this paper we consider p_{non} as 2-*modular tree transducer* (for short 2-*modtt*) [7], where it is allowed that a function in module 1 (here *rev*) calls a function in module 2 (here *app*) non-primitive-recursively. Additionally, the two modules of p_{non} fulfill sufficient conditions (*rev* and *app* are so called top-down tree transducer- and substitution-modules, respectively), such that we can apply a decomposition step and two subsequent composition steps to transform p_{non} into the (more efficient) mtt p_{acc}. Since these constructions (called *accumulation*) transform every 2-modtt that fulfills our restrictions into an mtt, and since we also present inverse constructions (called *deaccumulation*) to transform mtts into the same class of restricted 2-modtts, we get a nice characterization of mtts in terms of restricted 2-modtts.

Besides this introduction, the paper contains five further sections. In Section 2 we fix elementary notions and notations. Section 3 introduces our functional language and tree transducers. Section 4 and Section 5 present accumulation and deaccumulation, respectively. Finally, Section 6 contains future research topics.

2 Preliminaries

We denote the set of natural numbers including 0 by $\mathbb{N}$ and the set $\mathbb{N} - \{0\}$ by $\mathbb{N}_+$. For every $m \in \mathbb{N}$, the set $\{1, \ldots, m\}$ is denoted by $[m]$. The cardinality of a set K is denoted by $card(K)$. We will use the sets $X = \{x_1, x_2, x_3, \ldots\}$, $Y = \{y_1, y_2, y_3, \ldots\}$, and $Z = \{z\}$ of *variables*. For every $n \in \mathbb{N}$, let $X_n = \{x_1, \ldots, x_n\}$ and $Y_n = \{y_1, \ldots, y_n\}$. In particular, $X_0 = Y_0 = \emptyset$.

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

A *ranked alphabet* is a pair $(S, rank)$ where S is a finite set and *rank* is a mapping which associates with every symbol $s \in S$ a natural number called the *rank* of s. We simply write S instead of $(S, rank)$ and assume *rank* as implicitly given. The set of elements of S with rank n is denoted by $S^{(n)}$. The set of *trees over S*, denoted by T_S, is the smallest subset $T \subseteq (S \cup \{(,)\})^*$ such that $S^{(0)} \subseteq T$ and for every $s \in S^{(n)}$ with $n \in \mathbb{N}_+$ and $t_1, \ldots, t_n \in T$: $(s\ t_1 \ldots t_n) \in T$.

3 Language

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 modules and every module consists of several function definitions. The functions of a module are defined by a complete 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 function definition there is a call of a function that is defined in the same module, then the first argument of this function call 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. In example programs and program transformations we relax the completeness of function definitions on T_{C_p} by leaving out those equations, which are not intended to be used in evaluations. Sometimes this leads to small technical difficulties, but avoids the overhead to introduce types.

Definition 1 Let C and F be ranked alphabets of *constructors* and *function symbols* (for short *functions*), respectively, such that $F^{(0)} = \emptyset$ and X, Y, C, and F are pairwise disjoint. We define the classes P, M, D, and R of *programs*, *modules*, *function definitions*, and *right-hand sides*, respectively, by the following grammar and the subsequent restrictions. We assume that p, m, d, r, c, and f (also equipped with indices) range over the sets P, M, D, R, C, and F, respectively.

$$
\begin{array}{lll}
p & ::= m_1 \ldots m_l & \text{(program)} \\
m & ::= d_1 \ldots d_h & \text{(module)} \\
d & ::= f\ (c_1\ x_1 \ldots x_{k_1})\ y_1 \ldots y_n = r_1 & \text{(function definition)} \\
 & \qquad \vdots & \\
 & \quad\ f\ (c_q\ x_1 \ldots x_{k_q})\ y_1 \ldots y_n = r_q & \\
r & ::= x_i \mid y_j \mid c\, r_1 \ldots r_k \mid f\ r_0\, r_1 \ldots r_n & \text{(right-hand side)}
\end{array}
$$

The sets of constructors, functions, and modules that occur in $p \in P$ are denoted by C_p, F_p, and M_p respectively. The set of functions that is defined in $m \in M_p$ is denoted by F_m. For every $i, j \in [l]$ with $i \neq j$: $F_{m_i} \cap F_{m_j} = \emptyset$. For every $i \in [l]$ and $f \in F_{m_i}$, the module m_i contains exactly one function definition for f. For every $i \in [l]$, $f \in F_{m_i}^{(n+1)}$, and $c \in C_p^{(k)}$ there is exactly one equation of the form

$$f\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = rhs(f, c)$$

with $rhs(f,c) \in RHS(F_{m_i}, C_p \cup (F_p - F_{m_i}), X_k, Y_n)$, where for every $F' \subseteq F$, $C' \subseteq C \cup F$, and $k, n \in I\!N$, $RHS(F', C', X_k, Y_n)$ is the smallest set RHS with:

- For every $f \in F'^{(a+1)}$, $i \in [k]$, and $r_1, \ldots, r_a \in RHS$: $(f\ x_i\ r_1 \ldots r_a) \in RHS$.
- For every $c \in C'^{(a)}$ and $r_1, \ldots, r_a \in RHS$: $(c\, r_1 \ldots r_a) \in RHS$.
- For every $j \in [n]$: $y_j \in RHS$. □

Example 2

- $p_{non} \in P$ where $M_{p_{non}}$ contains two modules $m_{non,rev}$ and $m_{non,app}$ containing the definitions of *rev* and *app*, respectively.
- $p_{acc} \in P$ where $M_{p_{acc}}$ contains one module $m_{acc,rev}$ containing the definitions of *rev* and *rev'*.
- Let p_{fre} be the program

$$
\begin{array}{llll}
rev\ (A\ x_1) & = APP\ (rev\ x_1)\ (A\ N) & \qquad app\ (A\ x_1)\ y_1 & = A\ (app\ x_1\ y_1) \\
rev\ (B\ x_1) & = APP\ (rev\ x_1)\ (B\ N) & \qquad app\ (B\ x_1)\ y_1 & = B\ (app\ x_1\ y_1) \\
rev\ N & = N & \qquad app\ N\ y_1 & = y_1
\end{array}
$$

$$
\begin{array}{ll}
int\ (APP\ x_1\ x_2) & = app\ (int\ x_1)\ (int\ x_2) \\
int\ (A\ x_1) & = A\ (int\ x_1) \\
int\ (B\ x_1) & = B\ (int\ x_1) \\
int\ N & = N
\end{array}
$$

Then, $p_{fre} \in P$ where $M_{p_{fre}}$ contains three modules $m_{fre,rev}$, $m_{fre,app}$, and $m_{fre,int}$ containing the definitions of rev, app, and int, respectively. $\square$

Now we will introduce a modular tree transducer as hierarchy $(m_1, \ldots, m_u)$ of modules, where the functions in a module m_j may only call functions defined in the modules $m_j, \ldots, m_u$. Moreover, we will define interpretation-modules which interpret designated constructors as function calls, and substitution-modules which substitute designated constructors by context arguments.

