Deaccumulation — Improving Provability*

Jürgen Giesl[1], Armin Kühnemann[2], and Janis Voigtländer[2]**

[1]LuFG Informatik II, RWTH Aachen, Ahornstr. 55, D–52074 Aachen, Germany
giesl@informatik.rwth-aachen.de

[2]Institute for Theoretical Computer Science, Department of Computer Science, Dresden University of Technology, D–01062 Dresden, Germany
kuehne@tcs.inf.tu-dresden.de and voigt@tcs.inf.tu-dresden.de

Abstract. Several induction theorem provers were developed to verify functional programs mechanically. Unfortunately, automated verification usually fails for functions with accumulating arguments. In particular, this holds for tail-recursive functions that correspond to imperative programs, but also for programs with nested recursion.
Based on results from the theory of tree transducers, we develop an automatic transformation technique. It transforms accumulative functional programs into non-accumulative ones, which are much better suited for automated verification by induction theorem provers. Hence, in contrast to classical program transformations aiming at improving the efficiency, the goal of our deaccumulation technique is to improve the provability.

1 Introduction

In safety-critical applications, a formal verification of programs is required. However, since mathematical correctness proofs are very expensive and time-consuming, one tries to automate this task as much as possible. Since *induction* is an important proof technique required for program verification, several *induction theorem provers* have been developed, which can be used for mechanized reasoning about program properties (e.g., *NQTHM* [4], *ACL-2* [17], *RRL* [16], *CLAM* [5], *INKA* [1, 26], and *SPIKE* [3]). However, while such provers are successfully applied for *functional* programs, they often have severe problems in dealing with *imperative* programs.

As running example, we consider the calculation of a decreasing list containing the first x_1 even numbers (i.e., $[2x_1 - 2, \ldots, 4, 2, 0]$). This problem can be solved by the following part p_{even} of an imperative program (in C-like syntax):

```
[int] even (int x1)
{ int y1 = 0;  [int] y2 = [];
  while (x1!=0)  { y2 = y1:y2;  y1 = y1+2;  x1--; }
  return y2;                                        }
```

Here, `[int]` denotes the type of integer lists, `[]` denotes the empty list, and `:` denotes list insertion, i.e., $y_1 : y_2$ inserts the element y_1 in front of the list y_2.

* *Proceedings of the 8th Asian Computing Science Conference (ASIAN '03)*, Mumbai, India, LNCS, Springer-Verlag, 2003.

** Research of this author supported by the DFG under grants KU 1290/2-1 and 2-4.

Classical techniques for verifying imperative programs are based on inventing suitable *loop invariants* [13]. However, while there are heuristics for finding loop invariants [15, 23], in general this task is hard to mechanize [7].

Instead, our aim is to use the existing powerful induction theorem provers also for the verification of imperative programs. To this end, imperative programs are translated into the functional input language of induction provers. In the absence of pointers, such an automatic translation is easily possible [20] by transforming every while-loop into a separate function whose parameters record the changes during a run through the while-loop. For our program p_{even} we obtain the following tail-recursive program p_{acc} (in Haskell-like syntax) together with an initial call $r_{acc} = (f\ x_1\ 0\ [\,])$. It uses pattern matching on x_1 (called *recursion argument*) and represents natural numbers with the constructors 0 and S for the successor function:

$$\begin{array}{lll} p_{acc}: & f\ (S\ x_1)\ y_1\ y_2 & = f\ x_1\ (S\ (S\ y_1))\ (y_1 : y_2) \\ & f\ 0 \qquad\ y_1\ y_2 & = y_2 \end{array}$$

The above translation of imperative into functional programs always yields tail-recursive functions that compute their result using accumulators. Indeed, f accumulates values in its *context arguments* (arguments different from the recursion argument, i.e., f's second and third argument). A function is called *accumulative* if its context arguments are modified in its recursive calls. For instance, f is accumulative, because both the second and the third argument do not remain unchanged in the recursive call. A program like p_{acc} is called *accumulative* if it contains an accumulative function.

Assume that our aim is to verify the equivalence of r_{acc} and $r_q = (q\ x_1)$ for all natural numbers x_1, where p_q is the following functional specification of our problem. Here, $(q\ x_1)$ calculates the desired list and $(q'\ x_1)$ computes $2 \cdot x_1$:

$$\begin{array}{lll} q\ (S\ x_1) & = (q'\ x_1) : (q\ x_1) & \qquad q'\ (S\ x_1) = S\ (S\ (q'\ x_1)) \\ q\ 0 & = [\,] & \qquad q'\ 0 \qquad\ = 0 \end{array}$$

Note that even if there exists a "natural" non-accumulative recursive specification of a problem, imperative programs are typically written using loops, which translate into accumulative programs. The accumulative version may also be more efficient than a non-accumulative implementation (see e.g., App. B).

But unfortunately, accumulative programs are not suitable for mechanized verification. For example, an automatic proof of

$$(f\ x_1\ 0\ [\,]) = (q\ x_1)$$

by induction (using this equation for fixed x_1 as induction hypothesis) fails, because in the induction step $(x_1 \mapsto (S\ x_1))$ the induction hypothesis cannot be successfully applied to prove $(f\ (S\ x_1)\ 0\ [\,]) = (q\ (S\ x_1))$. For instance, for this conjecture the *ACL-2* prover performs a series of generalizations that do not increase verifiability, and it ends up with consuming all memory available. The reason for the verification problems is that f uses accumulators: the

context arguments of the term $(f\ x_1\ \underline{(S\ (S\ 0))}\ \underline{(0 : [\,])})$, which originates from rule application to $(f\ (S\ x_1)\ 0\ [\,])$, do not fit to the context arguments of the term $(f\ x_1\ \underline{0}\ \underline{[\,]})$ in the induction hypothesis! So the problem is that accumulating parameters are typically initialized with some fixed values (like 0 and $[\,]$), which then appear also in the conjecture to be proved and hence in the induction hypothesis. But since accumulators are changed in recursive calls, after rule application we have different values like $(S\ (S\ 0))$ and $(0 : [\,])$ in the induction conclusion of the step case.

In induction theorem proving, this problem is usually solved by transforming the conjecture to be proved. In other words, the aim is to invent a suitable *generalization* (see, e.g., [4, 14, 15, 26]). So, instead of the original conjecture $(f\ x_1\ 0\ [\,]) = (q\ x_1)$, one tries to find a *stronger* conjecture that however is *easier* to prove. In our example, the original conjecture may be generalized to

$$(f\ x_1\ y_1\ y_2) = (\bar{q}\ x_1\ y_1) +\!\!+\ y_2\,,$$

where $+\!\!+$ denotes list concatenation and where $\bar{q}$ and $\bar{q}'$ are defined as follows:

$$\begin{array}{llll} \bar{q}\ (S\ x_1)\ y_1 = (\bar{q}'\ x_1\ y_1) : (\bar{q}\ x_1\ y_1) & & \bar{q}'\ (S\ x_1)\ y_1 = S\ (S\ (\bar{q}'\ x_1\ y_1)) \\ \bar{q}\ 0 \qquad y_1 = [\,] & & \bar{q}'\ 0 \qquad y_1 = y_1 \end{array}$$

However, finding such generalizations automatically is again very hard. In fact, it is as difficult as discovering loop invariants for the original imperative program. Therefore, developing techniques to verify accumulative functions is one of the most important research topics in the area of inductive theorem proving [14].