Definition 3 Let $p \in P$.

- A sequence $(m_1, \ldots, m_u)$ with $u \in \mathbb{N}_+$ and $m_1, \ldots, m_u \in M_p$ is called *u-modular tree transducer* (for short *u-modtt*), iff $F_{m_1}^{(1)} \neq \emptyset$ and for every $i \in [u]$, $f \in F_{m_i}^{(n+1)}$, and $c \in C_p^{(k)}$: $rhs(f, c) \in RHS(F_{m_i}, C_p \cup \bigcup_{i+1 \leq j \leq u} F_{m_j}, X_k, Y_n)$. We call F_{m_i} and $\bigcup_{i+1 \leq j \leq u} F_{m_j}$ the set of *internal* and *external functions* (cf. also [9]) *of* m_i, respectively.[2]
- A 1-modtt (m_1), abbreviated by m_1, is called *macro tree transducer* (for short *mtt*).
- An mtt m with $F_m = F_m^{(1)}$ is called *top-down tree transducer* (for short *tdtt*) [19, 21].
- A module $m \in M_p$ is also called *mtt-module*.[3]
- An mtt-module $m \in M_p$ with $F_m = F_m^{(1)}$ is called *tdtt-module*.
- A tdtt-module $m \in M_p$ is called *interpretation-module* (for short *int-module*), iff $card(F_m) = 1$, and there is $C_m \subseteq C_p$ such that m contains for $int \in F_m$ and for every $k \in \mathbb{N}$, $c \in C_m^{(k)}$ and for some $f_c \in (F_p - F_m)^{(k)}$ the equation
$$int\ (c\ x_1 \ldots x_k) = f_c\ (int\ x_1) \ldots (int\ x_k)$$
and for $int \in F_m$ and for every $k \in \mathbb{N}$ and $c \in (C_p - C_m)^{(k)}$ the equation
$$int\ (c\ x_1 \ldots x_k) = c\ (int\ x_1) \ldots (int\ x_k).$$
- An mtt-module $m \in M_p$ is called *substitution-module* (for short *sub-module*), iff there are $n \in \mathbb{N}$ and distinct $\pi_1, \ldots, \pi_n \in C_p^{(0)}$ such that $card(F_m^{(n+1)}) = 1$, $card(F_m^{(i+1)}) = 0$ for every $i > n$, $card(F_m^{(i+1)}) \leq 1$ for every $i < n$, and m contains for every $i \in \mathbb{N}$, $sub_i \in F_m^{(i+1)}$, and $j \in [i]$ the equation
$$sub_i\ \pi_j\ y_1 \ldots y_i = y_j$$
and for every $i \in \mathbb{N}$, $sub_i \in F_m^{(i+1)}$, $k \in \mathbb{N}$, and $c \in (C_p - \{\pi_1, \ldots, \pi_i\})^{(k)}$ the equation
$$sub_i\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_i = c\ (sub_i\ x_1\ y_1 \ldots y_i) \ldots (sub_i\ x_k\ y_1 \ldots y_i). \quad \square$$

[2] Our definition of modtts differs slightly from that in [7], since it allows a variable x_i in a right-hand side only as first argument of an internal function. Arbitrary occurrences of x_i can be achieved by applying an identity function to x_i, which is an additional internal function. Further note that the assumption $F_{m_1}^{(1)} \neq \emptyset$ only simplifies our presentation, but could be avoided.

[3] A module m is not necessarily an mtt, since it may call external functions.

Example 4

- $(m_{non,rev}, m_{non,app})$ is a 2-modtt, $m_{non,rev}$ is a tdtt-module, and $m_{non,app}$ is a sub-module (where $n = 1$, $\pi_1 = N$, $F^{(2)}_{m_{non,app}} = \{app\}$, and $F^{(1)}_{m_{non,app}} = \emptyset$).
- $m_{acc,rev}$ is an mtt-module and a 1-modtt, thus also an mtt.
- $m_{fre,rev}$ is a tdtt, $(m_{fre,int}, m_{fre,app})$ is a 2-modtt, $m_{fre,int}$ is an int-module (where $C_{m_{fre,int}} = \{APP\}$ and $f_{APP} = app$), and $m_{fre,app}$ is a sub-module. □

We fix call-by-name semantics, i.e. for every program $p \in P$ we use a call-by-name reduction relation $\Rightarrow_p$ on $T_{C_p \cup F_p}$. It can be proved in analogy to [7] that for every program $p \in P$, u-modtt $(m_1, \ldots, m_u)$ with $m_1, \ldots, m_u \in M_p$, $f \in F^{(1)}_{m_1}$, and $t \in T_{C_p}$ the normal form $nf(\Rightarrow_p, (f\ t))$ exists. The proof is based on the result, that for every modtt the corresponding (nondeterministic) reduction relation is terminating (and confluent). This result can also be extended to normal forms of expressions of the form $(f_n\ (f_{n-1} \ldots (f_2\ (f_1\ t)) \ldots))$, where every f_i is a unary function of the first module of a modtt in p.

In the framework of this paper we would like to optimize the evaluation of expressions of the form $(f\ t)$, where t is a tree over constructors. Since the particular constructor trees are not relevant for the transformations, we abstract them by a variable z, i.e. we handle expressions of the form $(f\ z)$. The transformations will also deliver expressions of the form $(f_2\ (f_1\ z))$. All these expressions are initial expressions for programs, which are defined as follows.

Definition 5 Let $p \in P$ and let f range over $\{f \mid$ there is a modtt $(m_1, \ldots, m_u)$ with $m_1, \ldots, m_u \in M_p$ such that $f \in F^{(1)}_{m_1}\}$. The set of *initial expressions for* p, denoted by E_p, is defined as follows, where e ranges over E_p:

$e ::= f\ e \mid z$ (initial expression for a program) □

Example 6

- z, $(rev\ z)$, and $(rev\ (rev\ z))$ are initial expressions for p_{non} and for p_{acc}.
- z, $(rev\ z)$, $(int\ z)$, and $(int\ (rev\ z))$ are initial expressions for p_{fre}. □

Definition 7 Let $p \in P$ and $e \in E_p$. The function $\tau_{p,e} : T_{C_p} \longrightarrow T_{C_p}$ defined by $\tau_{p,e}(t) = nf(\Rightarrow_p, e[z/t])$ for every $t \in T_{C_p}$, where $e[z/t]$ denotes the substitution of z in e by t, is called the *tree transformation induced by* p *and* e. The *class of tree transformations* induced by all programs $p \in P$ and initial expressions $(f_n\ (f_{n-1} \ldots (f_2\ (f_1\ z)) \ldots)) \in E_p$ with $n \geq 1$, such that $p = m_1 \ldots m_l$ with $m_1, \ldots, m_l \in M_p$, and for every $i \in [n]$ there is $j_i \in [l]$ with $f_i \in F_{m_{j_i}}$, is denoted by

$$M_{j_1}\, ;\, M_{j_2}\, ; \ldots ;\, M_{j_n}, \quad \text{where } M_{j_i} \text{ is}$$

- T, if m_{j_i} is a tdtt,
- MT, if m_{j_i} is an mtt, and
- u-$ModT(M'_1, \ldots, M'_u)$, if there is a u-modtt $(m'_1, \ldots, m'_u)$ with $m'_1, \ldots, m'_u \in M_p$ and $m_{j_i} = m'_1$, where M'_j is

- T, if m'_j is a tdtt-module,
- INT, if m'_j is an int-module, and
- SUB, if m'_j is a sub-module. □

Example 8

- $\tau_{p_{acc},(rev\ z)} \in MT$
- $\tau_{p_{acc},(rev\ (rev\ z))} \in MT\ ;\ MT$
- $\tau_{p_{non},(rev\ z)} \in 2\text{-}ModT(T, SUB)$
- $\tau_{p_{fre},(int\ (rev\ z))} \in T\ ;\ 2\text{-}ModT(INT, SUB)$ □

Our program transformations should preserve the semantics, i.e. should transform pairs in $\{(p, e) \mid p \in P, e \in E_p\}$ into equivalent pairs. Since some transformations will introduce new constructors, the following definition of equivalence can later be instantiated, such that it is restricted to input trees over "old" constructors.

Definition 9 For every $i \in [2]$ let $(p_i, e_i) \in \{(p, e) \mid p \in P, e \in E_p\}$. Let $C' \subseteq C_{p_1} \cap C_{p_2}$. The pairs (p_1, e_1) and (p_2, e_2) are called *equivalent with respect to* C', denoted by $(p_1, e_1) \equiv_{C'} (p_2, e_2)$, if for every $t \in T_{C'}$: $\tau_{p_1,e_1}(t) = \tau_{p_2,e_2}(t)$. □

4 Accumulation

We would like to give an algorithm based on results from the theory of tree transducers, which translates non-accumulative programs of the kind "inefficient reverse" into accumulative programs of the kind "efficient reverse". According to our definitions in the previous section this means to compose the two modules of a 2-modtt to an mtt. A result in [7] shows that in general this is not possible, since there are 2-modtts, such that their induced tree transformations can even not be realized by any finite composition of mtts. Fortunately, there are sufficient conditions for the two modules, under which a composition to an mtt is possible, namely, if they are a tdtt-module and a sub-module, respectively.

Theorem 10 $2\text{-}ModT(T, SUB) \subseteq MT$.
Proof. Follows from Lemma 11, Corollary 15, and Lemma 17. □

4.1 Freezing

Surprisingly, in the first step we will decompose modtts. We use a technique, which is based on Lemma 5.1 of [7]: The first module of a modtt can be separated from the rest of the modtt by freezing the occurrences of external functions in right-hand sides, i.e. substituting them by new constructors. These new constructors are later activated by an interpretation function, which interprets every new constructor c as call of that function f_c, which corresponds to c by freezing.

Lemma 11 $2\text{-}ModT(T, SUB) \subseteq T\ ;\ 2\text{-}ModT(INT, SUB)$.
Proof. Let $p \in P$ and $e \in E_p$, such that $m_1, m_2 \in M_p$, (m_1, m_2) is a 2-modtt, m_1 is a tdtt-module, m_2 is a sub-module, and $e = (f\ z)$ for some $f \in F_{m_1}$. We construct $p' \in P$ from p by changing m_1 and by adding the int-module m_3:

1. For every $g \in F_{m_1}$ and $c \in C_p^{(k)}$, the equation $g\ (c\ x_1 \ldots x_k) = rhs(g, c)$ of m_1 is replaced by $g\ (c\ x_1 \ldots x_k) = \underline{freeze}(rhs(g, c))$, where $\underline{freeze}(rhs(g, c))$ is constructed from $rhs(g, c)$ by replacing every occurrence of $sub_i \in F_{m_2}^{(i+1)}$ by a new constructor $SUB_i \in (C - C_p)^{(i+1)}$. Thus, m_1 becomes a tdtt.
2. m_3 contains for a new function $int \in (F - F_p)^{(1)}$ and for every $c \in C_p^{(k)}$ the equation

$$int\ (c\ x_1 \ldots x_k) = c\ (int\ x_1) \ldots (int\ x_k)$$

   and for every $sub_i \in F_{m_2}^{(i+1)}$ the equation

$$int\ (SUB_i\ x_1 \ldots x_{i+1}) = sub_i\ (int\ x_1) \ldots (int\ x_{i+1}).$$

   Thus, (m_3, m_2) is a 2-modtt.