In contrast to the classical approach of generalizing conjectures, we suggest an automated program transformation, which transforms functions that are hard to verify into functions that are much more suitable for mechanized verification. The advantage of this approach is that it works fully automatically and that by transforming a function definition, the verification problems with this function are solved once and for all (i.e., for all conjectures one would like to prove about this function). In contrast, when using the generalization approach, one has to find a new generalization for every new conjecture to be proved. In particular, finding generalizations automatically is difficult for conjectures with *several* occurrences of an accumulative function (see e.g., [12] and App. A and B).

The semantics-preserving transformation to be presented in this paper transforms the original program p_{acc} into the following program p_{non}:

$$\begin{array}{lll} p_{non}: & f'\ (S\ x_1) & = sub\ (f'\ x_1)\ (S\ (S\ 0))\ (0 : [\,]) \\ & f'\ 0 & = [\,] \end{array}$$

$$\begin{array}{llll} sub\ (x_1 : x_2)\ y_1\ y_2 = (sub\ x_1\ y_1\ y_2) : (sub\ x_2\ y_1\ y_2) & & sub\ 0\ y_1\ y_2 = y_1 \\ sub\ (S\ x_1) \quad y_1\ y_2 = S\ (sub\ x_1\ y_1\ y_2) & & sub\ [\,]\ y_1\ y_2 = y_2 \end{array}$$

together with an initial call $r_{non} = (f'\ x_1)$. Since p_{non} contains a function f' without context arguments, and a function *sub* with unchanged context arguments in recursive calls, p_{non} is a *non-accumulative* program and our transformation technique is called *deaccumulation*. An application of the *substitution*

function[1] *sub* of the form $(sub\ t\ s_1\ s_2)$ replaces all occurrences of 0 in the term t by the term s_1 and all occurrences of $[\,]$ by s_2. For instance, the decreasing list of the first three even numbers is computed by p_{non} as follows:

$$\begin{aligned} f'\ (S^3\ 0) &\Rightarrow^4_{p_{non}} sub\ (sub\ (sub\ [\,]\ (S^2\ 0)\ (0:[\,]))\ (S^2\ 0)\ (0:[\,]))\ (S^2\ 0)\ (0:[\,]) \\ &\Rightarrow_{p_{non}} sub\ (sub\ (0:[\,])\ (S^2\ 0)\ (0:[\,]))\ (S^2\ 0)\ (0:[\,]) \\ &\Rightarrow^3_{p_{non}} sub\ ((S^2\ 0):(0:[\,]))\ (S^2\ 0)\ (0:[\,]) \\ &\Rightarrow^7_{p_{non}} (S^4\ 0):((S^2\ 0):(0:[\,])) \end{aligned}$$

This computation shows that the constructors 0 and $[\,]$ in p_{non} are used as "placeholders", which are repeatedly substituted by $(S^2\ 0)$ and $(0:[\,])$, respectively.

Now, the statement $(f'\ x_1) = (q\ x_1)$ (taken as induction hypothesis $IH1$) can be proved automatically by three nested inductions as follows. During the proof, the new subgoals

$$\begin{aligned} IH2: &\quad (sub\ (q\ x_1)\ (S^2\ 0)\ (0:[\,])) = ((q'\ x_1):(q\ x_1)) \qquad \text{and} \\ IH3: &\quad (sub\ (q'\ x_1)\ (S^2\ 0)\ (0:[\,])) = (S^2\ (q'\ x_1)) \end{aligned}$$

are generated. Note that there is no need to invent these subgoals manually here, as these proof obligations show up automatically during the course of the proof. We only give the induction steps $(x_1 \mapsto (S\ x_1))$ of the first two inductions and omit the base cases $(x_1 = 0)$. A similar proof can also be generated by existing induction theorem provers like *ACL-2*.

$$\begin{aligned} & f'\ (S\ x_1) \\ =\ & sub\ (f'\ x_1)\ (S^2\ 0)\ (0:[\,]) \\ =\ & sub\ (q\ x_1)\ (S^2\ 0)\ (0:[\,]) && (IH1) \\ =\ & (q'\ x_1):(q\ x_1) && (IH2) \\ =\ & q\ (S\ x_1) \end{aligned}$$

$$\begin{aligned} & sub\ (q\ (S\ x_1))\ (S^2\ 0)\ (0:[\,]) \\ =\ & sub\ ((q'\ x_1):(q\ x_1))\ (S^2\ 0)\ (0:[\,]) \\ =\ & (sub\ (q'\ x_1)\ (S^2\ 0)\ (0:[\,])):(sub\ (q\ x_1)\ (S^2\ 0)\ (0:[\,])) \\ =\ & (sub\ (q'\ x_1)\ (S^2\ 0)\ (0:[\,])):((q'\ x_1):(q\ x_1)) && (IH2) \\ =\ & (S^2\ (q'\ x_1)):((q'\ x_1):(q\ x_1)) && (IH3) \\ =\ & (q'\ (S\ x_1)):(q\ (S\ x_1)) \end{aligned}$$

In this paper we consider the definition of f in p_{acc} as a *macro tree transducer* (for short *mtt*) [8, 9, 11] with one function: in general, such an f is defined by case analysis on the root symbol of the recursion argument t. The right-hand side of an equation for f may only contain (*extended*) *primitive-recursive function calls*, i.e., the recursion argument of f has to be a variable that refers to a subtree of t. The functions f' and *sub* together are viewed as a *2-modular tree transducer* (for short *modtt*) [10], where it is allowed that a function in module 1 (here f') calls a function in module 2 (here *sub*) non-primitive-recursively.

[1] For simplicity, we regard an untyped language. When introducing types, one would generate several substitution functions for the different types of arguments.

We slightly modify a *decomposition* technique from [19] that is based on results in [8–10] and transforms mtts like f into modtts like f' and *sub* without accumulators. Unfortunately, it turns out that the new programs are still not suitable for automatic verification. Since their verification problems are caused only by the form of the new initial calls, we suggest another transformation step, called *constructor replacement*, which yields initial calls of the innocuous form $(f'\ x_1)$ without initial values like 0 and $[\,]$.