Let $e' = (int\ (f\ z))$, i.e. $e' \in E_{p'}$. Then, $(p, e) \equiv_{C_p} (p', e')$ (cf. [7]). □

Example 12 Freezing translates p_{non} with initial expression $(rev\ z)$ into p_{fre} with initial expression $(int\ (rev\ z))$. □

4.2 Integration

Now, having 2-modtts with int- and sub-module, we can compose the two modules to an mtt. The main idea is to integrate the behaviour of the interpretation into the substitution. This is best explained by means of our example: The equation $int\ (APP\ x_1\ x_2) = app\ (int\ x_1)\ (int\ x_2)$ is replaced by the new equation $app\ (APP\ x_1\ x_2)\ y_1 = app\ x_1\ (app\ x_2\ y_1)$, which interprets APP as function app and sends app into the subtrees of APP. Note that the new equation represents the associativity of app, if we interpret APP also on the left-hand side as app. The associativity of app, which is the function of a special sub-module (cf. Example 4), plays already in [1, 23] an important role, and, for functions of general sub-modules, it will be proved as basis for our integration technique.

Lemma 13 2-$ModT(INT, SUB) \subseteq MT$.
Proof. Let $p \in P$ and $e \in E_p$, such that $m_1, m_2 \in M_p$, (m_1, m_2) is a 2-modtt, m_1 is an int-module, m_2 is a sub-module, and $e = (int\ z)$ with $F_{m_1} = \{int\}$. We construct $p' \in P$ by replacing m_1 and m_2 in p by the following mtt m:

1. Every equation $sub_i\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_i = rhs(sub_i, c)$ with $sub_i \in F_{m_2}^{(i+1)}$ and $c \in (C_p - C_{m_1})^{(k)}$ (cf. Def. 3 for C_{m_1}) of m_2 is taken over to m, where every occurrence of $sub_i \in F_{m_2}^{(i+1)}$ is changed into $sub'_i \in F_m^{(i+1)}$.
2. If $F_{m_2}^{(1)} = \emptyset$, then for a new function $sub'_0 \in (F - F_p)^{(1)}$ and for every $c \in (C_p - C_{m_1})^{(k)}$ we add $sub'_0\ (c\ x_1 \ldots x_k) = c\ (sub'_0\ x1) \ldots (sub'_0\ x_k)$ to m.
3. For every $sub'_i \in F_m^{(i+1)}$ and $SUB_j \in C_{m_1}^{(j+1)}$, the equation

$$\begin{aligned} &sub'_i\ (SUB_j\ x_1\ x_2 \ldots x_{j+1})\ y_1 \ldots y_i \\ =\ &sub'_j\ x_1\ (sub'_i\ x_2\ y_1 \ldots y_i) \ldots (sub'_i\ x_{j+1}\ y_1 \ldots y_i) \end{aligned}$$

   is added to m.

Let $e' = (sub'_0\ z)$, i.e. $e' \in E_{p'}$. Then $(p, e) \equiv_{C_p} (p', e')$, since for every $t \in T_{C_p}$ and $sub_i \in F_{m_2}^{(i+1)}$:

$$\begin{array}{ll} \quad nf(\Rightarrow_p, (int\ t)) & \\ = nf(\Rightarrow_p, (sub_i\ (int\ t)\ \pi_1 \dots \pi_i)) & (\text{“}\pi\text{s are substituted by }\pi\text{s” (Struct. Ind.)}) \\ = nf(\Rightarrow_{p'}, (sub'_i\ t\ \pi_1 \dots \pi_i)) & (*) \\ = nf(\Rightarrow_{p'}, (sub'_0\ t)) & (\text{“}\pi\text{s are substituted by }\pi\text{s” (Struct. Ind.)}) \end{array}$$

The statement $(*)$ For every $sub_i \in F_{m_2}^{(i+1)}$, and $t, t_1, \dots, t_i \in T_{C_p}$:
$nf(\Rightarrow_p, (sub_i\ (int\ t)\ t_1 \dots t_i)) = nf(\Rightarrow_{p'}, (sub'_i\ t\ t_1 \dots t_i))$.

is proved by structural induction on $t \in T_{C_p}$. We only show the interesting case $t = (SUB_j\ t'_0\ t'_1 \dots t'_j)$ with $SUB_j \in C_{m_1}^{(j+1)}$ and $t'_0, \dots, t'_j \in T_{C_p}$:

$$\begin{array}{lll} \quad nf(\Rightarrow_p, (sub_i\ (int\ (SUB_j\ t'_0\ t'_1 \dots t'_j))\ t_1 \dots t_i)) & & \\ = nf(\Rightarrow_p, (sub_i\ (sub_j\ (int\ t'_0)\ (int\ t'_1) \dots (int\ t'_j))\ t_1 \dots t_i)) & & (\text{Def. } int) \\ = nf(\Rightarrow_p, (sub_j\ (int\ t'_0) & (sub_i\ (int\ t'_1)\ t_1 \dots t_i) \dots & (**) \\ & (sub_i\ (int\ t'_j)\ t_1 \dots t_i))) & \\ = nf(\Rightarrow_p, (sub_j\ (int\ t'_0) & nf(\Rightarrow_p, (sub_i\ (int\ t'_1)\ t_1 \dots t_i)) \dots & (\text{“Split } nf\text{”}) \\ & nf(\Rightarrow_p, (sub_i\ (int\ t'_j)\ t_1 \dots t_i)))) & \\ = nf(\Rightarrow_{p'}, (sub'_j\ t'_0 & nf(\Rightarrow_{p'}, (sub'_i\ t'_1\ t_1 \dots t_i)) \dots & (\text{Ind. Hyp. } (*)) \\ & nf(\Rightarrow_{p'}, (sub'_i\ t'_j\ t_1 \dots t_i)))) & \\ = nf(\Rightarrow_{p'}, (sub'_j\ t'_0\ (sub'_i\ t'_1\ t_1 \dots t_i) \dots (sub'_i\ t'_j\ t_1 \dots t_i))) & & (\text{“Collect } nf\text{”}) \\ = nf(\Rightarrow_{p'}, (sub'_i\ (SUB_j\ t'_0\ t'_1 \dots t'_j)\ t_1 \dots t_i)) & & (\text{Def. } sub'_i) \end{array}$$

The associativity of substitutions

$(**)$ For every $sub_j \in F_{m_2}^{(j+1)}, sub_i \in F_{m_2}^{(i+1)}, s_0, s_1, \dots, s_j \in T_{C_p - C_{m_1}}$, and $t_1, \dots, t_i \in T_{C_p}$:

$$\begin{array}{l} \quad nf(\Rightarrow_p, (sub_i\ (sub_j\ s_0\ s_1 \dots s_j)\ t_1 \dots t_i)) \\ = nf(\Rightarrow_p, (sub_j\ s_0\ (sub_i\ s_1\ t_1 \dots t_i) \dots (sub_i\ s_j\ t_1 \dots t_i))). \end{array}$$

is also proved by structural induction on $s_0 \in T_{C_p - C_{m_1}}$. We only show the "central" case $s_0 = \pi_k$ with $k \in [j]$:

$$\begin{array}{l} \quad nf(\Rightarrow_p, (sub_i\ (sub_j\ \pi_k\ s_1 \dots s_j)\ t_1 \dots t_i)) \\ = nf(\Rightarrow_p, (sub_i\ s_k\ t_1 \dots t_i)) \\ = nf(\Rightarrow_p, (sub_j\ \pi_k\ (sub_i\ s_1\ t_1 \dots t_i) \dots (sub_i\ s_j\ t_1 \dots t_i))) \end{array}$$ □

Example 14 Integration translates p_{fre} with initial expression $(int\ z)$ into the following program p_{int} with initial expression $(app'_0\ z)$:

$$\begin{array}{lll} rev\ (A\ x_1) = APP\ (rev\ x_1)\ (A\ N) & app'_0\ (APP\ x_1\ x_2) & = app'\ x_1\ (app'_0\ x_2) \\ rev\ (B\ x_1) = APP\ (rev\ x_1)\ (B\ N) & app'_0\ (A\ x_1) & = A\ (app'_0\ x_1) \\ rev\ N \qquad = N & app'_0\ (B\ x_1) & = B\ (app'_0\ x_1) \\ & app'_0\ N & = N \\ & app'\ (APP\ x_1\ x_2)\ y_1 & = app'\ x_1\ (app'\ x_2\ y_1) \\ & app'\ (A\ x_1)\ y_1 & = A\ (app'\ x_1\ y_1) \\ & app'\ (B\ x_1)\ y_1 & = B\ (app'\ x_1\ y_1) \\ & app'\ N\ y_1 & = y_1 \end{array}$$ □

Let $p, p' \in P$, $(f_1\ z), (f_2\ z), (f_2\ (f_1\ z)) \in E_p$, $(f_1\ z), (f_2'\ z), (f_2'\ (f_1\ z)) \in E_{p'}$, and $C' = C_p \cap C_{p'}$. If $(p, (f_1\ z)) \equiv_{C'} (p', (f_1\ z))$ and $(p, (f_2\ z)) \equiv_{C'} (p', (f_2'\ z))$, then $(p, (f_2\ (f_1\ z))) \equiv_{C'} (p', (f_2'\ (f_1\ z)))$. Thus in particular we get from Lemma 13:

Corollary 15 $T\,;\, 2\text{-}ModT(INT, SUB) \subseteq T\,;\, MT$. □

Example 16 Integration translates p_{fre} with initial expression $(int\ (rev\ z))$ into p_{int} with initial expression $(app_0'\ (rev\ z))$. □

4.3 Composition

In [19] it was shown that the composition of two tdtts can be simulated by only one tdtt. This result was generalized in [6], where in particular a composition technique is presented which constructs an mtt m for the composition of a tdtt m_1 with an mtt m_2. The central idea 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 m. For this purpose, m uses $F_{m_1} \times F_{m_2}$ as function set. In the following, we abbreviate every pair $(f, g) \in F_{m_1} \times F_{m_2}$ by $\overline{fg}$.

In our construction it will be necessary to extend the call-by-name reduction relation to expressions containing variables (they are handled like 0-ary constructors) and to restrict the call-by-name reduction relation to use only equations of a certain mtt m, which will be denoted by $\Rightarrow_m$.

Lemma 17 $T\,;\, MT \subseteq MT$.
Proof. Let $p \in P$ and $e \in E_p$, such that $m_1, m_2 \in M_p$, m_1 is a tdtt, m_2 is an mtt, and $e = (f_2\ (f_1\ z))$ for some $f_1 \in F_{m_1}$ and $f_2 \in F_{m_2}^{(1)}$. We construct $p' \in P$ by replacing m_1 and m_2 in p by the following mtt m:

1. From m_2 we construct an mtt $\bar{m}_2$ which is able to translate right-hand sides of equations of m_1. Note that $\bar{m}_2$ is not part of p'.
   - $\bar{m}_2$ contains the equations of m_2.
   - For every $g \in F_{m_2}^{(n+1)}$ and $f \in F_{m_1}$ we add the following equation to $\bar{m}_2$:

     $$g\ (f\ x_1)\ y_1 \ldots y_n = \overline{fg}\ x_1\ y_1 \ldots y_n$$

     where every $f \in F_{m_1}$ and $\overline{fg}$ with $f \in F_{m_1}$ and $g \in F_{m_2}^{(n+1)}$ is viewed as additional unary and $(n+1)$-ary constructor, respectively.[4]
2. Let $F_m^{(n+1)} = \{\overline{fg} \mid f \in F_{m_1}, g \in F_{m_2}^{(n+1)}\}$. For every $g \in F_{m_2}^{(n+1)}$, $f \in F_{m_1}$, and $c \in C_p^{(k)}$, such that $f\ (c\ x_1 \ldots x_k) = rhs(f, c)$ is an equation in m_1, m contains the equation

   $$\overline{fg}\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = nf(\Rightarrow_{\bar{m}_2}, g\ (rhs(f, c))\ y_1 \ldots y_n).$$

[4] In a strong sense, $\overline{fg}\ x_1\ y_1 \ldots y_n$ is not a legal right-hand side, since x_1 does not occur as first argument of a function symbol, but of a constructor. Again, an additional identity function would solve the problem formally.