Since the class of mtts contains not only tail-recursive programs, but also programs with nested recursion, we will demonstrate by examples that our transformation can not only be useful for functions resulting from the translation of imperative programs, but for accumulative functional programs in general!

Besides this introduction, the paper contains four further sections and two appendices. In Sect. 2 we fix the required notions and notations and introduce our functional language and tree transducers. Sect. 3 presents the deaccumulation technique. Sect. 4 compares our technique to related work. Finally, Sect. 5 contains future research topics. Two additional examples demonstrating the application of our approach can be found in the appendices.

2 Preliminaries and Language

For every natural number $m \in \mathbb{N}$, $[m]$ denotes the set $\{1, \ldots, m\}$. We use the sets $X = \{x_1, x_2, x_3, \ldots\}$ and $Y = \{y_1, y_2, y_3, \ldots\}$ 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, $X_0 = Y_0 = \emptyset$.

A *ranked alphabet* (C, rank) consists of a finite set C and a mapping $\mathit{rank} : C \to \mathbb{N}$ where $\mathit{rank}(c)$ is the *arity* of c. We define $C^{(n)} = \{c \in C \mid \mathit{rank}(c) = n\}$. The set of *trees* (or ground *terms*) *over* C, denoted by T_C, is the smallest subset $T \subseteq (C \cup \{(\} \cup \{)\})^*$ with $C^{(0)} \subseteq T$ and for every $c \in C^{(n)}$ with $n \in \mathbb{N} - \{0\}$ and $t_1, \ldots, t_n \in T$: $(c\ t_1 \ldots t_n) \in T$. For a term t, pairwise distinct variables $x_1, \ldots, x_n$, and terms $t_1, \ldots, t_n$, we denote by $t[x_1/t_1, \ldots, x_n/t_n]$ the term that is obtained from t by substituting every occurrence of x_j in t by t_j. We abbreviate $[x_1/t_1, \ldots, x_n/t_n]$ by $[x_j/t_j]$, if the involved variables and terms are clear.

We consider a simple first-order, constructor-based functional programming language P as source and target language for the transformations. Every program $p \in P$ consists of several modules. In every module a function is defined by complete case analysis on the first argument (*recursion argument*) via pattern matching, where only flat patterns of the form $(c\ x_1 \ldots x_k)$ for constructors c and variables x_i are allowed. The other arguments are called *context arguments*. If, in a right-hand side of a function definition, there is a call of the same function, then the first argument of this function call has to be a subtree x_i of the first argument in the corresponding left-hand side. To ease readability, we choose an untyped ranked alphabet C_p of constructors, which is used to build up input 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.

Definition 1 Let C and F be ranked alphabets of *constructors* and *defined function symbols*, respectively, such that $F^{(0)} = \emptyset$, and X, Y, C, F are pairwise disjoint. We define the sets P, M, R of *programs*, *modules*, and *right-hand sides* as follows. Here, p, m, r, c, f range over the sets P, M, R, C, F, respectively.

$$\begin{array}{lll} p & ::= & m_1 \dots m_l & \text{(program)} \\ m & ::= & f\ (c_1\ x_1 \dots x_{k_1})\ y_1 \dots y_n = r_1 & \text{(module)} \\ & & \dots \\ & & f\ (c_q\ x_1 \dots x_{k_q})\ y_1 \dots y_n = r_q & \\ r & ::= & x_i \mid y_j \mid c\ r_1 \dots r_k \mid f\ r_0\ r_1 \dots r_n & \text{(right-hand side)} \end{array}$$

The sets of constructors, defined functions, and modules that occur in $p \in P$ are denoted by C_p, F_p, and M_p, respectively. For every $f \in F_p$, there is exactly one $m \in M_p$ such that f is defined in m. Then, f is also denoted by f_m. For every $f \in F_p^{(n+1)}$ and $c \in C_p^{(k)}$, there is exactly one equation of the form

$$f\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = rhs_{p,f,c}$$

with $rhs_{p,f,c} \in RHS(f, C_p \cup F_p - \{f\}, X_k, Y_n)$, where for every $f \in F$, $C' \subseteq C \cup F$, and $k, n \in \mathbb{N}$, $RHS(f, C', X_k, Y_n)$ is the smallest set RHS satisfying:

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

Note that, in addition to constructors, defined function symbols may also be contained in the second argument C' of RHS in the previous definition. The functions in C' may then be called with arbitrary arguments in right-hand sides, whereas in recursive calls of f, the recursion argument must be an x_i.

Example 2 Consider the programs p_{acc} and p_{non} from the introduction:

- $p_{acc} \in P$, where $M_{p_{acc}}$ contains one module $m_{acc,f}$ with the definition of f.
- $p_{non} \in P$, $M_{p_{non}}$ contains modules $m_{non,f'}, m_{non,sub}$ defining f' and sub. □

Now, we introduce the classes of tree transducers relevant for this paper. Since in our language every module defines exactly one function, to simplify the presentation we also project this restriction on tree transducers. In the literature, more general classes of *macro tree transducers* [8, 9] and *modular tree transducers* [10] are studied, which allow mutual recursion. Our transformation could also handle these classes. In contrast to the literature, we include an *initial call* r in the definition of tree transducers, which has the form of a right-hand side.

Definition 3 Let $p \in P$.

- A pair (m, r) with $m \in M_p$ and $r \in RHS(f_m, C_p, X_1, Y_0)$ is called a *one-state macro tree transducer of* p (for short *1-mtt* of p), if for every $c \in C_p^{(k)}$ we have $rhs_{p,f_m,c} \in RHS(f_m, C_p, X_k, Y_n)$, where $f_m \in F_p^{(n+1)}$.
  Thus, the function f_m from module m may call itself in a primitive-recursive way, but it does not call any functions from other modules. Moreover, the initial call r is a term built from f_m, constructors, and the variable x_1 as first argument of all subterms rooted with f_m.

– A triple (m_1, m_2, r) with $m_1, m_2 \in M_p$ is called *homomorphism-substitution modular tree transducer of* p (for short *hsmodtt* of p), if there are $n \in \mathbb{N}$ and pairwise distinct *substitution constructors* $\pi_1, \ldots, \pi_n \in C_p^{(0)}$, such that:
  1. $f_{m_1} \in F_p^{(1)}$ and $f_{m_2} = sub \in F_p^{(n+1)}$,
  2. for every $c \in C_p^{(k)}$ we have $rhs_{p,f_{m_1},c} \in RHS(f_{m_1}, C_p \cup \{sub\}, X_k, Y_0)$,
  3. m_2 contains the equations
     $sub\ \pi_j \quad y_1 \ldots y_n = y_j\,, \qquad$ for every $j \in [n]$
     $sub\ (c\ x_1 \ldots x_k) \quad y_1 \ldots y_n = c\ (sub\ x_1\ y_1 \ldots y_n) \ldots (sub\ x_k\ y_1 \ldots y_n)\,,$
     for every $c \in (C_p - \{\pi_1, \ldots, \pi_n\})^{(k)}$
  4. $r \in RHS(f_{m_1}, (C_p - \{\pi_1, \ldots, \pi_n\}) \cup \{sub\}, X_1, Y_0)$.