Let $e' = (\overline{f_1 f_2}\ z)$, i.e. $e' \in E_{p'}$. Then, $(p, e) \equiv_{C_p} (p', e')$. We omit the proof and only mention that in [6] another construction was used, which first splits the mtt into a tdtt and a so called *yield function*, which handles parameter substitutions (cf. also Subsection 5.1), then composes the two tdtts, 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 [18]). □

Example 18 Composition translates p_{int} with initial expression $(app'_0\ (rev\ z))$ into a program and an initial expression, which can be obtained from p_{acc} and $(rev\ z)$, respectively, by a renaming of functions:

Let $M_{p_{int}}$ contain the tdtt $m_{int,rev}$ and the mtt $m_{int,app}$ containing the definitions of rev and of app'_0 and app', respectively. The mtt $\bar{m}_{int,app}$ is given by:

$$
\begin{array}{llll}
app'_0\,(APP\ x_1\ x_2) & = app'\ x_1\ (app'_0\ x_2) & app'\ (APP\ x_1\ x_2)\ y_1 & = app'\ x_1\ (app'\ x_2\ y_1) \\
app'_0\,(A\ x_1) & = A\ (app'_0\ x_1) & app'\ (A\ x_1)\ y_1 & = A\ (app'\ x_1\ y_1) \\
app'_0\,(B\ x_1) & = B\ (app'_0\ x_1) & app'\ (B\ x_1)\ y_1 & = B\ (app'\ x_1\ y_1) \\
app'_0\ N & = N & app'\ N\ y_1 & = y_1 \\
app'_0\,(rev\ x_1) & = \overline{revapp'_0}\ x_1 & app'\ (rev\ x_1)\ y_1 & = \overline{revapp'}\ x_1\ y_1
\end{array}
$$

The new program contains the following equations with underlined left- and right-hand sides:

$$
\begin{array}{ll}
\quad \underline{\overline{revapp'}\ (A\ x_1)\ y_1} & \quad \underline{\overline{revapp'}\ (B\ x_1)\ y_1} \\
= nf(\Rightarrow_{\bar{m}_{int,app}}, app'\ (rhs(rev, A))\ y_1) & = \ldots = \underline{\overline{revapp'}\ x_1\ (B\ y_1)} \\
= nf(\Rightarrow_{\bar{m}_{int,app}}, app'\ (APP\ (rev\ x_1)\ (A\ N))\ y_1) & \\
= nf(\Rightarrow_{\bar{m}_{int,app}}, app'\ (rev\ x_1)\ (app'\ (A\ N)\ y_1)) & \quad \underline{\overline{revapp'}\ N\ y_1} \\
= nf(\Rightarrow_{\bar{m}_{int,app}}, \overline{revapp'}\ x_1\ (app'\ (A\ N)\ y_1)) & = \ldots = \underline{y_1} \\
= \underline{\overline{revapp'}\ x_1\ (A\ y_1)} &
\end{array}
$$

$$
\begin{array}{ll}
\quad \underline{\overline{revapp'_0}\ (A\ x_1)} & \quad \underline{\overline{revapp'_0}\ (B\ x_1)} \\
= nf(\Rightarrow_{\bar{m}_{int,app}}, app'_0\ (rhs(rev, A))) & = \ldots = \underline{\overline{revapp'}\ x_1\ (B\ N)} \\
= nf(\Rightarrow_{\bar{m}_{int,app}}, app'_0\ (APP\ (rev\ x_1)\ (A\ N))) & \\
= nf(\Rightarrow_{\bar{m}_{int,app}}, app'\ (rev\ x_1)\ (app'_0\ (A\ N))) & \quad \underline{\overline{revapp'_0}\ N} \\
= nf(\Rightarrow_{\bar{m}_{int,app}}, \overline{revapp'}\ x_1\ (app'_0\ (A\ N))) & = \ldots = \underline{N} \\
= \underline{\overline{revapp'}\ x_1\ (A\ N)} &
\end{array}
$$

The new initial expression is $(\overline{revapp'_0}\ z)$. Note that by a result in [18] we could have used also deforestation to get the same transformation result. □

5 Deaccumulation

This section shows that results from the theory of tree transducers can also be applied to translate accumulative programs of the kind "efficient reverse" into non-accumulative programs of the kind "inefficient reverse", thereby proving that the classes 2-$ModT(T, SUB)$ and MT are equal. So far we consider this transformation direction as purely theoretical, but there may be cases, in which the non-accumulative programs are more efficient than their related accumulative versions (see Section 6).

Theorem 19 $MT \subseteq 2\text{-}ModT(T, SUB)$.
Proof. Follows from Lemma 21 and Lemma 23. □

From Theorems 10 and 19 we get a new characterization of MT:

Corollary 20 $MT = 2\text{-}ModT(T, SUB)$. □

5.1 Decomposition

We have already mentioned that an mtt can be simulated by the composition of a tdtt with a yield function [5,6]. In this paper we will use a construction for this result, which is based on the proof of Lemma 5.5 in [7]. The key idea is to simulate the task of an $(n+1)$-ary function g by a unary function g'.[5] Since g' does not know the current values of its context arguments, it uses a new constructor π_j, wherever g uses its j-th context argument. For this purpose, every variable y_j in the right-hand sides of equations for g is replaced by π_j. The current context arguments themselves are integrated into the calculation by replacing every occurrence of the form $(g\ x_i \ldots)$ in a right-hand side by $(SUB_n\ (g'\ x_i) \ldots)$, where SUB_n is a new constructor. Roughly speaking, an int-module interprets every occurrence of SUB_n as substitution, which replaces every occurrence of π_j in the first subtree of SUB_n by the j-th context argument.

Lemma 21 $MT \subseteq T\,;\,2\text{-}ModT(INT, SUB)$.
Proof. Let $p \in P$ and $e \in E_p$, such that $m \in M_p$ is an mtt and $e = (f\ z)$ for some $f \in F_m^{(1)}$. We construct $p' \in P$ by replacing m in p by the following tdtt m_1, int-module m_2, and sub-module m_3, where (m_2, m_3) is a 2-modtt. Let $A = \{n \in I\!N \mid F_m^{(n+1)} \neq \emptyset\}$ and mx be the maximum of A.