  Thus, the function from the module m_1 is unary. In its right-hand sides, it may call itself primitive-recursively and it may call the function *sub* from the module m_2 with arbitrary arguments. The function *sub* has the special form of a *substitution function*, where $(sub\ t\ s_1 \ldots s_n)$ replaces all occurrences of the substitution constructors $\pi_1, \ldots, \pi_n$ in t by $s_1, \ldots, s_n$, respectively. The initial call r is as for 1-mtts, but it may also contain *sub*, whereas the substitution constructors $\pi_1, \ldots, \pi_n$ may not appear in it.
– A 1-mtt (m, r) of p is called *nullary constructor disjoint* (for short *ncd*), if there are pairwise different nullary constructors $c_1, \ldots, c_n \in C_p^{(0)}$, such that $r = (f_m\ x_1\ c_1 \ldots c_n)$ and $c_1, \ldots, c_n$ do not occur in right-hand sides of m.
  An hsmodtt (m_1, m_2, r) of p is called *ncd*, if $r = (sub\ (f_{m_1}\ x_1)\ c_1 \ldots c_n)$ with pairwise different $c_1, \ldots, c_n \in C_p^{(0)} - \{\pi_1, \ldots, \pi_n\}$ that do not occur in right-hand sides of m_1.
– An hsmodtt (m_1, m_2, r) of p is *initial value free* (*ivf*), if $r = (f_{m_1}\ x_1)$. □

Example 4 (Ex. 2 continued)

– $(m_{acc,f}, r_{acc})$ with initial call $r_{acc} = (f\ x_1\ 0\ [\,])$ is a 1-mtt of p_{acc} that is ncd.
– Our transformation consists of the two steps "decomposition" and "constructor replacement". Decomposition transforms p_{acc} into the following program $p_{dec} \in P$, which contains the modules $m_{dec,f'}$ and $m_{dec,sub}$:

$$\begin{array}{lll} f'\ (S\ x_1) & = & sub\ (f'\ x_1)\ (S\ (S\ \pi_1))\ (\pi_1 : \pi_2) \\ f'\ 0 & = & \pi_2 \end{array}$$

$$\begin{array}{llll} sub\ (x_1 : x_2)\ y_1\ y_2 & = & (sub\ x_1\ y_1\ y_2) : (sub\ x_2\ y_1\ y_2) & \qquad sub\ [\,]\ y_1\ y_2 = [\,] \\ sub\ (S\ x_1) & y_1\ y_2 & = S\ (sub\ x_1\ y_1\ y_2) & \qquad sub\ \pi_1\ y_1\ y_2 = y_1 \\ sub\ 0 & y_1\ y_2 & = 0 & \qquad sub\ \pi_2\ y_1\ y_2 = y_2 \end{array}$$

  Here, $(m_{dec,f'}, m_{dec,sub}, r_{dec})$ with the initial call $r_{dec} = (sub\ (f'\ x_1)\ 0\ [\,])$ is an hsmodtt of p_{dec} that is ncd, but not ivf.
– $(m_{non,f'}, m_{non,sub}, r_{non})$ with $r_{non} = (f'\ x_1)$ and the modules from the introduction is an hsmodtt of p_{non} that is ivf ($n = 2$, $\pi_1 = 0$, $\pi_2 = [\,]$). □

For every program $p \in P$, its evaluation is described by a (nondeterministic) reduction relation $\Rightarrow_p$ on $T_{C_p \cup F_p}$. As usual, $\Rightarrow_p^n$ and $\Rightarrow_p^*$ denote the n-fold composition and the transitive, reflexive closure of $\Rightarrow_p$, respectively. If $t \Rightarrow_p^* t'$ and

there is no t'' such that $t' \Rightarrow_p t''$, then t' is called a *normal form of* t, which is denoted by $nf_p(t)$, if it exists and is unique. It can be proved in analogy to [10] that for every program $p \in P$, hsmodtt (m_1, m_2, r) of p (and 1-mtt (m, r) of p), and $t \in T_{\{f_{m_1}, f_{m_2}\} \cup C_p}$ (and $t \in T_{\{f_m\} \cup C_p}$, respectively), there exists a unique normal form $nf_p(t)$. In particular, for every $t \in T_{C_p}$ the normal form $nf_p(r[x_1/t])$ exists. The proof is based on the result that for every modtt and mtt the corresponding reduction relation is terminating and confluent. The normal form $nf_p(r[x_1/t])$ is called the *output tree* computed for the *input tree* t.

3 Deaccumulation

To improve verifiability we transform accumulative programs into non-accumulative programs by translating 1-mtts into hsmodtts. The defined functions of the resulting programs have no context arguments at all or they have context arguments that are not accumulating. Moreover, the resulting initial calls have no initial values in context argument positions. The transformation proceeds in two steps: "decomposition" (Sect. 3.1) and "constructor replacement" (Sect. 3.2).

3.1 Decomposition

In [8–10] it was shown that every mtt (with possibly several functions of arbitrary arity) can be decomposed into a *top-down tree transducer* (an mtt with unary functions only) plus a substitution device. In this paper, we use a modification of this result, integrating the constructions of Lemmata 21 and 23 of [19]. The key idea is to simulate an $(n+1)$-ary function f by a new unary function f'. To this end, all context arguments are deleted and only the recursion argument is maintained. Since f' does not know the current values of its context arguments, it uses a new constructor π_j, whenever f uses its j-th context argument. For this purpose, every occurrence of y_j in the right-hand sides of equations for f is replaced by π_j. The current context arguments themselves are integrated into the calculation by replacing every occurrence of the form $(f\ x_i \ldots)$ in a right-hand side or in the initial call by $(sub\ (f'\ x_i) \ldots)$. Here, the new function sub is a substitution function. As explained before, $(sub\ t\ s_1 \ldots s_n)$ replaces every π_j in the first argument t of sub by the j-th context argument s_j.

Lemma 5 For every $p \in P$ and 1-mtt (m, r) of p, there are $p' \in P$ and an hsmodtt (m_1, m_2, r') of p' such that for every $t \in T_{C_p}$: $nf_p(r[x_1/t]) = nf_{p'}(r'[x_1/t])$. Additionally, if (m, r) is ncd, then (m_1, m_2, r') is ncd, too.

Proof. We construct $p' \in P$ by adding modules m_1 and m_2 to p, and we construct r' from r. Let $n \in \mathbb{N}$, $f = f_m \in F_p^{(n+1)}$, $f' \in (F - F_p)^{(1)}$, $sub \in (F - F_p)^{(n+1)}$ with $sub \neq f'$, and pairwise distinct $\pi_1, \ldots, \pi_n \in (C - C_p)^{(0)}$.

1. For every $c \in C_p^{(k)}$ and for every equation $f\ (c\ x_1 \ldots x_k)\ y_1 \ldots y_n = rhs_{p,f,c}$ in m, the module m_1 contains $f'\ (c\ x_1 \ldots x_k) = \underline{dec}(rhs_{p,f,c})$, where $\underline{dec} : RHS(f, C_p, X_k, Y_n) \longrightarrow RHS(f', C_p \cup \{sub\} \cup \{\pi_1, \ldots, \pi_n\}, X_k, Y_0)$ with:

$$\begin{aligned}
\underline{dec}(f\ x_i\ r_1 \dots r_n) &= sub\ (f'\ x_i)\ \underline{dec}(r_1) \dots \underline{dec}(r_n)\,, \\
&\qquad \text{for all } i \in [k], r_1, \dots, r_n \in RHS(f, C_p, X_k, Y_n) \\
\underline{dec}(c'\ r_1 \dots r_a) &= c'\ \underline{dec}(r_1) \dots \underline{dec}(r_a)\,, \\
&\qquad \text{for all } c' \in C_p^{(a)}, r_1, \dots, r_a \in RHS(f, C_p, X_k, Y_n) \\
\underline{dec}(y_j) &= \pi_j\,, \quad \text{for all } j \in [n]
\end{aligned}$$

For every $j \in [n]$, m_1 contains a dummy-equation $f'\ \pi_j = \pi_j$.

2. m_2 contains the equations

$$\begin{aligned}
&sub\ (c\ x_1 ... x_k)\ y_1 ... y_n = c\ (sub\ x_1\ y_1 ... y_n) ... (sub\ x_k\ y_1 ... y_n)\,, && \text{for all } c \in C_p^{(k)} \\
&sub\ \pi_j \qquad\quad y_1 ... y_n = y_j\,, && \text{for all } j \in [n]
\end{aligned}$$

3. $r' = \underline{dec}(r)$.

Note that (m_1, m_2, r') is an hsmodtt of p'. Moreover, for every $t \in T_{C_p}$, we have $nf_p(r[x_1/t]) = nf_{p'}(r'[x_1/t])$. For the proof of this statement, the following statements $(*)$ and $(**)$ are proved by simultaneous induction (cf., e.g., [9, 11, 25]). For space reasons we omit this proof.
$(*)$ For every $t \in T_{C_p}$ and $s_1, \dots, s_n \in T_{C_p \cup \{\pi_1, \dots, \pi_n\}}$:
$nf_p(f\ t\ s_1 \dots s_n) = nf_{p'}(sub\ (f'\ t)\ s_1 \dots s_n)$.
$(**)$ For every $k \in \mathbb{N}$, $t_1, \dots, t_k \in T_{C_p}$, $\bar{r} \in RHS(f, C_p, X_k, Y_n)$, and $s_1, \dots, s_n \in T_{C_p \cup \{\pi_1, \dots, \pi_n\}}$: $nf_p(\bar{r}[x_j/t_j][y_j/s_j]) = nf_{p'}(sub\ (\underline{dec}(\bar{r})[x_j/t_j])\ s_1 \dots s_n)$.

Moreover, if (m, r) is ncd, then there are pairwise different $c_1, \dots, c_n \in C_p^{(0)}$ such that $r = (f\ x_1\ c_1 \dots c_n)$ and $c_1, \dots, c_n$ do not occur in right-hand sides of m. Thus, $r' = (sub\ (f'\ x_1)\ c_1 \dots c_n)$ and by the definition of $\underline{dec}$, $c_1, \dots, c_n$ are not introduced into right-hand sides of m_1. Hence, (m_1, m_2, r') is ncd, too. □

Example 6 Decomposition translates the 1-mtt $(m_{acc,f}, r_{acc})$ of p_{acc} into the hsmodtt $(m_{dec,f'}, m_{dec,sub}, r_{dec})$ of p_{dec}, which are both ncd, cf. Ex. 4. □

However, we have not yet improved the automatic verifiability of programs:

Example 7 Let $(m_{dec,f'}, m_{dec,sub}, r_{dec})$ be the hsmodtt of p_{dec} created by decomposition and resume the proof attempt from the introduction. Since the initial call has changed from $(f\ x_1\ 0\ [\,])$ to $(sub\ (f'\ x_1)\ 0\ [\,])$, we have to prove $(sub\ (f'\ x_1)\ 0\ [\,]) = (q\ x_1)$ by induction. Again, the automatic proof fails, because in the induction step $(x_1 \mapsto (S\ x_1))$ the induction hypothesis cannot be successfully applied to prove $(sub\ (f'\ (S\ x_1))\ 0\ [\,]) = (q\ (S\ x_1))$. The problem is that the context arguments of $(sub\ (f'\ x_1)\ \underline{(S\ (S\ \pi_1))}\ \overline{(\pi_1 : \pi_2)})$, which originates as subterm from rule application to $(sub\ \overline{(f'\ (S\ x_1))}\ \overline{0\ [\,]})$, do not fit to the context arguments of the term $(sub\ (f'\ x_1)\ \underline{0}\ \underline{[\,]})$ in the induction hypothesis. □

3.2 Constructor Replacement

We solve the above problem by avoiding applications of substitution functions (with specific context arguments like 0 and $[\,]$ in Ex. 7) in initial calls. Since then an initial call consists only of a unary function, induction hypotheses can

be applied without paying attention to context arguments. The idea, illustrated on Ex. 7, is to replace the substitution constructors π_1 and π_2 by 0 and [ ] from the initial call. Thus, the initial values of *sub*'s context arguments are encoded into the program and the substitution in the initial call becomes superfluous.

We restrict ourselves to 1-mtts that are ncd. Then, after decomposition, the initial calls have the form $(sub\ (f\ x_1)\ c_1 \dots c_n)$, where $c_1, \dots, c_n$ are pairwise different. Thus, when replacing each π_j by c_j, there is a unique correspondence between the nullary constructors $c_1, \dots, c_n$ and the substitution constructors $\pi_1, \dots, \pi_n$. In Ex. 10 we will demonstrate the problems with identical $c_1, \dots, c_n$.

When replacing π_j by c_j, the constructors $c_1, \dots, c_n$ now have two roles: If c_j occurs within a first argument of *sub*, then it acts like the former substitution constructor π_j, i.e., it will be substituted by the j-th context argument of *sub*. Thus, *sub* now has the defining equation $sub\ c_j\ y_1 \dots y_n = y_j$. Only occurrences of c_j outside of *sub*'s first argument are left unchanged, i.e., here the constructor c_j stands for its original value. To make sure that there is no conflict between these two roles of c_j, we again need the ncd-condition. It ensures that originally, c_j did not occur in right-hand sides of f's definition. Then the only occurrence of c_j, which does not stand for the substitution constructor π_j, is as context argument of *sub* in the initial call. This substitution, however, can be omitted, because the call $(sub\ (f\ x_1)\ c_1 \dots c_n)$ would now just mean to replace every c_j in $(f\ x_1)$ by c_j. In this way, the resulting hsmodtt is initial value free (ivf).