1. For every $a \in A - \{0\}$ let $SUB_a \in (C - C_p)^{(a+1)}$ and for every $j \in [mx]$ let $\pi_j \in (C - C_p)^{(0)}$ be distinct new constructors.
   For every $g \in F_m^{(n+1)}$, $c \in C_p^{(k)}$, and equation $g\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = rhs(g, c)$ in m, an equation $g\ (c\ x_1 \ldots x_k) = \underline{tr}(rhs(g,c))$ with $g \in F_{m_1}^{(1)}$ is contained in m_1, where $\underline{tr} : RHS(F_m, C_p, X_k, Y_n) \longrightarrow RHS(F_{m_1}, C_p \cup \{SUB_a \mid a \in A - \{0\}\} \cup \{\pi_1, \ldots, \pi_n\}, X_k, Y_0)$ is defined by:

   For every $a \in I\!N_+, h \in F_m^{(a+1)}, i \in [k]$,
   and $r_1, \ldots, r_a \in RHS(F_m, C_p, X_k, Y_n)$:
   $\quad \underline{tr}(h\ x_i\ r_1 \ldots r_a) = SUB_a\ (h\ x_i)\ \underline{tr}(r_1) \ldots \underline{tr}(r_a)$.
   For every $h \in F_m^{(1)}$ and $i \in [k]$:
   $\quad \underline{tr}(h\ x_i) \qquad = (h\ x_i)$.
   For every $a \in I\!N, c' \in C_p^{(a)}$, and $r_1, \ldots, r_a \in RHS(F_m, C_p, X_k, Y_n)$:
   $\quad \underline{tr}(c'\ r_1 \ldots r_a) \quad = c'\ \underline{tr}(r_1) \ldots \underline{tr}(r_a)$.
   For every $j \in [n]$:
   $\quad \underline{tr}(y_j) \qquad = \pi_j$.

[5] In the formal construction g is not renamed.

2. m_2 contains for a new function $int \in (F - F_p)^{(1)}$ and for every $c \in (C_p \cup \{\pi_1, \ldots, \pi_{mx}\})^{(k)}$ the equation

$$int\ (c\ x_1 \ldots x_k) = c\ (int\ x_1) \ldots (int\ x_k)$$

and for every $a \in A - \{0\}$ the equation

$$int\ (SUB_a\ x_1 \ldots x_{a+1}) = sub_a\ (int\ x_1) \ldots (int\ x_{a+1}).$$

3. m_3 contains for every $a \in A - \{0\}$[6], for every new function $sub_a \in (F - F_p)^{(a+1)}$, and for every $c \in C_p^{(k)}$ the equation

$$sub_a\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_a = c\ (sub_a\ x_1\ y_1 \ldots y_a) \ldots (sub_a\ x_k\ y_1 \ldots y_a)$$

and for every $j \in [a]$ the equation

$$sub_a\ \pi_j\ y_1 \ldots y_a = y_j.$$

Let $e' = (int\ (f\ z))$, i.e. $e' \in E_{p'}$. Then, $(p, e) \equiv_{C_p} (p', e')$ (cf. [7]). $\square$

Example 22 Decomposition translates p_{acc} with initial expression $(rev\ z)$ into the following program p_{dec} with initial expression $(int\ (rev\ z))$:

$$\begin{array}{ll}
rev\ (A\ x_1) = SUB_1\ (rev'\ x_1)\ (A\ N) & rev'\ (A\ x_1) = SUB_1\ (rev'\ x_1)\ (A\ \pi_1) \\
rev\ (B\ x_1) = SUB_1\ (rev'\ x_1)\ (B\ N) & rev'\ (B\ x_1) = SUB_1\ (rev'\ x_1)\ (B\ \pi_1) \\
rev\ N \quad\quad = N & rev'\ N \quad\quad = \pi_1
\end{array}$$

$$\begin{array}{ll}
int\ (SUB_1\ x_1\ x_2) = sub_1\ (int\ x_1)\ (int\ x_2) & sub_1\ (A\ x_1)\ y_1 = A\ (sub_1\ x_1\ y_1) \\
int\ (A\ x_1) \quad\quad\quad = A\ (int\ x_1) & sub_1\ (B\ x_1)\ y_1 = B\ (sub_1\ x_1\ y_1) \\
int\ (B\ x_1) \quad\quad\quad = B\ (int\ x_1) & sub_1\ N\ y_1 \quad\quad = N \\
int\ N \quad\quad\quad\quad\quad = N & sub_1\ \pi_1\ y_1 \quad\quad = y_1 \\
int\ \pi_1 \quad\quad\quad\quad\quad = \pi_1 &
\end{array}$$

$\square$

5.2 Thawing

We use an inverse construction as in the proof of Lemma 11 to thaw "frozen functions" of a tdtt, i.e. we substitute occurrences of those constructors SUB_i, which the interpretation replaces by functions sub_i, by occurrences of sub_i.

Lemma 23 $T\,;\, 2\text{-}ModT(INT, SUB) \subseteq 2\text{-}ModT(T, SUB)$.

Proof. Let $p \in P$ and $e \in E_p$, such that $m_1, m_2, m_3 \in M_p$, m_1 is a tdtt, (m_2, m_3) is a 2-modtt, m_2 is an int-module, m_3 is a sub-module, and $e = (int\ (f\ z))$ with $F_{m_2} = \{int\}$ and for some $f \in F_{m_1}$. We construct $p' \in P$ from p by dropping m_2 and by changing m_1:

For every $g \in F_{m_1}$ and $c \in C_p^{(k)}$, the equation $g\ (c\ x_1 \ldots x_k) = rhs(g, c)$ of m_1 is replaced by $g\ (c\ x_1 \ldots x_k) = \underline{thaw}(rhs(g, c))$, where $\underline{thaw}(rhs(g, c))$

[6] If $A = \{0\}$, i.e. m was already a tdtt, then we construct a "dummy function" sub_0.

is constructed from $rhs(g,c)$ by replacing every occurrence of $SUB_i \in C_{m_2}^{(i+1)}$ (cf. Def. 3 for C_{m_2}) by $sub_i \in F_{m_3}^{(i+1)}$, iff the equation $int\ (SUB_i\ x_1 \ldots x_{i+1}) = sub_i\ (int\ x_1) \ldots (int\ x_{i+1})$ is in m_2. Thus, m_1 becomes a tdtt-module and (m_1, m_3) a 2-modtt.