Lemma 8 Let $p \in P$ and (m_1, m_2, r) be an hsmodtt of p as constructed in the transformation of Lemma 5. Moreover, let (m_1, m_2, r) be ncd and $\pi_1, \dots, \pi_n$ be its substitution constructors. Then, there are $p' \in P$ and an hsmodtt (m_1', m_2', r') of p' that is ivf, such that for all $t \in T_{C_p - \{\pi_1, \dots, \pi_n\}}$: $nf_p(r[x_1/t]) = nf_{p'}(r'[x_1/t])$.

Proof. We construct $p' \in P$ by replacing m_1 and m_2 in p by modules m_1' and m_2', and we define r'. Let $f = f_{m_1} \in F_p^{(1)}$, $sub = f_{m_2} \in F_p^{(n+1)}$, and $c_1, \dots, c_n \in C_p^{(0)} - \{\pi_1, \dots, \pi_n\}$ be pairwise distinct, such that $r = (sub\ (f\ x_1)\ c_1 \dots c_n)$ and $c_1, \dots, c_n$ do not occur in right-hand sides of m_1. Let $C_{p'} = C_p - \{\pi_1, \dots, \pi_n\}$.

1. For every $c \in C_{p'}^{(k)}$ and for every equation $f\ (c\ x_1 \dots x_k) = rhs_{p,f,c}$ in m_1, the module m_1' contains $f\ (c\ x_1 \dots x_k) = \underline{repl}(rhs_{p,f,c})$, where $\underline{repl} : RHS(f, (C_p - \{c_1, \dots, c_n\}) \cup \{sub\}, X_k, Y_0) \to RHS(f, C_{p'} \cup \{sub\}, X_k, Y_0)$ replaces every occurrence of π_j by c_j, for all $j \in [n]$.
2. m_2' contains the equations

$$\begin{aligned} &sub\ (c\ x_1 \dots x_k)\ y_1 \dots y_n = c\ (sub\ x_1\ y_1 \dots y_n) \dots (sub\ x_k\ y_1 \dots y_n)\,, && \\ & && \text{for all } c \in C_{p'}^{(k)} - \{c_1, \dots, c_n\} \\ &sub\ c_j \qquad y_1 \dots y_n = y_j\,, && \text{for all } j \in [n] \end{aligned}$$

3. $r' = f\ x_1$.

Note that (m_1', m_2', r') is an hsmodtt of p' that is ivf. For every $t \in T_{C_{p'}}$, we have $nf_p(r[x_1/t]) = nf_{p'}(r'[x_1/t])$. For the proof of this statement, the following statements $(*)$ and $(**)$ are proved by simultaneous induction. For space reasons

we omit this proof.
($*$) For every $t \in T_{C_{p'}}$ and $s_1, \ldots, s_n \in T_{C_{p'}}$:
$nf_p(sub\ (f\ t)\ s_1 \ldots s_n) = nf_{p'}(sub\ (f\ t)\ s_1 \ldots s_n)$.
($**$) For every $k \in \mathbb{N}$, $t_1, \ldots, t_k \in T_{C_{p'}}$,
$\bar{r} \in RHS(f, (C_p - \{c_1, \ldots, c_n\}) \cup \{sub\}, X_k, Y_0)$, and $s_1, \ldots, s_n \in T_{C_{p'}}$:
$nf_p(sub\ (\bar{r}[x_j/t_j])\ s_1 \ldots s_n) = nf_{p'}(sub\ (\underline{repl}(\bar{r})[x_j/t_j])\ s_1 \ldots s_n)$. □

Example 9 Constructor replacement translates the ncd hsmodtt $(m_{dec,f'}, m_{dec,sub}, r_{dec})$ of p_{dec} into the ivf hsmodtt $(m_{non,f'}, m_{non,sub}, r_{non})$ of p_{non}. Essentially, all occurrences of π_1 and π_2 are replaced by 0 and [ ]. □

Now we demonstrate the problems with hsmodtts violating the condition ncd:

Example 10 Assume that p_{acc} and r_{acc} are changed into the following program:

$$\begin{array}{lll} f\ (S\ x_1)\ y_1\ y_2 & = & f\ x_1\ (S\ (S\ y_1))\ (y_1 + y_2) \\ f\ 0 \qquad\ y_1\ y_2 & = & y_2 \end{array}$$

and the initial call $(f\ x_1\ 0\ 0)$, computing the sum of the first x_1 even numbers. Now the same constructor 0 occurs in the initial values for both context arguments. Decomposition delivers the program:[2]

$$\begin{array}{lll} f'\ (S\ x_1) & = & sub\ (f'\ x_1)\ (S\ (S\ \pi_1))\ (\pi_1 + \pi_2) \\ f'\ 0 \qquad & = & \pi_2 \end{array}$$

$$\begin{array}{llll} sub\ (x_1 + x_2)\ y_1\ y_2 & = & (sub\ x_1\ y_1\ y_2) + (sub\ x_2\ y_1\ y_2) & \qquad sub\ \pi_1\ y_1\ y_2 = y_1 \\ sub\ (S\ x_1) \quad y_1\ y_2 & = & S\ (sub\ x_1\ y_1\ y_2) & \qquad sub\ \pi_2\ y_1\ y_2 = y_2 \\ sub\ 0 \qquad\ y_1\ y_2 & = & 0 & \end{array}$$

and initial call $(sub\ (f'\ x_1)\ 0\ 0)$. Constructor replacement would replace π_1 and π_2 by 0, which leads to different rules $sub\ 0\ y_1\ y_2 = y_1$ and $sub\ 0\ y_1\ y_2 = y_2$ with same left-hand side. In Sect. 5 we give an idea how to overcome this problem. □

We conclude this section with some statements about substitution functions which are often helpful for the verification of transformed programs (cf. the examples in Sect. 4 and App. A and B). Instead of proving these statements during verification, they should be generated during program transformation. This is possible because the substitution functions only depend on the set of constructors but not on the transformed function.

Lemma 11 Let $p \in P$ and (m_1, m_2, r) be an hsmodtt of p with substitution constructors $c_1, \ldots, c_n$ and substitution function sub.

1. A_{sub} (Associativity of sub). For every $t_0, t_1, ..., t_n, s_1, ..., s_n \in T_{C_p}$ we have $nf_p(sub\ (sub\ t_0\ t_1 ... t_n)\ s_1 ... s_n) = nf_p(sub\ t_0\ (sub\ t_1\ s_1 ... s_n) ... (sub\ t_n\ s_1 ... s_n))$.
2. U_{sub} (Right Units of sub). For every $t \in T_{C_p}$ we have $nf_p(sub\ t\ c_1 \ldots c_n) = t$.
3. $+_{sub}$ (Addition by sub). If $n = 1$, $C_p = \{S, 0\}$, and $nf_p((S^{z_1}\ 0) + (S^{z_2}\ 0)) = S^{z_1+z_2}\ 0$ for all $z_1, z_2 \in \mathbb{N}$, then $nf_p(sub\ s\ t) = nf_p(s + t)$ for all $s, t \in T_{C_p}$.

Proof. The proofs are straightforward inductions on T_{C_p} and $\mathbb{N}$, respectively. □

[2] During the transformation, + is treated as an ordinary binary constructor.

4 Related Work

Program transformations are a well-established field in software engineering and compiler construction (see, e.g., [2, 6, 21, 22]). However, we suggested a novel application area for program transformations by applying them in order to increase *verifiability*. This goal is often in contrast to the classical aim of increasing efficiency, since a more efficient program is usually harder to verify. In particular, while *composition results* from the theory of tree transducers are usually applied in order to improve the efficiency of functional programs (cf., e.g., [18, 19, 24, 25]), we have demonstrated that also the corresponding *decomposition results* are not only of theoretical interest.

Program transformations that improve verifiability have rarely been investigated before. A first step into this direction was taken in [12]. There, two transformations were presented that can remove accumulators. They are based on the associativity and commutativity of auxiliary functions like + occurring in accumulator arguments. The advantage of the approach in [12] is that it does not require the strict syntactic restrictions of 1-mtts that are ncd. Moreover, [12] does not require that functions from other modules may not be called in right-hand sides. Because of that restriction, in the present paper, we have to treat all auxiliary functions like + as constructors and exclude the use of any information about these functions during the transformation.

On the other hand, the technique of [12] can essentially only remove *one* accumulator argument (e.g., in contrast to our method, it cannot eliminate both accumulators of p_{acc}). Moreover, the approach in [12] relies on knowledge about auxiliary functions like +. Hence, it is not applicable if the context of accumulator arguments on the right-hand side is not associative or commutative. Thus, it fails on examples like the following program p_{exp}. In particular, this demonstrates that in contrast to [12], our technique can also handle nested recursion. Indeed, deaccumulation is useful for functional programs in general — not just for functions resulting from translating imperative programs.

$$\begin{array}{lll} exp\ (S\ x_1)\ y_1 & = & exp\ x_1\ (exp\ x_1\ y_1) \\ exp\ 0 \qquad\ y_1 & = & S\ y_1 \end{array}$$

The initial call is $(exp\ x_1\ 0)$. We want to prove $(exp\ x_1\ 0) = (e\ x_1)$, where $(e\ (S^n\ 0))$ computes $(S^{2^n}\ 0)$, see below. Here, $(S^{z_1}\ 0) + (S^{z_2}\ 0)$ computes $S^{z_1+z_2}\ 0$.

$$\begin{array}{lll} e\ (S\ x_1) & = & (e\ x_1)\ +\ (e\ x_1) \\ e\ 0 \qquad & = & S\ 0 \end{array}$$

Since *exp* is a 1-mtt that is ncd, deaccumulation delivers the program:

$$\begin{array}{lll} exp'\ (S\ x_1) & = sub\ (exp'\ x_1)\ (sub\ (exp'\ x_1)\ 0) & sub\ (S\ x_1)\ y_1 = S\ (sub\ x_1\ y_1) \\ exp'\ 0 \qquad & = S\ 0 & sub\ 0 \qquad y_1 = y_1 \end{array}$$

and the initial call $(exp'\ x_1)$, which are better suited for induction provers, because there are no accumulating arguments anymore. For instance, instead of

proving $(exp\ x_1\ 0) = (e\ x_1)$ for the original program (which requires a generalization), now the statement $(exp'\ x_1) = (e\ x_1)$ (taken as induction hypothesis IH) can be proved automatically. We only show the induction step $(x_1 \mapsto (S\ x_1))$.

$$\begin{array}{lcll} exp'\ (S\ x_1) & = & sub\ (exp'\ x_1)\ (sub\ (exp'\ x_1)\ 0) & \\ & = & sub\ (e\ x_1)\ (sub\ (e\ x_1)\ 0) & (2 * IH) \\ & = & sub\ (e\ x_1)\ (e\ x_1) & (U_{sub}) \\ & = & (e\ x_1)\ +\ (e\ x_1) & (+_{sub}) \\ & = & e\ (S\ x_1) & \end{array}$$

While in many examples generalizations can be avoided by our technique, it does not render generalization techniques superfluous. There exist accumulative functions where our transformation is not applicable, cf. Ex. 10 [3], and even if it is applicable, there may still be conjectures that can only be proved via a suitable generalization. However, even then our transformation is advantageous, because the generalizations for the transformed functions are usually much easier than the ones required for the original accumulative functions (cf. App. A).

5 Conclusion and Future Work

Imperative programs and accumulative functional programs resulting from their translation are hard to verify with induction provers. Therefore, we introduced an automatic technique that transforms accumulative functions into non-accumulative functions, whose verification is often significantly easier with existing proof tools. However, it remains to characterize (at least informally) the class of verification problems, for which there is a real improvement.

To increase the applicability of our approach, we plan to extend it to more general forms of algorithms. For example, the requirement ncd should be weakened, such that examples with equal constructors in initial calls can be handled as well. The idea is to use different substitution functions such that at every node of a tree it can be read from the substitution function, how a nullary constructor has to be substituted. To this end, one must analyze the decomposed program prior to constructor replacement to find out which substitution constructors can occur in which contexts. For instance, in Ex. 10 it can be shown that π_1 can only occur in a left subtree of a $+$, whereas π_2 cannot occur in such positions. Thus, in the program after constructor replacement every occurrence of a 0 in a left subtree of a $+$ must be substituted by y_1, whereas all other occurrences must be substituted by y_2.

An extension beyond mtts seems to be possible as well. For example, the requirement of flat patterns on left-hand sides may be relaxed. Moreover, one could consider different constructor *terms* instead of nullary constructors in initial calls. Further extensions include a decomposition that only removes those context arguments from a function that are modified in recursive calls. Finally, we also investigate how to incorporate the transformations of [12] into our approach.

[3] Note that for this example, however, one can construct an equivalent non-accumulative program, cf. Sect. 5.