Let $e' = (f\ z)$, i.e. $e' \in E_{p'}$. Then, $(p,e) \equiv_{C_p - C_{m_2}} (p', e')$, since

(∗) For every $g \in F_{m_1}$ and $t \in T_{C_p - C_{m_2}}$: $nf(\Rightarrow_p, (int\ (g\ t))) = nf(\Rightarrow_{p'}, (g\ t))$.

is proved by structural induction on $t \in T_{C_p - C_{m_2}}$. This proof requires another structural induction on $r \in RHS(F_{m_1}, C_p, X_k, Y_0)$ to prove

(∗∗) For every $k \in I\!N, r \in RHS(F_{m_1}, C_p, X_k, Y_0)$, and $t_1, \ldots, t_k \in T_{C_p - C_{m_2}}$:
$nf(\Rightarrow_p, (int\ r[x_1/t_1, \ldots, x_k/t_k])) = nf(\Rightarrow_{p'}, \underline{thaw}(r)[x_1/t_1, \ldots, x_k/t_k])$,

where $[x_1/t_1, \ldots, x_k/t_k]$ denotes the substitution of every occurrence of x_i in r and $\underline{thaw}(r)$, respectively, by t_i. □

Example 24 Thawing translates p_{dec} with initial expression $(int\ (rev\ z))$ into the following program p'_{non} with initial expression $(rev\ z)$:

$$
\begin{array}{lll}
rev\ (A\ x_1) = sub_1\ (rev'\ x_1)\ (A\ N) & \quad & rev'\ (A\ x_1) = sub_1\ (rev'\ x_1)\ (A\ \pi_1) \\
rev\ (B\ x_1) = sub_1\ (rev'\ x_1)\ (B\ N) & & rev'\ (B\ x_1) = sub_1\ (rev'\ x_1)\ (B\ \pi_1) \\
rev\ N \qquad = N & & rev'\ N \qquad = \pi_1
\end{array}
$$

$$
\begin{array}{ll}
sub_1\ (A\ x_1)\ y_1 & = A\ (sub_1\ x_1\ y_1) \\
sub_1\ (B\ x_1)\ y_1 & = B\ (sub_1\ x_1\ y_1) \\
sub_1\ N\ y_1 & = N \\
sub_1\ \pi_1\ y_1 & = y_1
\end{array}
$$

It is surprising that p'_{non} is very similar to p_{non}: sub_1 in p'_{non} corresponds to app in p_{non}, but substitutes the symbol π_1 instead of N. rev' in p'_{non} corresponds to rev in p_{non}, but uses sub_1 and π_1 instead of app and N. The additional rev in p'_{non} achieves that π_1 does not occur in the output. □

6 Future Work

In this paper we have always considered non-accumulative functional programs as less efficient than their related accumulative versions, like we have considered in [17] the pure elimination of intermediate results as success. This assumption is true for so far studied example programs, but may be wrong in general. It is necessary to find out sufficient conditions for our source programs, under which we can guarantee that the accumulation technique (and maybe also the deaccumulation technique, respectively) does not deteriorate the efficiency. In particular, is the linearity of programs a sufficient condition for accumulation (like for deforestation [24] and tree transducer composition [18, 15])?

We have presented sufficient conditions, such that we can compose the modules of a 2-modtt. Is it possible to extend the applicability of the accumulation technique by relaxing these conditions or by using other conditions? Additionally, it would be interesting to analyze u-modtts with $u > 2$ in this context.

References

1. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. Assoc. Comput. Mach.*, 24:44–67, 1977.
2. W.-N. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4:515–555, 1994.
3. O. Chitil. Type-inference based short cut deforestation (nearly) without inlining. In *IFL'99, Lochem, The Netherlands, Proceedings, September 1999*, volume 1868 of *LNCS*, pages 19–36. Springer-Verlag, April 2000.
4. B. Courcelle and P. Franchi–Zannettacci. Attribute grammars and recursive program schemes. *Theor. Comp. Sci.*, 17:163–191, 235–257, 1982.
5. 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.
6. J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comp. Syst. Sci.*, 31:71–145, 1985.
7. J. Engelfriet and H. Vogler. Modular tree transducers. *Theor. Comp. Sci.*, 78:267–304, 1991.
8. Z. Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981.
9. Z. Fülöp, F. Herrmann, S. Vágvölgyi, and H. Vogler. Tree transducers with external functions. *Theor. Comp. Sci.*, 108:185–236, 1993.
10. Z. Fülöp and H. Vogler. *Syntax-directed semantics — Formal models based on tree transducers*. Monographs in Theoretical Computer Science. Springer-Verlag, 1998.
11. H. Ganzinger. Increasing modularity and language–independency in automatically generated compilers. *Science of Computer Programming*, 3:223–278, 1983.
12. R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.
13. A. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.
14. A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *FPCA'93, Copenhagen, Denmark, Proceedings*, pages 223–231. ACM Press, 1993.
15. M. Höff. Vergleich von Verfahren zur Elimination von Zwischenergebnissen bei funktionalen Programmen. Master's thesis, Dresden University of Technology,1999.
16. D.E. Knuth. Semantics of context–free languages. *Math. Syst. Th.*, 2:127–145, 1968. Corrections in *Math. Syst. Th.*, 5:95-96, 1971.
17. A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In *FST & TCS'98, Chennai, India, Proceedings*, volume 1530 of *LNCS*, pages 146–157. Springer-Verlag, December 1998.
18. A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In *FLOPS'99, Tsukuba, Japan, Proceedings*, volume 1722 of *LNCS*, pages 114–130. Springer-Verlag, November 1999.
19. W.C. Rounds. Mappings and grammars on trees. *Math. Syst.Th.*, 4:257–287, 1970.
20. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6:811–838, 1996.
21. J.W. Thatcher. Generalized^2 sequential machine maps. *J. Comp. Syst. Sci.*, 4:339–367, 1970.
22. V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8:292–325, 1986.
23. P. Wadler. The concatenate vanishes. Note, University of Glasgow, December 1987 (Revised, November 1989).
24. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comp. Sci.*, 73:231–248, 1990.