References

1. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Inka 5.0 - A logical voyager. In *Proc. CADE-16*, LNAI 1632, pages 207–211, 1999.
2. F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, 1982.
3. A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14:189–235, 1995.
4. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
5. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 63:185–253, 1993.
6. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.
7. E. W. Dijkstra. Invariance and non-determinacy. In *Mathematical Logic and Programming Languages*, chapter 9, pages 157–165. Prentice-Hall, 1985.
8. J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R. V. Book (ed.), *Formal language theory; perspectives and open problems*, pages 241–286. Academic Press, 1980.
9. J. Engelfriet and H. Vogler. Macro tree transducers. *JCSS*, 31:71–145, 1985.
10. J. Engelfriet and H. Vogler. Modular tree transducers. *TCS*, 78:267–304, 1991.
11. Z. Fülöp and H. Vogler. *Syntax-directed semantics — Formal models based on tree transducers.* Monographs in Theoretical Comp. Science, EATCS. Springer, 1998.
12. J. Giesl. Context-moving transformations for function verification. In *Proc. LOPSTR'99*, LNCS 1817, pages 293–312, 2000.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
14. A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming*, 9:225–245, 1999.
15. A. Ireland and J. Stark. On the automatic discovery of loop invariants. In *4th NASA Langley Formal Methods Workshop.* NASA Conf. Publication 3356, 1997.
16. D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *Journal of Computer and Mathematics with Applications*, 29:91–114, 1995.
17. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer, 2000.
18. A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In *Proc. FST &TCS'98*, LNCS 1530, pages 146–157, 1998.
19. A. Kühnemann, R. Glück, K. Kakehi. Relating accumulative and non-accumulative functional programs. In *Proc. RTA'01*, LNCS 2051, pages 154–168, 2001.
20. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.
21. H. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.
22. A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28:360–414, 1996.
23. J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In *Proc. LOPSTR'98*, LNCS 1559, pages 271–288, 1998.
24. J. Voigtländer. Conditions for efficiency improvement by tree transducer composition. In *Proc. RTA'02*, LNCS 2378, pages 222–236, 2002.
25. J. Voigtländer and A. Kühnemann. Composition of functions with accumulating parameters. To appear in *Journal of Functional Programming*, 2004.
26. C. Walther. Mathematical induction. In Gabbay, Hogger, Robinson (eds.), *Handbook of Logic in AI & Logic Prog., Vol. 2,* 127-228. Oxford University Press, 1994.

A Example: Splitting Monadic Trees

The program
$$\begin{array}{lll} \mathit{split}\ (A\ x_1)\ y_1 & = & A\ (\mathit{split}\ x_1\ y_1) \\ \mathit{split}\ (B\ x_1)\ y_1 & = & \mathit{split}\ x_1\ (B\ y_1) \\ \mathit{split}\ N\ \ \ \ \ \ y_1 & = & y_1 \end{array}$$
with initial call $(\mathit{split}\ x_1\ N)$ translates monadic trees with n_1 and n_2 occurrences of the unary constructors A and B, respectively, into the tree $A^{n_1}(B^{n_2}N)$ by accumulating the B's in the context argument of *split*. It is transformed into:

$$\begin{array}{lll} \mathit{split}'\ (A\ x_1) & = & A\ (\mathit{sub}\ (\mathit{split}'\ x_1)\ N) \\ \mathit{split}'\ (B\ x_1) & = & \mathit{sub}\ (\mathit{split}'\ x_1)\ (B\ N) \\ \mathit{split}'\ N & = & N \end{array} \qquad \begin{array}{lll} \mathit{sub}\ (A\ x_1)\ y_1 & = & A\ (\mathit{sub}\ x_1\ y_1) \\ \mathit{sub}\ (B\ x_1)\ y_1 & = & B\ (\mathit{sub}\ x_1\ y_1) \\ \mathit{sub}\ N\ \ \ \ \ \ y_1 & = & y_1 \end{array}$$

with initial call $(\mathit{split}'\ x_1)$. If we want to prove the idempotence of the splitting operation, then the proof for the original program requires a generalization from $(\mathit{split}\ (\mathit{split}\ x_1\ N)\ N) = (\mathit{split}\ x_1\ N)$ to $(\mathit{split}\ (\mathit{split}\ x_1\ (b\ x_2))\ (b\ x_3)) = (\mathit{split}\ x_1\ (b\ (x_2 + x_3)))$, where $(b\ n)$ computes $(B^n\ N)$. Such a generalization is difficult to find. On the other hand, $(\mathit{split}'\ (\mathit{split}'\ x_1)) = (\mathit{split}'\ x_1)$ can be proved automatically. In the step case $(x_1 \mapsto (A\ x_1))$, U_{sub} from Lemma 11 is used to infer $(\mathit{sub}\ (\mathit{split}'\ x_1)\ N) = (\mathit{split}'\ x_1)$. In the step case $(x_1 \mapsto (B\ x_1))$, a straightforward generalization step is required by identifying two common subexpressions in a proof subgoal. More precisely, by applying the induction hypothesis, the induction conclusion is transformed into $(\mathit{split}'\ (\mathit{sub}\ \underline{(\mathit{split}'\ x_1)}\ (B\ N))) = (\mathit{sub}\ (\mathit{split}'\ \underline{(\mathit{split}'\ x_1)})\ (B\ N))$. Now, the two underlined occurrences of $(\mathit{split}'\ x_1)$ are generalized to a fresh variable x, and then the proof works by induction on x.

B Example: Reversing Monadic Trees

The program
$$\begin{array}{lll} \mathit{rev}\ (A\ x_1)\ y_1 & = & \mathit{rev}\ x_1\ (A\ y_1) \\ \mathit{rev}\ (B\ x_1)\ y_1 & = & \mathit{rev}\ x_1\ (B\ y_1) \\ \mathit{rev}\ N\ \ \ \ \ \ y_1 & = & y_1 \end{array}$$
with initial call $(\mathit{rev}\ x_1\ N)$ is transformed into the program

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

with initial call $(\mathit{rev}'\ x_1)$. Taking into account that *sub* is just the concatenation function on monadic trees, the above programs correspond to the efficient and the inefficient reverse function, which have linear and quadratic time-complexity in the size of the input tree, respectively. Thus, this example shows that the aim of our technique contrasts with the aim of classical program transformations, i.e., the efficiency is decreased, but the suitability for verification is improved: If we want to show that the reverse of two concatenated lists is the concatenation of the reversed lists in exchanged order, then the proof of $(\mathit{rev}\ (\mathit{sub}\ x_1\ x_2)\ N) = (\mathit{sub}\ (\mathit{rev}\ x_2\ N)\ (\mathit{rev}\ x_1\ N))$ again requires considerable generalization effort, whereas $(\mathit{rev}'\ (\mathit{sub}\ x_1\ x_2)) = (\mathit{sub}\ (\mathit{rev}'\ x_2)\ (\mathit{rev}'\ x_1))$ can be proved by a straightforward induction on x_1, exploiting U_{sub} and A_{sub} from Lemma 11.